

Spring Cloud

Table of Contents

1. Features	2
2. Release Train Versions	3
3. Cloud Native Applications	4
3.1. Spring Cloud Context: Application Context Services	4
3.2. Spring Cloud Commons: Common Abstractions	10
3.3. Spring Cloud LoadBalancer	23
3.4. Spring Cloud Circuit Breaker	28
3.5. CachedRandomPropertySource	31
3.6. Configuration Properties	31
4. Spring Cloud Config	32
4.1. Quick Start	32
4.2. Spring Cloud Config Server	36
4.3. Serving Alternative Formats	64
4.4. Serving Plain Text	65
4.5. Embedding the Config Server	67
4.6. Push Notifications and Spring Cloud Bus	68
4.7. Spring Cloud Config Client	69
5. Spring Cloud Netflix	75
5.1. Service Discovery: Eureka Clients	75
5.2. Service Discovery: Eureka Server	83
5.3. Configuration properties	87
6. Spring Cloud OpenFeign	88
6.1. Declarative REST Client: Feign	88
6.2. Configuration properties	100
7. Spring Cloud Bus	101
7.1. Quick Start	101
7.2. Bus Endpoints	101
7.3. Addressing an Instance	102
7.4. Addressing All Instances of a Service	103
7.5. Service ID Must Be Unique	103
7.6. Customizing the Message Broker	103
7.7. Tracing Bus Events	103
7.8. Broadcasting Your Own Events	104
7.9. Configuration properties	106
8. Spring Cloud Sleuth	107
8.1. Overview	107
8.2. Features	107
8.3. Adding Sleuth to your Project	110

8.4. How Sleuth works	115
8.5. Sampling	116
8.6. Baggage	117
8.7. Instrumentation	118
8.8. Span lifecycle	118
8.9. Naming spans	121
8.10. Managing Spans with Annotations	122
8.11. Customizations	125
8.12. Sending Spans to Zipkin	131
8.13. Integrations	132
8.14. Log integration	142
8.15. Configuration properties	146
9. Spring Cloud Consul	147
9.1. Install Consul	147
9.2. Consul Agent	147
9.3. Service Discovery with Consul	147
9.4. Distributed Configuration with Consul	154
9.5. Consul Retry	157
9.6. Spring Cloud Bus with Consul	157
9.7. Circuit Breaker with Hystrix	157
9.8. Hystrix metrics aggregation with Turbine and Consul	158
9.9. Configuration Properties	158
10. Spring Cloud Zookeeper	159
10.1. Install Zookeeper	159
10.2. Service Discovery with Zookeeper	160
10.3. Using Spring Cloud Zookeeper with Spring Cloud Netflix Components	162
10.4. Spring Cloud Zookeeper and Service Registry	162
10.5. Zookeeper Dependencies	163
10.6. Spring Cloud Zookeeper Dependency Watcher	167
10.7. Distributed Configuration with Zookeeper	168
11. Spring Boot Cloud CLI	171
11.1. Installation	171
11.2. Running Spring Cloud Services in Development	171
11.3. Writing Groovy Scripts and Running Applications	174
11.4. Encryption and Decryption	175
12. Spring Cloud Security	176
12.1. Quickstart	176
12.2. More Detail	178
12.3. Configuring Authentication Downstream of a Zuul Proxy	181
13. Spring Cloud for Cloud Foundry	183
13.1. Discovery	183

13.2. Single Sign On	184
13.3. Configuration	184
14. Spring Cloud Contract Reference Documentation	185
Legal	185
14.1. Getting Started	185
14.2. Using Spring Cloud Contract	225
14.3. Spring Cloud Contract Features	244
14.4. Maven Project	436
14.5. Gradle Project	453
14.6. Docker Project	465
14.7. Spring Cloud Contract customization	472
14.8. “How-to” Guides	492
15. Spring Cloud Vault	536
15.1. Quick Start	536
15.2. Client Side Usage	538
15.3. Authentication methods	541
15.4. Secret Backends	554
15.5. Database backends	559
15.6. Configure PropertySourceLocator behavior	564
15.7. Service Registry Configuration	565
15.8. Vault Client Fail Fast	566
15.9. Vault Enterprise Namespace Support	566
15.10. Vault Client SSL configuration	566
15.11. Lease lifecycle management (renewal and revocation)	567
16. Spring Cloud Gateway	568
16.1. How to Include Spring Cloud Gateway	568
16.2. Glossary	568
16.3. How It Works	568
16.4. Configuring Route Predicate Factories and Gateway Filter Factories	569
16.5. Route Predicate Factories	570
16.6. GatewayFilter Factories	577
16.7. Global Filters	603
16.8. HttpHeadersFilters	608
16.9. TLS and SSL	609
16.10. Configuration	611
16.11. Route Metadata Configuration	612
16.12. Http timeouts configuration	613
16.13. Reactor Netty Access Logs	616
16.14. CORS Configuration	617
16.15. Actuator API	617
16.16. Troubleshooting	622

16.17. Developer Guide	622
16.18. Building a Simple Gateway by Using Spring MVC or Webflux	626
16.19. Configuration properties	628
17. Spring Cloud Function	629
17.1. Introduction	629
17.2. Getting Started	630
17.3. Programming model	631
17.4. Standalone Web Applications	635
17.5. Standalone Streaming Applications	637
17.6. Deploying a Packaged Function	637
17.7. Functional Bean Definitions	641
17.8. Testing Functional Applications	644
17.9. Dynamic Compilation	647
17.10. Serverless Platform Adapters	649
18. Spring Cloud Kubernetes	663
18.1. Why do you need Spring Cloud Kubernetes?	663
18.2. Starters	663
18.3. DiscoveryClient for Kubernetes	664
18.4. Kubernetes native service discovery	665
18.5. Kubernetes PropertySource implementations	665
18.6. Kubernetes Ecosystem Awareness	678
18.7. Pod Health Indicator	679
18.8. Leader Election	679
18.9. Security Configurations Inside Kubernetes	679
18.10. Service Registry Implementation	681
18.11. Examples	681
18.12. Other Resources	682
18.13. Configuration properties	682
18.14. Building	682
18.15. Contributing	684
19. Spring Cloud GCP	691
19.1. Introduction	691
19.2. Getting Started	691
19.3. Spring Cloud GCP Core	694
19.4. Google Cloud Pub/Sub	698
19.5. Google Cloud Storage	710
19.6. Spring JDBC	713
19.7. Spring Integration	716
19.8. Spring Cloud Stream	724
19.9. Spring Cloud Bus	728
19.10. Spring Cloud Sleuth	730

19.11. Stackdriver Logging	734
19.12. Spring Cloud Config	739
19.13. Spring Data Cloud Spanner	742
19.14. Spring Data Cloud Datastore	773
19.15. Spring Data Reactive Repositories for Cloud Firestore	804
19.16. Cloud Memorystore for Redis	815
19.17. Cloud Identity-Aware Proxy (IAP) Authentication	816
19.18. Google Cloud Vision	818
19.19. Google Cloud BigQuery	823
19.20. Secret Manager	826
19.21. Cloud Foundry	829
19.22. Kotlin Support	831
19.23. Configuration properties	831
20. Spring Cloud Circuit Breaker	832
20.1. Configuring Resilience4J Circuit Breakers	832
20.2. Configuring Spring Retry Circuit Breakers	834
20.3. Building	836
20.4. Contributing	838
20.5. Preface	844
20.6. What's New in 3.0?	849
20.7. Introducing Spring Cloud Stream	850
20.8. Main Concepts	851
20.9. Programming Model	856
20.10. Binders	889
20.11. Configuration Options	901
20.12. Content Type Negotiation	908
20.13. Testing	916
20.14. Health Indicator	923
20.15. Samples	924
20.16. Binder Implementations	925
21. Spring Cloud Stream RabbitMQ Binder Reference Guide	926
22. Reference Guide	927
22.1. Usage	927
22.2. RabbitMQ Binder Overview	927
22.3. Configuration Options	928
22.4. Using Existing Queues/Exchanges	943
22.5. Retry With the RabbitMQ Binder	944
22.6. Error Channels	946
22.7. Dead-Letter Queue Processing	947
22.8. Partitioning with the RabbitMQ Binder	952
23. Appendices	956

Appendix B: Building	956
Appendix C: Contributing	957
24. Spring Cloud Stream Kafka Binder Reference Guide	959
25. Reference Guide	960
25.1. Apache Kafka Binder	960
25.2. Kafka Streams Binder	982
26. Appendices	1022
Appendix D: Building	1022
Appendix: Compendium of Configuration Properties	1025

Spring Cloud provides tools for developers to quickly build some of the common patterns in distributed systems (e.g. configuration management, service discovery, circuit breakers, intelligent routing, micro-proxy, control bus). Coordination of distributed systems leads to boiler plate patterns, and using Spring Cloud developers can quickly stand up services and applications that implement those patterns. They will work well in any distributed environment, including the developer's own laptop, bare metal data centres, and managed platforms such as Cloud Foundry.

Release Train Version: **2020.0.0-M2**

Supported Boot Version: **2.3.0.RELEASE**

Chapter 1. Features

Spring Cloud focuses on providing good out of box experience for typical use cases and extensibility mechanism to cover others.

- Distributed/versioned configuration
- Service registration and discovery
- Routing
- Service-to-service calls
- Load balancing
- Circuit Breakers
- Distributed messaging

Chapter 2. Release Train Versions

Table 1. Release Train Project Versions

Project Name	Project Version
spring-cloud-build	3.0.0-M2
spring-cloud-commons	3.0.0-M2
spring-cloud-function	3.1.0.M1
spring-cloud-stream-core	3.1.0.M1
spring-cloud-stream-binder-rabbit	3.1.0.M1
spring-cloud-stream-binder-kafka	3.1.0.M1
spring-cloud-bus	3.0.0-M2
spring-cloud-task	2.3.0-M1
spring-cloud-config	3.0.0-M2
spring-cloud-netflix	3.0.0-M2
spring-cloud-cloudfoundry	3.0.0-M2
spring-cloud-kubernetes	2.0.0-M2
spring-cloud-openfeign	3.0.0-M2
spring-cloud-consul	3.0.0-M2
spring-cloud-gateway	3.0.0-M2
spring-cloud-security	3.0.0-M2
spring-cloud-sleuth	3.0.0-M2
spring-cloud-zookeeper	3.0.0-M2
spring-cloud-contract	3.0.0-M2
spring-cloud-vault	3.0.0-M2
spring-cloud-circuitbreaker	2.0.0-M2
spring-cloud-cli	3.0.0-M2

Chapter 3. Cloud Native Applications

Cloud Native is a style of application development that encourages easy adoption of best practices in the areas of continuous delivery and value-driven development. A related discipline is that of building **12-factor Applications**, in which development practices are aligned with delivery and operations goals—for instance, by using declarative programming and management and monitoring. Spring Cloud facilitates these styles of development in a number of specific ways. The starting point is a set of features to which all components in a distributed system need easy access.

Many of those features are covered by **Spring Boot**, on which Spring Cloud builds. Some more features are delivered by Spring Cloud as two libraries: Spring Cloud Context and Spring Cloud Commons. Spring Cloud Context provides utilities and special services for the **ApplicationContext** of a Spring Cloud application (bootstrap context, encryption, refresh scope, and environment endpoints). Spring Cloud Commons is a set of abstractions and common classes used in different Spring Cloud implementations (such as Spring Cloud Netflix and Spring Cloud Consul).

If you get an exception due to "Illegal key size" and you use Sun's JDK, you need to install the Java Cryptography Extension (JCE) Unlimited Strength Jurisdiction Policy Files. See the following links for more information:

- [Java 6 JCE](#)
- [Java 7 JCE](#)
- [Java 8 JCE](#)

Extract the files into the `JDK/jre/lib/security` folder for whichever version of JRE/JDK x64/x86 you use.



Spring Cloud is released under the non-restrictive Apache 2.0 license. If you would like to contribute to this section of the documentation or if you find an error, you can find the source code and issue trackers for the project at [{docslink}\[github\]](#).

3.1. Spring Cloud Context: Application Context Services

Spring Boot has an opinionated view of how to build an application with Spring. For instance, it has conventional locations for common configuration files and has endpoints for common management and monitoring tasks. Spring Cloud builds on top of that and adds a few features that many components in a system would use or occasionally need.

3.1.1. The Bootstrap Application Context

A Spring Cloud application operates by creating a “bootstrap” context, which is a parent context for the main application. This context is responsible for loading configuration properties from the external sources and for decrypting properties in the local external configuration files. The two contexts share an **Environment**, which is the source of external properties for any Spring application. By default, bootstrap properties (not `bootstrap.properties` but properties that are loaded during the

bootstrap phase) are added with high precedence, so they cannot be overridden by local configuration.

The bootstrap context uses a different convention for locating external configuration than the main application context. Instead of `application.yml` (or `.properties`), you can use `bootstrap.yml`, keeping the external configuration for bootstrap and main context nicely separate. The following listing shows an example:

Example 1. bootstrap.yml

```
spring:
  application:
    name: foo
  cloud:
    config:
      uri: ${SPRING_CONFIG_URI:http://localhost:8888}
```

If your application needs any application-specific configuration from the server, it is a good idea to set the `spring.application.name` (in `bootstrap.yml` or `application.yml`). For the property `spring.application.name` to be used as the application's context ID, you must set it in `bootstrap.[properties | yml]`.

If you want to retrieve specific profile configuration, you should also set `spring.profiles.active` in `bootstrap.[properties | yml]`.

You can disable the bootstrap process completely by setting `spring.cloud.bootstrap.enabled=false` (for example, in system properties).

3.1.2. Application Context Hierarchies

If you build an application context from `SpringApplication` or `SpringApplicationBuilder`, the Bootstrap context is added as a parent to that context. It is a feature of Spring that child contexts inherit property sources and profiles from their parent, so the “main” application context contains additional property sources, compared to building the same context without Spring Cloud Config. The additional property sources are:

- “bootstrap”: If any `PropertySourceLocators` are found in the bootstrap context and if they have non-empty properties, an optional `CompositePropertySource` appears with high priority. An example would be properties from the Spring Cloud Config Server. See “[Customizing the Bootstrap Property Sources](#)” for how to customize the contents of this property source.
- “applicationConfig: [classpath:bootstrap.yml]” (and related files if Spring profiles are active): If you have a `bootstrap.yml` (or `.properties`), those properties are used to configure the bootstrap context. Then they get added to the child context when its parent is set. They have lower precedence than the `application.yml` (or `.properties`) and any other property sources that are added to the child as a normal part of the process of creating a Spring Boot application. See “[Changing the Location of Bootstrap Properties](#)” for how to customize the contents of these property sources.

Because of the ordering rules of property sources, the “bootstrap” entries take precedence. However, note that these do not contain any data from `bootstrap.yml`, which has very low precedence but can be used to set defaults.

You can extend the context hierarchy by setting the parent context of any `ApplicationContext` you create—for example, by using its own interface or with the `SpringApplicationBuilder` convenience methods (`parent()`, `child()` and `sibling()`). The bootstrap context is the parent of the most senior ancestor that you create yourself. Every context in the hierarchy has its own “bootstrap” (possibly empty) property source to avoid promoting values inadvertently from parents down to their descendants. If there is a config server, every context in the hierarchy can also (in principle) have a different `spring.application.name` and, hence, a different remote property source. Normal Spring application context behavior rules apply to property resolution: properties from a child context override those in the parent, by name and also by property source name. (If the child has a property source with the same name as the parent, the value from the parent is not included in the child).

Note that the `SpringApplicationBuilder` lets you share an `Environment` amongst the whole hierarchy, but that is not the default. Thus, sibling contexts (in particular) do not need to have the same profiles or property sources, even though they may share common values with their parent.

3.1.3. Changing the Location of Bootstrap Properties

The `bootstrap.yml` (or `.properties`) location can be specified by setting `spring.cloud.bootstrap.name` (default: `bootstrap`), `spring.cloud.bootstrap.location` (default: `empty`) or `spring.cloud.bootstrap.additional-location` (default: `empty`)—for example, in System properties.

Those properties behave like the `spring.config.*` variants with the same name. With `spring.cloud.bootstrap.location` the default locations are replaced and only the specified ones are used. To add locations to the list of default ones, `spring.cloud.bootstrap.additional-location` could be used. In fact, they are used to set up the bootstrap `ApplicationContext` by setting those properties in its `Environment`. If there is an active profile (from `spring.profiles.active` or through the `Environment` API in the context you are building), properties in that profile get loaded as well, the same as in a regular Spring Boot app—for example, from `bootstrap-development.properties` for a `development` profile.

3.1.4. Overriding the Values of Remote Properties

The property sources that are added to your application by the bootstrap context are often “remote” (from example, from Spring Cloud Config Server). By default, they cannot be overridden locally. If you want to let your applications override the remote properties with their own system properties or config files, the remote property source has to grant it permission by setting `spring.cloud.config.allowOverride=true` (it does not work to set this locally). Once that flag is set, two finer-grained settings control the location of the remote properties in relation to system properties and the application’s local configuration:

- `spring.cloud.config.overrideNone=true`: Override from any local property source.
- `spring.cloud.config.overrideSystemProperties=false`: Only system properties, command line arguments, and environment variables (but not the local config files) should override the

remote settings.

3.1.5. Customizing the Bootstrap Configuration

The bootstrap context can be set to do anything you like by adding entries to `/META-INF/spring.factories` under a key named `org.springframework.cloud.bootstrap.BootstrapConfiguration`. This holds a comma-separated list of Spring `@Configuration` classes that are used to create the context. Any beans that you want to be available to the main application context for autowiring can be created here. There is a special contract for `@Beans` of type `ApplicationContextInitializer`. If you want to control the startup sequence, you can mark classes with the `@Order` annotation (the default order is `last`).



When adding custom `BootstrapConfiguration`, be careful that the classes you add are not `@ComponentScanned` by mistake into your “main” application context, where they might not be needed. Use a separate package name for boot configuration classes and make sure that name is not already covered by your `@ComponentScan` or `@SpringBootApplication` annotated configuration classes.

The bootstrap process ends by injecting initializers into the main `SpringApplication` instance (which is the normal Spring Boot startup sequence, whether it runs as a standalone application or is deployed in an application server). First, a bootstrap context is created from the classes found in `spring.factories`. Then, all `@Beans` of type `ApplicationContextInitializer` are added to the main `SpringApplication` before it is started.

3.1.6. Customizing the Bootstrap Property Sources

The default property source for external configuration added by the bootstrap process is the Spring Cloud Config Server, but you can add additional sources by adding beans of type `PropertySourceLocator` to the bootstrap context (through `spring.factories`). For instance, you can insert additional properties from a different server or from a database.

As an example, consider the following custom locator:

```
@Configuration
public class CustomPropertySourceLocator implements PropertySourceLocator {

    @Override
    public PropertySource<?> locate(Environment environment) {
        return new MapPropertySource("customProperty",
            Collections.<String,
                Object>singletonMap("property.from.sample.custom.source", "worked as intended"));
    }
}
```

The `Environment` that is passed in is the one for the `ApplicationContext` about to be created—in

other words, the one for which we supply additional property sources. It already has its normal Spring Boot-provided property sources, so you can use those to locate a property source specific to this `Environment` (for example, by keying it on `spring.application.name`, as is done in the default Spring Cloud Config Server property source locator).

If you create a jar with this class in it and then add a `META-INF/spring.factories` containing the following setting, the `customProperty PropertySource` appears in any application that includes that jar on its classpath:

```
org.springframework.cloud.bootstrap.BootstrapConfiguration=sample.custom.CustomPropertySourceLocator
```

3.1.7. Logging Configuration

If you use Spring Boot to configure log settings, you should place this configuration in `bootstrap.[yml | properties]` if you would like it to apply to all events.



For Spring Cloud to initialize logging configuration properly, you cannot use a custom prefix. For example, using `custom.loggin.logpath` is not recognized by Spring Cloud when initializing the logging system.

3.1.8. Environment Changes

The application listens for an `EnvironmentChangeEvent` and reacts to the change in a couple of standard ways (additional `ApplicationListeners` can be added as `@Beans` in the normal way). When an `EnvironmentChangeEvent` is observed, it has a list of key values that have changed, and the application uses those to:

- Re-bind any `@ConfigurationProperties` beans in the context.
- Set the logger levels for any properties in `logging.level.*`.

Note that the Spring Cloud Config Client does not, by default, poll for changes in the `Environment`. Generally, we would not recommend that approach for detecting changes (although you could set it up with a `@Scheduled` annotation). If you have a scaled-out client application, it is better to broadcast the `EnvironmentChangeEvent` to all the instances instead of having them polling for changes (for example, by using the [Spring Cloud Bus](#)).

The `EnvironmentChangeEvent` covers a large class of refresh use cases, as long as you can actually make a change to the `Environment` and publish the event. Note that those APIs are public and part of core Spring). You can verify that the changes are bound to `@ConfigurationProperties` beans by visiting the `/configprops` endpoint (a standard Spring Boot Actuator feature). For instance, a `DataSource` can have its `maxPoolSize` changed at runtime (the default `DataSource` created by Spring Boot is a `@ConfigurationProperties` bean) and grow capacity dynamically. Re-binding `@ConfigurationProperties` does not cover another large class of use cases, where you need more control over the refresh and where you need a change to be atomic over the whole `ApplicationContext`. To address those concerns, we have `@RefreshScope`.

3.1.9. Refresh Scope

When there is a configuration change, a Spring `@Bean` that is marked as `@RefreshScope` gets special treatment. This feature addresses the problem of stateful beans that get their configuration injected only when they are initialized. For instance, if a `DataSource` has open connections when the database URL is changed through the `Environment`, you probably want the holders of those connections to be able to complete what they are doing. Then, the next time something borrows a connection from the pool, it gets one with the new URL.

Sometimes, it might even be mandatory to apply the `@RefreshScope` annotation on some beans that can be only initialized once. If a bean is “immutable”, you have to either annotate the bean with `@RefreshScope` or specify the classname under the property key: `spring.cloud.refresh.extra-refreshable`.



If you have a `DataSource` bean that is a `HikariDataSource`, it can not be refreshed. It is the default value for `spring.cloud.refresh.never-refreshable`. Choose a different `DataSource` implementation if you need it to be refreshed.

Refresh scope beans are lazy proxies that initialize when they are used (that is, when a method is called), and the scope acts as a cache of initialized values. To force a bean to re-initialize on the next method call, you must invalidate its cache entry.

The `RefreshScope` is a bean in the context and has a public `refreshAll()` method to refresh all beans in the scope by clearing the target cache. The `/refresh` endpoint exposes this functionality (over HTTP or JMX). To refresh an individual bean by name, there is also a `refresh(String)` method.

To expose the `/refresh` endpoint, you need to add following configuration to your application:

```
management:
  endpoints:
    web:
      exposure:
        include: refresh
```



`@RefreshScope` works (technically) on a `@Configuration` class, but it might lead to surprising behavior. For example, it does not mean that all the `@Beans` defined in that class are themselves in `@RefreshScope`. Specifically, anything that depends on those beans cannot rely on them being updated when a refresh is initiated, unless it is itself in `@RefreshScope`. In that case, it is rebuilt on a refresh and its dependencies are re-injected. At that point, they are re-initialized from the refreshed `@Configuration`.

3.1.10. Encryption and Decryption

Spring Cloud has an `Environment` pre-processor for decrypting property values locally. It follows the same rules as the Spring Cloud Config Server and has the same external configuration through

`encrypt.*`. Thus, you can use encrypted values in the form of `{cipher}*`, and, as long as there is a valid key, they are decrypted before the main application context gets the `Environment` settings. To use the encryption features in an application, you need to include Spring Security RSA in your classpath (Maven co-ordinates: `org.springframework.security:spring-security-rsa`), and you also need the full strength JCE extensions in your JVM.

If you get an exception due to "Illegal key size" and you use Sun's JDK, you need to install the Java Cryptography Extension (JCE) Unlimited Strength Jurisdiction Policy Files. See the following links for more information:

- [Java 6 JCE](#)
- [Java 7 JCE](#)
- [Java 8 JCE](#)

Extract the files into the `JDK/jre/lib/security` folder for whichever version of JRE/JDK x64/x86 you use.

3.1.11. Endpoints

For a Spring Boot Actuator application, some additional management endpoints are available. You can use:

- `POST` to `/actuator/env` to update the `Environment` and rebind `@ConfigurationProperties` and log levels.
- `/actuator/refresh` to re-load the boot strap context and refresh the `@RefreshScope` beans.
- `/actuator/restart` to close the `ApplicationContext` and restart it (disabled by default).
- `/actuator/pause` and `/actuator/resume` for calling the `Lifecycle` methods (`stop()` and `start()` on the `ApplicationContext`).



If you disable the `/actuator/restart` endpoint then the `/actuator/pause` and `/actuator/resume` endpoints will also be disabled since they are just a special case of `/actuator/restart`.

3.2. Spring Cloud Commons: Common Abstractions

Patterns such as service discovery, load balancing, and circuit breakers lend themselves to a common abstraction layer that can be consumed by all Spring Cloud clients, independent of the implementation (for example, discovery with Eureka or Consul).

3.2.1. The `@EnableDiscoveryClient` Annotation

Spring Cloud Commons provides the `@EnableDiscoveryClient` annotation. This looks for implementations of the `DiscoveryClient` and `ReactiveDiscoveryClient` interfaces with `META-INF/spring.factories`. Implementations of the discovery client add a configuration class to `spring.factories` under the `org.springframework.cloud.client.discovery.EnableDiscoveryClient` key. Examples of `DiscoveryClient` implementations include [Spring Cloud Netflix Eureka](#), [Spring Cloud](#)

[Consul Discovery](#), and [Spring Cloud Zookeeper Discovery](#).

Spring Cloud will provide both the blocking and reactive service discovery clients by default. You can disable the blocking and/or reactive clients easily by setting `spring.cloud.discovery.blocking.enabled=false` or `spring.cloud.discovery.reactive.enabled=false`. To completely disable service discovery you just need to set `spring.cloud.discovery.enabled=false`.

By default, implementations of `DiscoveryClient` auto-register the local Spring Boot server with the remote discovery server. This behavior can be disabled by setting `autoRegister=false` in `@EnableDiscoveryClient`.



`@EnableDiscoveryClient` is no longer required. You can put a `DiscoveryClient` implementation on the classpath to cause the Spring Boot application to register with the service discovery server.

Health Indicator

Commons creates a Spring Boot `HealthIndicator` that `DiscoveryClient` implementations can participate in by implementing `DiscoveryHealthIndicator`. To disable the composite `HealthIndicator`, set `spring.cloud.discovery.client.composite-indicator.enabled=false`. A generic `HealthIndicator` based on `DiscoveryClient` is auto-configured (`DiscoveryClientHealthIndicator`). To disable it, set `spring.cloud.discovery.client.health-indicator.enabled=false`. To disable the description field of the `DiscoveryClientHealthIndicator`, set `spring.cloud.discovery.client.health-indicator.include-description=false`. Otherwise, it can bubble up as the `description` of the rolled up `HealthIndicator`.

Ordering `DiscoveryClient` instances

`DiscoveryClient` interface extends `Ordered`. This is useful when using multiple discovery clients, as it allows you to define the order of the returned discovery clients, similar to how you can order the beans loaded by a Spring application. By default, the order of any `DiscoveryClient` is set to `0`. If you want to set a different order for your custom `DiscoveryClient` implementations, you just need to override the `getOrder()` method so that it returns the value that is suitable for your setup. Apart from this, you can use properties to set the order of the `DiscoveryClient` implementations provided by Spring Cloud, among others `ConsulDiscoveryClient`, `EurekaDiscoveryClient` and `ZookeeperDiscoveryClient`. In order to do it, you just need to set the `spring.cloud.{clientId}.discovery.order` (or `eureka.client.order` for Eureka) property to the desired value.

SimpleDiscoveryClient

If there is no Service-Registry-backed `DiscoveryClient` in the classpath, `SimpleDiscoveryClient` instance, that uses properties to get information on service and instances, will be used.

The information about the available instances should be passed to via properties in the following format: `spring.cloud.discovery.client.simple.instances.service1[0].uri=http://s11:8080`, where `spring.cloud.discovery.client.simple.instances` is the common prefix, then `service1` stands for the ID of the service in question, while `[0]` indicates the index number of the instance (as visible in the example, indexes start with `0`), and then the value of `uri` is the actual URI under which the instance is available.

3.2.2. ServiceRegistry

Commons now provides a `ServiceRegistry` interface that provides methods such as `register(Registration)` and `deregister(Registration)`, which let you provide custom registered services. `Registration` is a marker interface.

The following example shows the `ServiceRegistry` in use:

```
@Configuration
@EnableDiscoveryClient(autoRegister=false)
public class MyConfiguration {
    private ServiceRegistry registry;

    public MyConfiguration(ServiceRegistry registry) {
        this.registry = registry;
    }

    // called through some external process, such as an event or a custom actuator
    endpoint
    public void register() {
        Registration registration = constructRegistration();
        this.registry.register(registration);
    }
}
```

Each `ServiceRegistry` implementation has its own `Registry` implementation.

- `ZookeeperRegistration` used with `ZookeeperServiceRegistry`
- `EurekaRegistration` used with `EurekaServiceRegistry`
- `ConsulRegistration` used with `ConsulServiceRegistry`

If you are using the `ServiceRegistry` interface, you are going to need to pass the correct `Registry` implementation for the `ServiceRegistry` implementation you are using.

ServiceRegistry Auto-Registration

By default, the `ServiceRegistry` implementation auto-registers the running service. To disable that behavior, you can set: * `@EnableDiscoveryClient(autoRegister=false)` to permanently disable auto-registration. * `spring.cloud.service-registry.auto-registration.enabled=false` to disable the behavior through configuration.

ServiceRegistry Auto-Registration Events

There are two events that will be fired when a service auto-registers. The first event, called `InstancePreRegisteredEvent`, is fired before the service is registered. The second event, called `InstanceRegisteredEvent`, is fired after the service is registered. You can register an `ApplicationListener`(s) to listen to and react to these events.



These events will not be fired if the `spring.cloud.service-registry.auto-registration.enabled` property is set to `false`.

Service Registry Actuator Endpoint

Spring Cloud Commons provides a `/service-registry` actuator endpoint. This endpoint relies on a `Registration` bean in the Spring Application Context. Calling `/service-registry` with GET returns the status of the `Registration`. Using POST to the same endpoint with a JSON body changes the status of the current `Registration` to the new value. The JSON body has to include the `status` field with the preferred value. Please see the documentation of the `ServiceRegistry` implementation you use for the allowed values when updating the status and the values returned for the status. For instance, Eureka's supported statuses are `UP`, `DOWN`, `OUT_OF_SERVICE`, and `UNKNOWN`.

3.2.3. Spring RestTemplate as a Load Balancer Client

You can configure a `RestTemplate` to use a Load-balancer client. To create a load-balanced `RestTemplate`, create a `RestTemplate @Bean` and use the `@LoadBalanced` qualifier, as the following example shows:

```
@Configuration
public class MyConfiguration {

    @LoadBalanced
    @Bean
    RestTemplate restTemplate() {
        return new RestTemplate();
    }
}

public class MyClass {
    @Autowired
    private RestTemplate restTemplate;

    public String doOtherStuff() {
        String results = restTemplate.getForObject("http://stores/stores",
String.class);
        return results;
    }
}
```



A `RestTemplate` bean is no longer created through auto-configuration. Individual applications must create it.

The URI needs to use a virtual host name (that is, a service name, not a host name). The `BlockingLoadBalancerClient` is used to create a full physical address.



To use a load-balanced `RestTemplate`, you need to have a load-balancer implementation in your classpath. Add `Spring Cloud LoadBalancer starter` to your project in order to use it.

3.2.4. Spring WebClient as a Load Balancer Client

You can configure `WebClient` to automatically use a load-balancer client. To create a load-balanced `WebClient`, create a `WebClient.Builder` `@Bean` and use the `@LoadBalanced` qualifier, as follows:

```
@Configuration
public class MyConfiguration {

    @Bean
    @LoadBalanced
    public WebClient.Builder loadBalancedWebClientBuilder() {
        return WebClient.builder();
    }
}

public class MyClass {
    @Autowired
    private WebClient.Builder webClientBuilder;

    public Mono<String> doOtherStuff() {
        return webClientBuilder.build().get().uri("http://stores/stores")
            .retrieve().bodyToMono(String.class);
    }
}
```

The URI needs to use a virtual host name (that is, a service name, not a host name). The Spring Cloud LoadBalancer is used to create a full physical address.



If you want to use a `@LoadBalanced WebClient.Builder`, you need to have a load balancer implementation in the classpath. We recommend that you add the `Spring Cloud LoadBalancer starter` to your project. Then, `ReactiveLoadBalancer` is used underneath.

Retrying Failed Requests

A load-balanced `RestTemplate` can be configured to retry failed requests. By default, this logic is disabled. You can enable it by adding `Spring Retry` to your application's classpath. If you would like to disable the retry logic with Spring Retry on the classpath, you can set `spring.cloud.loadbalancer.retry.enabled=false`.

If you would like to implement a `BackOffPolicy` in your retries, you need to create a bean of type `LoadBalancedRetryFactory` and override the `createBackOffPolicy` method:

```
@Configuration
public class MyConfiguration {
    @Bean
    LoadBalancedRetryFactory retryFactory() {
        return new LoadBalancedRetryFactory() {
            @Override
            public BackOffPolicy createBackOffPolicy(String service) {
                return new ExponentialBackOffPolicy();
            }
        };
    }
}
```

If you want to add one or more `RetryListener` implementations to your retry functionality, you need to create a bean of type `LoadBalancedRetryListenerFactory` and return the `RetryListener` array you would like to use for a given service, as the following example shows:

```

@Configuration
public class MyConfiguration {
    @Bean
    LoadBalancedRetryListenerFactory retryListenerFactory() {
        return new LoadBalancedRetryListenerFactory() {
            @Override
            public RetryListener[] createRetryListeners(String service) {
                return new RetryListener[]{new RetryListener() {
                    @Override
                    public <T, E extends Throwable> boolean open(RetryContext
context, RetryCallback<T, E> callback) {
                        //TODO Do you business...
                        return true;
                    }

                    @Override
                    public <T, E extends Throwable> void close(RetryContext
context, RetryCallback<T, E> callback, Throwable throwable) {
                        //TODO Do you business...
                    }

                    @Override
                    public <T, E extends Throwable> void onError(RetryContext
context, RetryCallback<T, E> callback, Throwable throwable) {
                        //TODO Do you business...
                    }
                }};
            }
        };
    }
}

```

3.2.5. Multiple `RestTemplate` Objects

If you want a `RestTemplate` that is not load-balanced, create a `RestTemplate` bean and inject it. To access the load-balanced `RestTemplate`, use the `@LoadBalanced` qualifier when you create your `@Bean`, as the following example shows:

```

@Configuration
public class MyConfiguration {

    @LoadBalanced
    @Bean
    RestTemplate loadBalanced() {
        return new RestTemplate();
    }

    @Primary
    @Bean
    RestTemplate restTemplate() {
        return new RestTemplate();
    }
}

public class MyClass {
    @Autowired
    private RestTemplate restTemplate;

    @Autowired
    @LoadBalanced
    private RestTemplate loadBalanced;

    public String doOtherStuff() {
        return loadBalanced.getForObject("http://stores/stores", String.class);
    }

    public String doStuff() {
        return restTemplate.getForObject("http://example.com", String.class);
    }
}

```



Notice the use of the `@Primary` annotation on the plain `RestTemplate` declaration in the preceding example to disambiguate the unqualified `@Autowired` injection.



If you see errors such as `java.lang.IllegalArgumentException: Can not set org.springframework.web.client.RestTemplate field com.my.app.Foo.restTemplate to com.sun.proxy.$Proxy89`, try injecting `RestOperations` or setting `spring.aop.proxyTargetClass=true`.

3.2.6. Multiple WebClient Objects

If you want a `WebClient` that is not load-balanced, create a `WebClient` bean and inject it. To access the load-balanced `WebClient`, use the `@LoadBalanced` qualifier when you create your `@Bean`, as the following example shows:

```

@Configuration
public class MyConfiguration {

    @LoadBalanced
    @Bean
    WebClient.Builder loadBalanced() {
        return WebClient.builder();
    }

    @Primary
    @Bean
    WebClient.Builder webClient() {
        return WebClient.builder();
    }
}

public class MyClass {
    @Autowired
    private WebClient.Builder webClientBuilder;

    @Autowired
    @LoadBalanced
    private WebClient.Builder loadBalanced;

    public Mono<String> doOtherStuff() {
        return loadBalanced.build().get().uri("http://stores/stores")
            .retrieve().bodyToMono(String.class);
    }

    public Mono<String> doStuff() {
        return webClientBuilder.build().get().uri("http://example.com")
            .retrieve().bodyToMono(String.class);
    }
}

```

3.2.7. Spring WebFlux `WebClient` as a Load Balancer Client

The Spring WebFlux can work with both reactive and non-reactive `WebClient` configurations, as the topics describe:

- [Spring WebFlux `WebClient` with `ReactorLoadBalancerExchangeFilterFunction`](#)
- [\[load-balancer-exchange-filter-functionload-balancer-exchange-filter-function\]](#)

Spring WebFlux `WebClient` with `ReactorLoadBalancerExchangeFilterFunction`

You can configure `WebClient` to use the `ReactiveLoadBalancer`. If you add [Spring Cloud LoadBalancer starter](#) to your project and if `spring-webflux` is on the classpath,

`ReactorLoadBalancerExchangeFilterFunction` is auto-configured. The following example shows how to configure a `WebClient` to use reactive load-balancer:

```
public class MyClass {
    @Autowired
    private ReactorLoadBalancerExchangeFilterFunction lbFunction;

    public Mono<String> doOtherStuff() {
        return WebClient.builder().baseUrl("http://stores")
            .filter(lbFunction)
            .build()
            .get()
            .uri("/stores")
            .retrieve()
            .bodyToMono(String.class);
    }
}
```

The URI needs to use a virtual host name (that is, a service name, not a host name). The `ReactorLoadBalancer` is used to create a full physical address.

Spring WebFlux `WebClient` with a Non-reactive Load Balancer Client

If `spring-webflux` is on the classpath, `LoadBalancerExchangeFilterFunction` is auto-configured. Note, however, that this uses a non-reactive client under the hood. The following example shows how to configure a `WebClient` to use load-balancer:

```
public class MyClass {
    @Autowired
    private LoadBalancerExchangeFilterFunction lbFunction;

    public Mono<String> doOtherStuff() {
        return WebClient.builder().baseUrl("http://stores")
            .filter(lbFunction)
            .build()
            .get()
            .uri("/stores")
            .retrieve()
            .bodyToMono(String.class);
    }
}
```

The URI needs to use a virtual host name (that is, a service name, not a host name). The `LoadBalancerClient` is used to create a full physical address.

WARN: This approach is now deprecated. We suggest that you use [WebFlux with reactive Load-Balancer](#) instead.

3.2.8. Ignore Network Interfaces

Sometimes, it is useful to ignore certain named network interfaces so that they can be excluded from Service Discovery registration (for example, when running in a Docker container). A list of regular expressions can be set to cause the desired network interfaces to be ignored. The following configuration ignores the `docker0` interface and all interfaces that start with `veth`:

Example 2. application.yml

```
spring:
  cloud:
    inetutils:
      ignoredInterfaces:
        - docker0
        - veth.*
```

You can also force the use of only specified network addresses by using a list of regular expressions, as the following example shows:

Example 3. bootstrap.yml

```
spring:
  cloud:
    inetutils:
      preferredNetworks:
        - 192.168
        - 10.0
```

You can also force the use of only site-local addresses, as the following example shows:

Example 4. application.yml

```
spring:
  cloud:
    inetutils:
      useOnlySiteLocalInterfaces: true
```

See [Inet4Address.html.isSiteLocalAddress\(\)](#) for more details about what constitutes a site-local address.

3.2.9. HTTP Client Factories

Spring Cloud Commons provides beans for creating both Apache HTTP clients (`ApacheHttpClientFactory`) and OK HTTP clients (`OkHttpClientFactory`). The `OkHttpClientFactory` bean is created only if the OK HTTP jar is on the classpath. In addition, Spring Cloud Commons provides beans for creating the connection managers used by both clients: `ApacheHttpClientConnectionFactory` for the Apache HTTP client and `OkHttpClientConnectionPoolFactory` for the OK HTTP client. If you would like to customize how the HTTP clients are created in downstream projects, you can provide your own implementation of these beans. In addition, if you provide a bean of type `HttpClientBuilder` or `OkHttpClient.Builder`, the default factories use these builders as the basis for the builders returned to downstream projects. You can also disable the creation of these beans by setting `spring.cloud.httpclientfactories.apache.enabled` or `spring.cloud.httpclientfactories.ok.enabled` to `false`.

3.2.10. Enabled Features

Spring Cloud Commons provides a `/features` actuator endpoint. This endpoint returns features available on the classpath and whether they are enabled. The information returned includes the feature type, name, version, and vendor.

Feature types

There are two types of 'features': abstract and named.

Abstract features are features where an interface or abstract class is defined and that an implementation the creates, such as `DiscoveryClient`, `LoadBalancerClient`, or `LockService`. The abstract class or interface is used to find a bean of that type in the context. The version displayed is `bean.getClass().getPackage().getImplementationVersion()`.

Named features are features that do not have a particular class they implement. These features include “Circuit Breaker”, “API Gateway”, “Spring Cloud Bus”, and others. These features require a name and a bean type.

Declaring features

Any module can declare any number of `HasFeature` beans, as the following examples show:


```

@Bean
public HasFeatures commonsFeatures() {
    return HasFeatures.abstractFeatures(DiscoveryClient.class,
    LoadBalancerClient.class);
}

@Bean
public HasFeatures consulFeatures() {
    return HasFeatures.namedFeatures(
        new NamedFeature("Spring Cloud Bus", ConsulBusAutoConfiguration.class),
        new NamedFeature("Circuit Breaker", HystrixCommandAspect.class));
}

@Bean
HasFeatures localFeatures() {
    return HasFeatures.builder()
        .abstractFeature(Something.class)
        .namedFeature(new NamedFeature("Some Other Feature", Someother.class))
        .abstractFeature(Somethingelse.class)
        .build();
}

```

Each of these beans should go in an appropriately guarded `@Configuration`.

3.2.11. Spring Cloud Compatibility Verification

Due to the fact that some users have problem with setting up Spring Cloud application, we've decided to add a compatibility verification mechanism. It will break if your current setup is not compatible with Spring Cloud requirements, together with a report, showing what exactly went wrong.

At the moment we verify which version of Spring Boot is added to your classpath.

Example of a report

```
*****
APPLICATION FAILED TO START
*****
```

Description:

Your project setup is incompatible with our requirements due to following reasons:

- Spring Boot [2.1.0.RELEASE] is not compatible with this Spring Cloud release train

Action:

Consider applying the following actions:

- Change Spring Boot version to one of the following versions [1.2.x, 1.3.x] .

You can find the latest Spring Boot versions here

[<https://spring.io/projects/spring-boot#learn>].

If you want to learn more about the Spring Cloud Release train compatibility, you can visit this page [<https://spring.io/projects/spring-cloud#overview>] and check the [Release Trains] section.

In order to disable this feature, set `spring.cloud.compatibility-verifier.enabled` to `false`. If you want to override the compatible Spring Boot versions, just set the `spring.cloud.compatibility-verifier.compatible-boot-versions` property with a comma separated list of compatible Spring Boot versions.

3.3. Spring Cloud LoadBalancer

Spring Cloud provides its own client-side load-balancer abstraction and implementation. For the load-balancing mechanism, `ReactiveLoadBalancer` interface has been added and a Round-Robin-based implementation has been provided for it. In order to get instances to select from reactive `ServiceInstanceListSupplier` is used. Currently we support a service-discovery-based implementation of `ServiceInstanceListSupplier` that retrieves available instances from Service Discovery using a `Discovery Client` available in the classpath.

3.3.1. Spring Cloud LoadBalancer integrations

In order to make it easy to use Spring Cloud LoadBalancer, we provide `ReactorLoadBalancerExchangeFilterFunction` that can be used with `WebClient` and `BlockingLoadBalancerClient` that works with `RestTemplate`. You can see more information and examples of usage in the following sections:

- [Spring RestTemplate as a Load Balancer Client](#)
- [Spring WebClient as a Load Balancer Client](#)

- [Spring WebFlux WebClient with ReactorLoadBalancerExchangeFilterFunction](#)

3.3.2. Spring Cloud LoadBalancer Caching

Apart from the basic `ServiceInstanceListSupplier` implementation that retrieves instances via `DiscoveryClient` each time it has to choose an instance, we provide two caching implementations.

Caffeine-backed LoadBalancer Cache Implementation

If you have `com.github.ben-manes.caffeine:caffeine` in the classpath, Caffeine-based implementation will be used. See the [LoadBalancerCacheConfiguration](#) section for information on how to configure it.

If you are using Caffeine, you can also override the default Caffeine Cache setup for the LoadBalancer by passing your own [Caffeine Specification](#) in the `spring.cloud.loadbalancer.cache.caffeine.spec` property.

WARN: Passing your own Caffeine specification will override any other `LoadBalancerCache` settings, including [General LoadBalancer Cache Configuration](#) fields, such as `ttl` and `capacity`.

Default LoadBalancer Cache Implementation

If you do not have Caffeine in the classpath, the `DefaultLoadBalancerCache`, which comes automatically with `spring-cloud-starter-loadbalancer`, will be used. See the [LoadBalancerCacheConfiguration](#) section for information on how to configure it.



To use Caffeine instead of the default cache, add the `com.github.ben-manes.caffeine:caffeine` dependency to classpath.

LoadBalancer Cache Configuration

You can set your own `ttl` value (the time after write after which entries should be expired), expressed as `Duration`, by passing a `String` compliant with the [Spring Boot String to Duration converter syntax](#). as the value of the `spring.cloud.loadbalancer.cache.ttl` property. You can also set your own LoadBalancer cache initial capacity by setting the value of the `spring.cloud.loadbalancer.cache.capacity` property.

The default setup includes `ttl` set to 35 seconds and the default `initialCapacity` is 256.

You can also altogether disable loadBalancer caching by setting the value of `spring.cloud.loadbalancer.cache.enabled` to `false`.



Although the basic, non-cached, implementation is useful for prototyping and testing, it's much less efficient than the cached versions, so we recommend always using the cached version in production.

3.3.3. Zone-Based Load-Balancing

To enable zone-based load-balancing, we provide the `ZonePreferenceServiceInstanceListSupplier`.

We use `DiscoveryClient`-specific `zone` configuration (for example, `eureka.instance.metadata-map.zone`) to pick the zone that the client tries to filter available service instances for.



You can also override `DiscoveryClient`-specific zone setup by setting the value of `spring.cloud.loadbalancer.zone` property.



For the time being, only Eureka Discovery Client is instrumented to set the `LoadBalancer zone`. For other discovery client, set the `spring.cloud.loadbalancer.zone` property. More instrumentations coming shortly.



To determine the zone of a retrieved `ServiceInstance`, we check the value under the `"zone"` key in its metadata map.

The `ZonePreferenceServiceInstanceListSupplier` filters retrieved instances and only returns the ones within the same zone. If the zone is `null` or there are no instances within the same zone, it returns all the retrieved instances.

In order to use the zone-based load-balancing approach, you will have to instantiate a `ZonePreferenceServiceInstanceListSupplier` bean in a `custom configuration`.

We use delegates to work with `ServiceInstanceListSupplier` beans. We suggest passing a `DiscoveryClientServiceInstanceListSupplier` delegate in the constructor of `ZonePreferenceServiceInstanceListSupplier` and, in turn, wrapping the latter with a `CachingServiceInstanceListSupplier` to leverage `LoadBalancer caching mechanism`.

You could use this sample configuration to set it up:

```
public class CustomLoadBalancerConfiguration {

    @Bean
    public ServiceInstanceListSupplier discoveryClientServiceInstanceListSupplier(
        ConfigurableApplicationContext context) {
        return ServiceInstanceListSuppliers.builder()
            .withDiscoveryClient()
            .withZonePreference()
            .withCaching()
            .build(context);
    }
}
```

3.3.4. Instance Health-Check for LoadBalancer

It is possible to enable a scheduled `HealthCheck` for the `LoadBalancer`. The `HealthCheckServiceInstanceListSupplier` is provided for that. It regularly verifies if the instances provided by a delegate `ServiceInstanceListSupplier` are still alive and only returns the healthy instances, unless there are none - then it returns all the retrieved instances.



This mechanism is particularly helpful while using the `SimpleDiscoveryClient`. For the clients backed by an actual Service Registry, it's not necessary to use, as we already get healthy instances after querying the external ServiceDiscovery.

TIP

This supplier is also recommended for setups with a small number of instances per service in order to avoid retrying calls on a failing instance.

`HealthCheckServiceInstanceListSupplier` uses properties prefixed with `spring.cloud.loadbalancer.healthcheck`. You can set the `initialDelay` and `interval` for the scheduler. You can set the default path for the healthcheck URL by setting the value of the `spring.cloud.loadbalancer.healthcheck.path.default`. You can also set a specific value for any given service by setting the value of the `spring.cloud.loadbalancer.healthcheck.path.[SERVICE_ID]`, substituting the `[SERVICE_ID]` with the correct ID of your service. If the path is not set, `/actuator/health` is used by default.

TIP

If you rely on the default path (`/actuator/health`), make sure you add `spring-boot-starter-actuator` to your collaborator's dependencies, unless you are planning to add such an endpoint on your own.

In order to use the health-check scheduler approach, you will have to instantiate a `HealthCheckServiceInstanceListSupplier` bean in a [custom configuration](#).

We use delegates to work with `ServiceInstanceListSupplier` beans. We suggest passing a `DiscoveryClientServiceInstanceListSupplier` delegate in the constructor of `HealthCheckServiceInstanceListSupplier`.

You could use this sample configuration to set it up:

```
public class CustomLoadBalancerConfiguration {

    @Bean
    public ServiceInstanceListSupplier discoveryClientServiceInstanceListSupplier(
        ConfigurableApplicationContext context) {
        return ServiceInstanceListSupplier.builder()
            .withDiscoveryClient()
            .withHealthChecks()
            .build(context);
    }
}
```

NOTE

`HealthCheckServiceInstanceListSupplier` has its own caching mechanism based on Reactor Flux `replay()`, therefore, if it's being used, you may want to skip wrapping that supplier with `CachingServiceInstanceListSupplier`.

3.3.5. Spring Cloud LoadBalancer Starter

We also provide a starter that allows you to easily add Spring Cloud LoadBalancer in a Spring Boot app. In order to use it, just add `org.springframework.cloud:spring-cloud-starter-loadbalancer` to your Spring Cloud dependencies in your build file.



Spring Cloud LoadBalancer starter includes [Spring Boot Caching](#) and [Evictor](#).

3.3.6. Passing Your Own Spring Cloud LoadBalancer Configuration

You can also use the `@LoadBalancerClient` annotation to pass your own load-balancer client configuration, passing the name of the load-balancer client and the configuration class, as follows:

```
@Configuration
@LoadBalancerClient(value = "stores", configuration =
    CustomLoadBalancerConfiguration.class)
public class MyConfiguration {

    @Bean
    @LoadBalanced
    public WebClient.Builder loadBalancedWebClientBuilder() {
        return WebClient.builder();
    }
}
```

TIP

In order to make working on your own LoadBalancer configuration easier, we have added a `builder()` method to the `ServiceInstanceListSupplier` class.

TIP

You can also use our alternative predefined configurations in place of the default ones by setting the value of `spring.cloud.loadbalancer.configurations` property to `zone-preference` to use `ZonePreferenceServiceInstanceListSupplier` with caching or to `health-check` to use `HealthCheckServiceInstanceListSupplier` with caching.

You can use this feature to instantiate different implementations of `ServiceInstanceListSupplier` or `ReactorLoadBalancer`, either written by you, or provided by us as alternatives (for example `ZonePreferenceServiceInstanceListSupplier`) to override the default setup.

You can see an example of a custom configuration [here](#).



The annotation `value` arguments (`stores` in the example above) specifies the service id of the service that we should send the requests to with the given custom configuration.

You can also pass multiple configurations (for more than one load-balancer client) through the

`@LoadBalancerClients` annotation, as the following example shows:

```
@Configuration
@LoadBalancerClients({@LoadBalancerClient(value = "stores", configuration =
StoresLoadBalancerClientConfiguration.class), @LoadBalancerClient(value =
"customers", configuration = CustomersLoadBalancerClientConfiguration.class)})
public class MyConfiguration {

    @Bean
    @LoadBalanced
    public WebClient.Builder loadBalancedWebClientBuilder() {
        return WebClient.builder();
    }
}
```

3.4. Spring Cloud Circuit Breaker

3.4.1. Introduction

Spring Cloud Circuit breaker provides an abstraction across different circuit breaker implementations. It provides a consistent API to use in your applications, letting you, the developer, choose the circuit breaker implementation that best fits your needs for your application.

Supported Implementations

Spring Cloud supports the following circuit-breaker implementations:

- [Resilience4j](#)
- [Sentinel](#)
- [Spring Retry](#)

3.4.2. Core Concepts

To create a circuit breaker in your code, you can use the `CircuitBreakerFactory` API. When you include a Spring Cloud Circuit Breaker starter on your classpath, a bean that implements this API is automatically created for you. The following example shows a simple example of how to use this API:

```

@Service
public static class DemoControllerService {
    private RestTemplate rest;
    private CircuitBreakerFactory cbFactory;

    public DemoControllerService(RestTemplate rest, CircuitBreakerFactory
cbFactory) {
        this.rest = rest;
        this.cbFactory = cbFactory;
    }

    public String slow() {
        return cbFactory.create("slow").run(() -> rest.getForObject("/slow",
String.class), throwable -> "fallback");
    }
}

```

The `CircuitBreakerFactory.create` API creates an instance of a class called `CircuitBreaker`. The `run` method takes a `Supplier` and a `Function`. The `Supplier` is the code that you are going to wrap in a circuit breaker. The `Function` is the fallback that is executed if the circuit breaker is tripped. The function is passed the `Throwable` that caused the fallback to be triggered. You can optionally exclude the fallback if you do not want to provide one.

Circuit Breakers In Reactive Code

If Project Reactor is on the class path, you can also use `ReactiveCircuitBreakerFactory` for your reactive code. The following example shows how to do so:


```

@Service
public static class DemoControllerService {
    private ReactiveCircuitBreakerFactory cbFactory;
    private WebClient webClient;

    public DemoControllerService(WebClient webClient,
ReactiveCircuitBreakerFactory cbFactory) {
        this.webClient = webClient;
        this.cbFactory = cbFactory;
    }

    public Mono<String> slow() {
        return
webClient.get().uri("/slow").retrieve().bodyToMono(String.class).transform(
            it -> cbFactory.create("slow").run(it, throwable -> return
Mono.just("fallback")));
    }
}

```

The `ReactiveCircuitBreakerFactory.create` API creates an instance of a class called `ReactiveCircuitBreaker`. The `run` method takes a `Mono` or a `Flux` and wraps it in a circuit breaker. You can optionally profile a fallback `Function`, which will be called if the circuit breaker is tripped and is passed the `Throwable` that caused the failure.

3.4.3. Configuration

You can configure your circuit breakers by creating beans of type `Customizer`. The `Customizer` interface has a single method (called `customize`) that takes the `Object` to customize.

For detailed information on how to customize a given implementation see the following documentation:

- [Resilience4j](#)
- [Sentinal](#)
- [Spring Retry](#)

Some `CircuitBreaker` implementations such as `Resilience4JCircuitBreaker` call `customize` method every time `CircuitBreaker#run` is called. It can be inefficient. In that case, you can use `CircuitBreaker#once` method. It is useful where calling `customize` many times doesn't make sense, for example, in case of [consuming Resilience4j's events](#).

The following example shows the way for each `io.github.resilience4j.circuitbreaker.CircuitBreaker` to consume events.

```
Customizer.once(circuitBreaker -> {
    circuitBreaker.getEventPublisher()
        .onStateTransition(event -> log.info("{}: {}", event.getCircuitBreakerName(),
            event.getStateTransition()));
}, CircuitBreaker::getName)
```

3.5. CachedRandomPropertySource

Spring Cloud Context provides a `PropertySource` that caches random values based on a key. Outside of the caching functionality it works the same as Spring Boot's `RandomValuePropertySource`. This random value might be useful in the case where you want a random value that is consistent even after the Spring Application context restarts. The property value takes the form of `cachedrandom.[yourkey].[type]` where `yourkey` is the key in the cache. The `type` value can be any type supported by Spring Boot's `RandomValuePropertySource`.

```
myrandom=${cachedrandom.appname.value}
```

3.6. Configuration Properties

To see the list of all Spring Cloud Commons related configuration properties please check [the Appendix page](#).

Chapter 4. Spring Cloud Config

2020.0.0-M2

Spring Cloud Config provides server-side and client-side support for externalized configuration in a distributed system. With the Config Server, you have a central place to manage external properties for applications across all environments. The concepts on both client and server map identically to the Spring `Environment` and `PropertySource` abstractions, so they fit very well with Spring applications but can be used with any application running in any language. As an application moves through the deployment pipeline from dev to test and into production, you can manage the configuration between those environments and be certain that applications have everything they need to run when they migrate. The default implementation of the server storage backend uses git, so it easily supports labelled versions of configuration environments as well as being accessible to a wide range of tooling for managing the content. It is easy to add alternative implementations and plug them in with Spring configuration.

4.1. Quick Start

This quick start walks through using both the server and the client of Spring Cloud Config Server.

First, start the server, as follows:

```
$ cd spring-cloud-config-server
$ ../mvnw spring-boot:run
```

The server is a Spring Boot application, so you can run it from your IDE if you prefer to do so (the main class is `ConfigServerApplication`).

Next try out a client, as follows:

```
$ curl localhost:8888/foo/development
{"name":"foo","label":"master","propertySources":[
  {"name":"https://github.com/scratches/config-repo/foo-
development.properties","source":{"bar":"spam"}},
  {"name":"https://github.com/scratches/config-
repo/foo.properties","source":{"foo":"bar"}}
]}
```

The default strategy for locating property sources is to clone a git repository (at `spring.cloud.config.server.git.uri`) and use it to initialize a mini `SpringApplication`. The mini application's `Environment` is used to enumerate property sources and publish them at a JSON endpoint.

The HTTP service has resources in the following form:

```
/{application}/{profile}[/{label}]
/{application}-{profile}.yml
/{label}/{application}-{profile}.yml
/{application}-{profile}.properties
/{label}/{application}-{profile}.properties
```

where `application` is injected as the `spring.config.name` in the `SpringApplication` (what is normally `application` in a regular Spring Boot app), `profile` is an active profile (or comma-separated list of properties), and `label` is an optional git label (defaults to `master`.)

Spring Cloud Config Server pulls configuration for remote clients from various sources. The following example gets configuration from a git repository (which must be provided), as shown in the following example:

```
spring:
  cloud:
    config:
      server:
        git:
          uri: https://github.com/spring-cloud-samples/config-repo
```

Other sources are any JDBC compatible database, Subversion, Hashicorp Vault, Credhub and local filesystems.

4.1.1. Client Side Usage

To use these features in an application, you can build it as a Spring Boot application that depends on `spring-cloud-config-client` (for an example, see the test cases for the `config-client` or the sample application). The most convenient way to add the dependency is with a Spring Boot starter `org.springframework.cloud:spring-cloud-starter-config`. There is also a parent pom and BOM (`spring-cloud-starter-parent`) for Maven users and a Spring IO version management properties file for Gradle and Spring CLI users. The following example shows a typical Maven configuration:

```
<parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-parent</artifactId>
  <version>{spring-boot-docs-version}</version>
  <relativePath /> <!-- lookup parent from repository -->
</parent>

<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>org.springframework.cloud</groupId>
      <artifactId>spring-cloud-dependencies</artifactId>
      <version>{spring-cloud-version}</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
  </dependencies>
</dependencyManagement>

<dependencies>
  <dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-config</artifactId>
  </dependency>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
    <scope>test</scope>
  </dependency>
</dependencies>

<build>
  <plugins>
    <plugin>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-maven-plugin</artifactId>
    </plugin>
  </plugins>
</build>

<!-- repositories also needed for snapshots and milestones -->
```

Now you can create a standard Spring Boot application, such as the following HTTP server:

```

@SpringBootApplication
@RestController
public class Application {

    @RequestMapping("/")
    public String home() {
        return "Hello World!";
    }

    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }

}

```

When this HTTP server runs, it picks up the external configuration from the default local config server (if it is running) on port 8888. To modify the startup behavior, you can change the location of the config server by using `bootstrap.properties` (similar to `application.properties` but for the bootstrap phase of an application context), as shown in the following example:

```
spring.cloud.config.uri: http://myconfigserver.com
```

By default, if no application name is set, `application` will be used. To modify the name, the following property can be added to the `bootstrap.properties` file:

```
spring.application.name: myapp
```



When setting the property `spring.application.name` do not prefix your app name with the reserved word `application-` to prevent issues resolving the correct property source.

The bootstrap properties show up in the `/env` endpoint as a high-priority property source, as shown in the following example.

```

$ curl localhost:8080/env
{
  "profiles":[],
  "configService:https://github.com/spring-cloud-samples/config-repo/bar.properties":{"foo":"bar"},
  "servletContextInitParams":{},
  "systemProperties":{...},
  ...
}

```

A property source called `configService:<URL of remote repository>/<file name>` contains the `foo`

property with a value of `bar` and is the highest priority.



The URL in the property source name is the git repository, not the config server URL.

4.2. Spring Cloud Config Server

Spring Cloud Config Server provides an HTTP resource-based API for external configuration (name-value pairs or equivalent YAML content). The server is embeddable in a Spring Boot application, by using the `@EnableConfigServer` annotation. Consequently, the following application is a config server:

ConfigServer.java

```
@SpringBootApplication
@EnableConfigServer
public class ConfigServer {
    public static void main(String[] args) {
        SpringApplication.run(ConfigServer.class, args);
    }
}
```

Like all Spring Boot applications, it runs on port 8080 by default, but you can switch it to the more conventional port 8888 in various ways. The easiest, which also sets a default configuration repository, is by launching it with `spring.config.name=configserver` (there is a `configserver.yml` in the Config Server jar). Another is to use your own `application.properties`, as shown in the following example:

application.properties

```
server.port: 8888
spring.cloud.config.server.git.uri: file://${user.home}/config-repo
```

where `${user.home}/config-repo` is a git repository containing YAML and properties files.



On Windows, you need an extra `"/` in the file URL if it is absolute with a drive prefix (for example, `/${user.home}/config-repo`).

The following listing shows a recipe for creating the git repository in the preceding example:



```
$ cd $HOME
$ mkdir config-repo
$ cd config-repo
$ git init .
$ echo info.foo: bar > application.properties
$ git add -A .
$ git commit -m "Add application.properties"
```



Using the local filesystem for your git repository is intended for testing only. You should use a server to host your configuration repositories in production.



The initial clone of your configuration repository can be quick and efficient if you keep only text files in it. If you store binary files, especially large ones, you may experience delays on the first request for configuration or encounter out of memory errors in the server.

4.2.1. Environment Repository

Where should you store the configuration data for the Config Server? The strategy that governs this behaviour is the `EnvironmentRepository`, serving `Environment` objects. This `Environment` is a shallow copy of the domain from the Spring `Environment` (including `propertySources` as the main feature). The `Environment` resources are parametrized by three variables:

- `{application}`, which maps to `spring.application.name` on the client side.
- `{profile}`, which maps to `spring.profiles.active` on the client (comma-separated list).
- `{label}`, which is a server side feature labelling a "versioned" set of config files.

Repository implementations generally behave like a Spring Boot application, loading configuration files from a `spring.config.name` equal to the `{application}` parameter, and `spring.profiles.active` equal to the `{profiles}` parameter. Precedence rules for profiles are also the same as in a regular Spring Boot application: Active profiles take precedence over defaults, and, if there are multiple profiles, the last one wins (similar to adding entries to a `Map`).

The following sample client application has this bootstrap configuration:

bootstrap.yml

```
spring:
  application:
    name: foo
  profiles:
    active: dev,mysql
```


(As usual with a Spring Boot application, these properties could also be set by environment variables or command line arguments).

If the repository is file-based, the server creates an `Environment` from `application.yml` (shared between all clients) and `foo.yml` (with `foo.yml` taking precedence). If the YAML files have documents inside them that point to Spring profiles, those are applied with higher precedence (in order of the profiles listed). If there are profile-specific YAML (or properties) files, these are also applied with higher precedence than the defaults. Higher precedence translates to a `PropertySource` listed earlier in the `Environment`. (These same rules apply in a standalone Spring Boot application.)

You can set `spring.cloud.config.server.accept-empty` to `false` so that Server would return a HTTP 404 status, if the application is not found. By default, this flag is set to `true`.

Git Backend

The default implementation of `EnvironmentRepository` uses a Git backend, which is very convenient for managing upgrades and physical environments and for auditing changes. To change the location of the repository, you can set the `spring.cloud.config.server.git.uri` configuration property in the Config Server (for example in `application.yml`). If you set it with a `file:` prefix, it should work from a local repository so that you can get started quickly and easily without a server. However, in that case, the server operates directly on the local repository without cloning it (it does not matter if it is not bare because the Config Server never makes changes to the "remote" repository). To scale the Config Server up and make it highly available, you need to have all instances of the server pointing to the same repository, so only a shared file system would work. Even in that case, it is better to use the `ssh:` protocol for a shared filesystem repository, so that the server can clone it and use a local working copy as a cache.

This repository implementation maps the `{label}` parameter of the HTTP resource to a git label (commit id, branch name, or tag). If the git branch or tag name contains a slash (`/`), then the label in the HTTP URL should instead be specified with the special string `(_)` (to avoid ambiguity with other URL paths). For example, if the label is `foo/bar`, replacing the slash would result in the following label: `foo(_)``bar`. The inclusion of the special string `(_)` can also be applied to the `{application}` parameter. If you use a command-line client such as `curl`, be careful with the brackets in the URL — you should escape them from the shell with single quotes (`'`).

Skipping SSL Certificate Validation

The configuration server's validation of the Git server's SSL certificate can be disabled by setting the `git.skipSslValidation` property to `true` (default is `false`).

```
spring:
  cloud:
    config:
      server:
        git:
          uri: https://example.com/my/repo
          skipSslValidation: true
```

Setting HTTP Connection Timeout

You can configure the time, in seconds, that the configuration server will wait to acquire an HTTP connection. Use the `git.timeout` property.

```
spring:
  cloud:
    config:
      server:
        git:
          uri: https://example.com/my/repo
          timeout: 4
```

Placeholders in Git URI

Spring Cloud Config Server supports a git repository URL with placeholders for the `{application}` and `{profile}` (and `{label}` if you need it, but remember that the label is applied as a git label anyway). So you can support a “one repository per application” policy by using a structure similar to the following:

```
spring:
  cloud:
    config:
      server:
        git:
          uri: https://github.com/myorg/{application}
```

You can also support a “one repository per profile” policy by using a similar pattern but with `{profile}`.

Additionally, using the special string “()” within your `{application}` parameters can enable support for multiple organizations, as shown in the following example:

```
spring:
  cloud:
    config:
      server:
        git:
          uri: https://github.com/{application}
```

where `{application}` is provided at request time in the following format: `organization(_)application`.

Pattern Matching and Multiple Repositories

Spring Cloud Config also includes support for more complex requirements with pattern matching on the application and profile name. The pattern format is a comma-separated list of `{application}/{profile}` names with wildcards (note that a pattern beginning with a wildcard may

need to be quoted), as shown in the following example:

```
spring:
  cloud:
    config:
      server:
        git:
          uri: https://github.com/spring-cloud-samples/config-repo
          repos:
            simple: https://github.com/simple/config-repo
            special:
              pattern: special*/dev*,*special*/dev*
              uri: https://github.com/special/config-repo
            local:
              pattern: local*
              uri: file:/home/configsvc/config-repo
```

If `{application}/{profile}` does not match any of the patterns, it uses the default URI defined under `spring.cloud.config.server.git.uri`. In the above example, for the “simple” repository, the pattern is `simple/*` (it only matches one application named `simple` in all profiles). The “local” repository matches all application names beginning with `local` in all profiles (the `/*` suffix is added automatically to any pattern that does not have a profile matcher).



The “one-liner” short cut used in the “simple” example can be used only if the only property to be set is the URI. If you need to set anything else (credentials, pattern, and so on) you need to use the full form.

The `pattern` property in the repo is actually an array, so you can use a YAML array (or `[0]`, `[1]`, etc. suffixes in properties files) to bind to multiple patterns. You may need to do so if you are going to run apps with multiple profiles, as shown in the following example:

```
spring:
  cloud:
    config:
      server:
        git:
          uri: https://github.com/spring-cloud-samples/config-repo
          repos:
            development:
              pattern:
                - '*/development'
                - '*/staging'
              uri: https://github.com/development/config-repo
            staging:
              pattern:
                - '*/qa'
                - '*/production'
              uri: https://github.com/staging/config-repo
```



Spring Cloud guesses that a pattern containing a profile that does not end in `*` implies that you actually want to match a list of profiles starting with this pattern (so `*/staging` is a shortcut for `["*/staging", "*/staging,*"]`, and so on). This is common where, for instance, you need to run applications in the “development” profile locally but also the “cloud” profile remotely.

Every repository can also optionally store config files in sub-directories, and patterns to search for those directories can be specified as `searchPaths`. The following example shows a config file at the top level:

```
spring:
  cloud:
    config:
      server:
        git:
          uri: https://github.com/spring-cloud-samples/config-repo
          searchPaths: foo,bar*
```

In the preceding example, the server searches for config files in the top level and in the `foo/` sub-directory and also any sub-directory whose name begins with `bar`.

By default, the server clones remote repositories when configuration is first requested. The server can be configured to clone the repositories at startup, as shown in the following top-level example:

```
spring:
  cloud:
    config:
      server:
        git:
          uri: https://git/common/config-repo.git
          repos:
            team-a:
              pattern: team-a-*
              cloneOnStart: true
              uri: https://git/team-a/config-repo.git
            team-b:
              pattern: team-b-*
              cloneOnStart: false
              uri: https://git/team-b/config-repo.git
            team-c:
              pattern: team-c-*
              uri: https://git/team-a/config-repo.git
```

In the preceding example, the server clones team-a’s config-repo on startup, before it accepts any requests. All other repositories are not cloned until configuration from the repository is requested.



Setting a repository to be cloned when the Config Server starts up can help to identify a misconfigured configuration source (such as an invalid repository URI) quickly, while the Config Server is starting up. With `cloneOnStart` not enabled for a configuration source, the Config Server may start successfully with a misconfigured or invalid configuration source and not detect an error until an application requests configuration from that configuration source.

Authentication

To use HTTP basic authentication on the remote repository, add the `username` and `password` properties separately (not in the URL), as shown in the following example:

```
spring:
  cloud:
    config:
      server:
        git:
          uri: https://github.com/spring-cloud-samples/config-repo
          username: trolley
          password: strongpassword
```

If you do not use HTTPS and user credentials, SSH should also work out of the box when you store keys in the default directories (`~/.ssh`) and the URI points to an SSH location, such as `git@github.com:configuration/cloud-configuration`. It is important that an entry for the Git server be present in the `~/.ssh/known_hosts` file and that it is in `ssh-rsa` format. Other formats (such as `ecdsa-sha2-nistp256`) are not supported. To avoid surprises, you should ensure that only one entry is present in the `known_hosts` file for the Git server and that it matches the URL you provided to the config server. If you use a hostname in the URL, you want to have exactly that (not the IP) in the `known_hosts` file. The repository is accessed by using JGit, so any documentation you find on that should be applicable. HTTPS proxy settings can be set in `~/.git/config` or (in the same way as for any other JVM process) with system properties (`-Dhttps.proxyHost` and `-Dhttps.proxyPort`).



If you do not know where your `~/.git` directory is, use `git config --global` to manipulate the settings (for example, `git config --global http.sslVerify false`).

JGit requires RSA keys in PEM format. Below is an example `ssh-keygen` (from `openssh`) command that will generate a key in the correct format:

```
ssh-keygen -m PEM -t rsa -b 4096 -f ~/config_server_deploy_key.rsa
```

Warning: When working with SSH keys, the expected ssh private-key must begin with `-----BEGIN RSA PRIVATE KEY-----`. If the key starts with `-----BEGIN OPENSSSH PRIVATE KEY-----` then the RSA key will not load when `spring-cloud-config` server is started. The error looks like:

```
- Error in object 'spring.cloud.config.server.git': codes
[PrivateKeyIsValid.spring.cloud.config.server.git,PrivateKeyIsValid]; arguments
[org.springframework.context.support.DefaultMessageSourceResolvable: codes
[spring.cloud.config.server.git.,]; arguments []]; default message []]; default message
[Property 'spring.cloud.config.server.git.privateKey' is not a valid private key]
```

To correct the above error the RSA key must be converted to PEM format. An example using openssh is provided above for generating a new key in the appropriate format.

Authentication with AWS CodeCommit

Spring Cloud Config Server also supports [AWS CodeCommit](#) authentication. AWS CodeCommit uses an authentication helper when using Git from the command line. This helper is not used with the JGit library, so a JGit CredentialProvider for AWS CodeCommit is created if the Git URI matches the AWS CodeCommit pattern. AWS CodeCommit URIs follow this pattern://git-codecommit.\${AWS_REGION}.amazonaws.com/\${repopath}.

If you provide a username and password with an AWS CodeCommit URI, they must be the [AWS accessKeyId](#) and [secretAccessKey](#) that provide access to the repository. If you do not specify a username and password, the accessKeyId and secretAccessKey are retrieved by using the [AWS Default Credential Provider Chain](#).

If your Git URI matches the CodeCommit URI pattern (shown earlier), you must provide valid AWS credentials in the username and password or in one of the locations supported by the default credential provider chain. AWS EC2 instances may use [IAM Roles for EC2 Instances](#).



The `aws-java-sdk-core` jar is an optional dependency. If the `aws-java-sdk-core` jar is not on your classpath, the AWS Code Commit credential provider is not created, regardless of the git server URI.

Authentication with Google Cloud Source

Spring Cloud Config Server also supports authenticating against [Google Cloud Source](#) repositories.

If your Git URI uses the `http` or `https` protocol and the domain name is `source.developers.google.com`, the Google Cloud Source credentials provider will be used. A Google Cloud Source repository URI has the format `source.developers.google.com/p/${GCP_PROJECT}/r/${REPO}`. To obtain the URI for your repository, click on "Clone" in the Google Cloud Source UI, and select "Manually generated credentials". Do not generate any credentials, simply copy the displayed URI.

The Google Cloud Source credentials provider will use Google Cloud Platform application default credentials. See [Google Cloud SDK documentation](#) on how to create application default credentials for a system. This approach will work for user accounts in dev environments and for service accounts in production environments.



`com.google.auth:google-auth-library-oauth2-http` is an optional dependency. If the `google-auth-library-oauth2-http` jar is not on your classpath, the Google Cloud Source credential provider is not created, regardless of the git server URI.

Git SSH configuration using properties

By default, the JGit library used by Spring Cloud Config Server uses SSH configuration files such as `~/.ssh/known_hosts` and `/etc/ssh/ssh_config` when connecting to Git repositories by using an SSH URI. In cloud environments such as Cloud Foundry, the local filesystem may be ephemeral or not easily accessible. For those cases, SSH configuration can be set by using Java properties. In order to activate property-based SSH configuration, the `spring.cloud.config.server.git.ignoreLocalSshSettings` property must be set to `true`, as shown in the following example:

```
spring:
  cloud:
    config:
      server:
        git:
          uri: git@gitserver.com:team/repo1.git
          ignoreLocalSshSettings: true
          hostKey: someHostKey
          hostKeyAlgorithm: ssh-rsa
          privateKey: |
            -----BEGIN RSA PRIVATE KEY-----
MIIEPgIBAAKCAQEAx4UbaDzY5xjW6hc9jwN0mX33XpTDVW9WqHp5AKaRbtAC3DqX
IXFMPgw3K45jxRb93f8tv9vL3rD9CUG1Gv4FM+o7ds7FRES5RTjv2RT/JVNJCqoF
o18+ngLqRZCyBtQN7zYByWMRirPGoDUqdPYrj2yq+ObBBNhg5N+h0wKjjpzdj2Ud
117R+wxIqmJo1IYyy16xS8WsJyQuyC01L456qkd5BDZ0Ag8j2X9H9D5220Ln7s9i
oezTipXipS7p7Jekf3Ywx6abJw0mB0rX79dV4qiNcGgzATnG1PkXxqt76VhcGa0W
DDVHEEYGbSQ6hIGSh0I7BQun0aLRZojfE3gqHQIDAQABAoIBAQCZmGrk8BK6tXCd
fY6yTiKxFzwb38IQP0oIuWNRq0+9Xt+NsyPviLHkXfXXCKKU4zUHeIGVRq5MN9b
B056/RrcQHh0oJdUWuOV2qMqJvPUtC0CpGkD+valhfD75MxoXU7s3FK7yxy3rsG
EmfA6tHV8/4a5umo5TqSd2YTm5B19AhRqiuUVI1wTB41DjULUGiMYrnYrhzQ1Vvj
5MjnKTLYu3V8PoYDfv1GmxPPH6vlpafXEeEYN8VB97e5x3DGHjZ5UrurAmTLtd08
+AahyoKsIY612TkkQthJl7FJAwnCGMgY6podzzvzICLFmmTXYiZ/28I4BX/m0Se
pZVnfRixAoGBA06UiwT40/PKs53mCEWngs1SCsh9oGAaLTf/XdvMns5VmuyyAyKG
```

```

ti80l5wqBmi4GIUzjbgUvSut+IowIrG3f5tN85wpjQ1UGVcpTnl5Qo9xaS1PFScQ
xrtWZ9eNj2TsIAMp/svJsyGG30ibxfnuAIPsXNQIJPwRlW3irzpgGvX/AoGBANYW
dnhshUcEHMJi3aXwR120TDnaLoanVGLwLnkqLSYUZA7ZegpKq90UAuBdcEfgdpyi
PhKpeaeIiAaNnFo8m9aoTKr+7I6/uMTlwrVnfrsVTZv3orxjwQV20YIBCVRKD1uX
VhE0ozPZxwwKSPAFocpyWpGHGreGF1AIYBE9UBtjAoGBAI8bfPgJpyFyMiGBj06z
FwLJc/xlFqDusrCHL7abW5qq0L4v3R+FrJw3ZYufzLTVcKfdj6GelwJJ0+8wBm+R
gTKYJIteHt48duLIftDyIphGvm9+I1MGhh5zKuCqIhxIYr9jHloBB7kRm0rPvYY4
VAykNgyDvtAVODP+4m6JvhjAoGBALbtTqErKN47V0+JJpapLnF0KxGrqeGIjIRV
cYA6V4WYGr7NeIfesecf0C356PyhgPfpCVyEztlvwTKb3RzIT1TZN8fH4YBr6Ee
KTbtJefRFhVUjQqnucAvfGi29f+9oE3Ei9f7wA+H35ocF6JvTYUsHNMIO/3gZ38N
CPjyCma9AoGBAMhsITNe3QcbsXAbdUR00dDsIFVROzyFJ2m40i4KCRM35bC/BIBs
q0TY3we+ERB40U8Z2BvU61QuwaunJ2+uGadHo58VSVdggqAo0BSkH58innKKt96J
69pcVH/4rmlbXdcnNYGm6iu+MlPQk4BUZknHSmVHIFdJ0EPupVaQ8RHT
-----END RSA PRIVATE KEY-----

```

The following table describes the SSH configuration properties.

Table 2. SSH Configuration Properties

Property Name	Remarks
ignoreLocalSshSettings	If true , use property-based instead of file-based SSH config. Must be set at as <code>spring.cloud.config.server.git.ignoreLocalSshSettings</code> , not inside a repository definition.
privateKey	Valid SSH private key. Must be set if <code>ignoreLocalSshSettings</code> is true and Git URI is SSH format.
hostKey	Valid SSH host key. Must be set if <code>hostKeyAlgorithm</code> is also set.
hostKeyAlgorithm	One of <code>ssh-dss</code> , <code>ssh-rsa</code> , <code>ecdsa-sha2-nistp256</code> , <code>ecdsa-sha2-nistp384</code> , or <code>ecdsa-sha2-nistp521</code> . Must be set if <code>hostKey</code> is also set.
strictHostKeyChecking	true or false . If false, ignore errors with host key.
knownHostsFile	Location of custom <code>.known_hosts</code> file.

Property Name	Remarks
preferredAuthentications	Override server authentication method order. This should allow for evading login prompts if server has keyboard-interactive authentication before the publickey method.

Placeholders in Git Search Paths

Spring Cloud Config Server also supports a search path with placeholders for the **{application}** and **{profile}** (and **{label}** if you need it), as shown in the following example:

```
spring:
  cloud:
    config:
      server:
        git:
          uri: https://github.com/spring-cloud-samples/config-repo
          searchPaths: '{application}'
```

The preceding listing causes a search of the repository for files in the same name as the directory (as well as the top level). Wildcards are also valid in a search path with placeholders (any matching directory is included in the search).

Force pull in Git Repositories

As mentioned earlier, Spring Cloud Config Server makes a clone of the remote git repository in case the local copy gets dirty (for example, folder content changes by an OS process) such that Spring Cloud Config Server cannot update the local copy from remote repository.

To solve this issue, there is a **force-pull** property that makes Spring Cloud Config Server force pull from the remote repository if the local copy is dirty, as shown in the following example:

```
spring:
  cloud:
    config:
      server:
        git:
          uri: https://github.com/spring-cloud-samples/config-repo
          force-pull: true
```

If you have a multiple-repositories configuration, you can configure the **force-pull** property per repository, as shown in the following example:

```
spring:
  cloud:
    config:
      server:
        git:
          uri: https://git/common/config-repo.git
          force-pull: true
          repos:
            team-a:
              pattern: team-a-*
              uri: https://git/team-a/config-repo.git
              force-pull: true
            team-b:
              pattern: team-b-*
              uri: https://git/team-b/config-repo.git
              force-pull: true
            team-c:
              pattern: team-c-*
              uri: https://git/team-a/config-repo.git
```



The default value for `force-pull` property is `false`.

Deleting untracked branches in Git Repositories

As Spring Cloud Config Server has a clone of the remote git repository after check-outing branch to local repo (e.g fetching properties by label) it will keep this branch forever or till the next server restart (which creates new local repo). So there could be a case when remote branch is deleted but local copy of it is still available for fetching. And if Spring Cloud Config Server client service starts with `--spring.cloud.config.label=deletedRemoteBranch,master` it will fetch properties from `deletedRemoteBranch` local branch, but not from `master`.

In order to keep local repository branches clean and up to remote - `deleteUntrackedBranches` property could be set. It will make Spring Cloud Config Server **force** delete untracked branches from local repository. Example:

```
spring:
  cloud:
    config:
      server:
        git:
          uri: https://github.com/spring-cloud-samples/config-repo
          deleteUntrackedBranches: true
```



The default value for `deleteUntrackedBranches` property is `false`.

Git Refresh Rate

You can control how often the config server will fetch updated configuration data from your Git backend by using `spring.cloud.config.server.git.refreshRate`. The value of this property is specified in seconds. By default the value is 0, meaning the config server will fetch updated configuration from the Git repo every time it is requested.

Version Control Backend Filesystem Use



With VCS-based backends (git, svn), files are checked out or cloned to the local filesystem. By default, they are put in the system temporary directory with a prefix of `config-repo-`. On linux, for example, it could be `/tmp/config-repo-<randomid>`. Some operating systems **routinely clean out** temporary directories. This can lead to unexpected behavior, such as missing properties. To avoid this problem, change the directory that Config Server uses by setting `spring.cloud.config.server.git.basedir` or `spring.cloud.config.server.svn.basedir` to a directory that does not reside in the system temp structure.

File System Backend

There is also a “native” profile in the Config Server that does not use Git but loads the config files from the local classpath or file system (any static URL you want to point to with `spring.cloud.config.server.native.searchLocations`). To use the native profile, launch the Config Server with `spring.profiles.active=native`.



Remember to use the `file:` prefix for file resources (the default without a prefix is usually the classpath). As with any Spring Boot configuration, you can embed `${}`-style environment placeholders, but remember that absolute paths in Windows require an extra `/` (for example, `/${user.home}/config-repo`).



The default value of the `searchLocations` is identical to a local Spring Boot application (that is, `[classpath:/, classpath:/config, file:./, file:./config]`). This does not expose the `application.properties` from the server to all clients, because any property sources present in the server are removed before being sent to the client.



A filesystem backend is great for getting started quickly and for testing. To use it in production, you need to be sure that the file system is reliable and shared across all instances of the Config Server.

The search locations can contain placeholders for `{application}`, `{profile}`, and `{label}`. In this way, you can segregate the directories in the path and choose a strategy that makes sense for you (such as subdirectory per application or subdirectory per profile).

If you do not use placeholders in the search locations, this repository also appends the `{label}` parameter of the HTTP resource to a suffix on the search path, so properties files are loaded from each search location **and** a subdirectory with the same name as the label (the labelled properties

take precedence in the Spring Environment). Thus, the default behaviour with no placeholders is the same as adding a search location ending with `/{label}/`. For example, `file:/tmp/config` is the same as `file:/tmp/config,file:/tmp/config/{label}`. This behavior can be disabled by setting `spring.cloud.config.server.native.addLabelLocations=false`.

Vault Backend

Spring Cloud Config Server also supports [Vault](#) as a backend.

Vault is a tool for securely accessing secrets. A secret is anything that to which you want to tightly control access, such as API keys, passwords, certificates, and other sensitive information. Vault provides a unified interface to any secret while providing tight access control and recording a detailed audit log.

For more information on Vault, see the [Vault quick start guide](#).

To enable the config server to use a Vault backend, you can run your config server with the `vault` profile. For example, in your config server's `application.properties`, you can add `spring.profiles.active=vault`.

By default, the config server assumes that your Vault server runs at `127.0.0.1:8200`. It also assumes that the name of backend is `secret` and the key is `application`. All of these defaults can be configured in your config server's `application.properties`. The following table describes configurable Vault properties:

Name	Default Value
host	127.0.0.1
port	8200
scheme	http
backend	secret
defaultKey	application
profileSeparator	,
kvVersion	1
skipSslValidation	false
timeout	5
namespace	null



All of the properties in the preceding table must be prefixed with `spring.cloud.config.server.vault` or placed in the correct Vault section of a composite configuration.

All configurable properties can be found in `org.springframework.cloud.config.server.environment.VaultEnvironmentProperties`.



Vault 0.10.0 introduced a versioned key-value backend (k/v backend version 2) that exposes a different API than earlier versions, it now requires a `data/` between the mount path and the actual context path and wraps secrets in a `data` object. Setting `spring.cloud.config.server.vault.kv-version=2` will take this into account.

Optionally, there is support for the Vault Enterprise `X-Vault-Namespace` header. To have it sent to Vault set the `namespace` property.

With your config server running, you can make HTTP requests to the server to retrieve values from the Vault backend. To do so, you need a token for your Vault server.

First, place some data in you Vault, as shown in the following example:

```
$ vault kv put secret/application foo=bar baz=bam
$ vault kv put secret/myapp foo=myappsbar
```

Second, make an HTTP request to your config server to retrieve the values, as shown in the following example:

```
$ curl -X "GET" "http://localhost:8888/myapp/default" -H "X-Config-Token: yourtoken"
```

You should see a response similar to the following:

```
{
  "name":"myapp",
  "profiles":[
    "default"
  ],
  "label":null,
  "version":null,
  "state":null,
  "propertySources":[
    {
      "name":"vault:myapp",
      "source":{"foo":"myappsbar"}
    },
    {
      "name":"vault:application",
      "source":{"baz":"bam", "foo":"bar"}
    }
  ]
}
```

The default way for a client to provide the necessary authentication to let Config Server talk to Vault is to set the X-Config-Token header. However, you can instead omit the header and configure the authentication in the server, by setting the same configuration properties as Spring Cloud Vault. The property to set is `spring.cloud.config.server.vault.authentication`. It should be set to one of the supported authentication methods. You may also need to set other properties specific to the authentication method you use, by using the same property names as documented for `spring.cloud.vault` but instead using the `spring.cloud.config.server.vault` prefix. See the [Spring Cloud Vault Reference Guide](#) for more detail.



If you omit the X-Config-Token header and use a server property to set the authentication, the Config Server application needs an additional dependency on Spring Vault to enable the additional authentication options. See the [Spring Vault Reference Guide](#) for how to add that dependency.

Multiple Properties Sources

When using Vault, you can provide your applications with multiple properties sources. For example, assume you have written data to the following paths in Vault:

```
secret/myApp,dev
secret/myApp
secret/application,dev
secret/application
```

Properties written to `secret/application` are available to [all applications using the Config Server](#). An application with the name, `myApp`, would have any properties written to `secret/myApp` and `secret/application` available to it. When `myApp` has the `dev` profile enabled, properties written to all of the above paths would be available to it, with properties in the first path in the list taking priority over the others.

Accessing Backends Through a Proxy

The configuration server can access a Git or Vault backend through an HTTP or HTTPS proxy. This behavior is controlled for either Git or Vault by settings under `proxy.http` and `proxy.https`. These settings are per repository, so if you are using a [composite environment repository](#) you must configure proxy settings for each backend in the composite individually. If using a network which requires separate proxy servers for HTTP and HTTPS URLs, you can configure both the HTTP and the HTTPS proxy settings for a single backend.

The following table describes the proxy configuration properties for both HTTP and HTTPS proxies. All of these properties must be prefixed by `proxy.http` or `proxy.https`.

Table 3. Proxy Configuration Properties

Property Name	Remarks
<code>host</code>	The host of the proxy.
<code>port</code>	The port with which to access the proxy.

Property Name	Remarks
nonProxyHosts	Any hosts which the configuration server should access outside the proxy. If values are provided for both <code>proxy.http.nonProxyHosts</code> and <code>proxy.https.nonProxyHosts</code> , the <code>proxy.http</code> value will be used.
username	The username with which to authenticate to the proxy. If values are provided for both <code>proxy.http.username</code> and <code>proxy.https.username</code> , the <code>proxy.http</code> value will be used.
password	The password with which to authenticate to the proxy. If values are provided for both <code>proxy.http.password</code> and <code>proxy.https.password</code> , the <code>proxy.http</code> value will be used.

The following configuration uses an HTTPS proxy to access a Git repository.

```
spring:
  profiles:
    active: git
  cloud:
    config:
      server:
        git:
          uri: https://github.com/spring-cloud-samples/config-repo
          proxy:
            https:
              host: my-proxy.host.io
              password: myproxypassword
              port: '3128'
              username: myproxyusername
              nonProxyHosts: example.com
```

Sharing Configuration With All Applications

Sharing configuration between all applications varies according to which approach you take, as described in the following topics:

- [File Based Repositories](#)
- [Vault Server](#)

File Based Repositories

With file-based (git, svn, and native) repositories, resources with file names in `application*` (`application.properties`, `application.yml`, `application-*.properties`, and so on) are shared between all client applications. You can use resources with these file names to configure global defaults and have them be overridden by application-specific files as necessary.

The [property overrides](#) feature can also be used for setting global defaults, with placeholders applications allowed to override them locally.



With the “native” profile (a local file system backend) , you should use an explicit search location that is not part of the server’s own configuration. Otherwise, the `application*` resources in the default search locations get removed because they are part of the server.

Vault Server

When using Vault as a backend, you can share configuration with all applications by placing configuration in `secret/application`. For example, if you run the following Vault command, all applications using the config server will have the properties `foo` and `baz` available to them:

```
$ vault write secret/application foo=bar baz=bam
```

CredHub Server

When using CredHub as a backend, you can share configuration with all applications by placing configuration in `/application/` or by placing it in the `default` profile for the application. For example, if you run the following CredHub command, all applications using the config server will have the properties `shared.color1` and `shared.color2` available to them:

```
credhub set --name "/application/profile/master/shared" --type=json  
value: {"shared.color1": "blue", "shared.color": "red"}
```

```
credhub set --name "/my-app/default/master/more-shared" --type=json  
value: {"shared.word1": "hello", "shared.word2": "world"}
```

JDBC Backend

Spring Cloud Config Server supports JDBC (relational database) as a backend for configuration properties. You can enable this feature by adding `spring-jdbc` to the classpath and using the `jdbc` profile or by adding a bean of type `JdbcEnvironmentRepository`. If you include the right dependencies on the classpath (see the user guide for more details on that), Spring Boot configures a data source.

The database needs to have a table called `PROPERTIES` with columns called `APPLICATION`, `PROFILE`, and `LABEL` (with the usual `Environment` meaning), plus `KEY` and `VALUE` for the key and value pairs in `Properties` style. All fields are of type `String` in Java, so you can make them `VARCHAR` of whatever length you need. Property values behave in the same way as they would if they came from Spring Boot properties files named `{application}-{profile}.properties`, including all the encryption and decryption, which will be applied as post-processing steps (that is, not in the repository implementation directly).

Redis Backend

Spring Cloud Config Server supports Redis as a backend for configuration properties. You can enable this feature by adding a dependency to [Spring Data Redis](#).

pom.xml

```
<dependencies>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-redis</artifactId>
  </dependency>
</dependencies>
```

The following configuration uses Spring Data `RedisTemplate` to access a Redis. We can use `spring.redis.*` properties to override default connection settings.

```
spring:
  profiles:
    active: redis
  redis:
    host: redis
    port: 16379
```

The properties should be stored as fields in a hash. The name of hash should be the same as `spring.application.name` property or conjunction of `spring.application.name` and `spring.profiles.active[n]`.

```
HMSET sample-app server.port "8100" sample.topic.name "test" test.property1
"property1"
```

After executing the command visible above a hash should contain the following keys with values:

```
HGETALL sample-app
{
  "server.port": "8100",
  "sample.topic.name": "test",
  "test.property1": "property1"
}
```



When no profile is specified `default` will be used.

AWS S3 Backend

Spring Cloud Config Server supports AWS S3 as a backend for configuration properties. You can enable this feature by adding a dependency to the [AWS Java SDK For Amazon S3](#).

pom.xml

```
<dependencies>
  <dependency>
    <groupId>com.amazonaws</groupId>
    <artifactId>aws-java-sdk-s3</artifactId>
  </dependency>
</dependencies>
```

The following configuration uses the AWS S3 client to access configuration files. We can use `spring.awss3.*` properties to select the bucket where your configuration is stored.

```
spring:
  profiles:
    active: awss3
  cloud:
    config:
      server:
        awss3:
          region: us-east-1
          bucket: bucket1
```

It is also possible to specify an AWS URL to [override the standard endpoint](#) of your S3 service with `spring.awss3.endpoint`. This allows support for beta regions of S3, and other S3 compatible storage APIs.

Credentials are found using the [Default AWS Credential Provider Chain](#). Versioned and encrypted buckets are supported without further configuration.

Configuration files are stored in your bucket as `{application}-{profile}.properties`, `{application}-{profile}.yaml` or `{application}-{profile}.json`. An optional label can be provided to specify a directory path to the file.



When no profile is specified `default` will be used.

CredHub Backend

Spring Cloud Config Server supports [CredHub](#) as a backend for configuration properties. You can enable this feature by adding a dependency to [Spring CredHub](#).

pom.xml

```
<dependencies>
  <dependency>
    <groupId>org.springframework.credhub</groupId>
    <artifactId>spring-credhub-starter</artifactId>
  </dependency>
</dependencies>
```

The following configuration uses mutual TLS to access a CredHub:

```
spring:
  profiles:
    active: credhub
  cloud:
    config:
      server:
        credhub:
          url: https://credhub:8844
```

The properties should be stored as JSON, such as:

```
credhub set --name "/demo-app/default/master/toggles" --type=json
value: {"toggle.button": "blue", "toggle.link": "red"}
```

```
credhub set --name "/demo-app/default/master/abs" --type=json
value: {"marketing.enabled": true, "external.enabled": false}
```

All client applications with the name `spring.cloud.config.name=demo-app` will have the following properties available to them:

```
{
  toggle.button: "blue",
  toggle.link: "red",
  marketing.enabled: true,
  external.enabled: false
}
```



When no profile is specified `default` will be used and when no label is specified `master` will be used as a default value. NOTE: Values added to `application` will be shared by all the applications.

OAuth 2.0

You can authenticate with [OAuth 2.0](#) using [UAA](#) as a provider.

pom.xml

```
<dependencies>
  <dependency>
    <groupId>org.springframework.security</groupId>
    <artifactId>spring-security-config</artifactId>
  </dependency>
  <dependency>
    <groupId>org.springframework.security</groupId>
    <artifactId>spring-security-oauth2-client</artifactId>
  </dependency>
</dependencies>
```

The following configuration uses OAuth 2.0 and UAA to access a CredHub:

```
spring:
  profiles:
    active: credhub
  cloud:
    config:
      server:
        credhub:
          url: https://credhub:8844
          oauth2:
            registration-id: credhub-client
  security:
    oauth2:
      client:
        registration:
          credhub-client:
            provider: uaa
            client-id: credhub_config_server
            client-secret: asecret
            authorization-grant-type: client_credentials
        provider:
          uaa:
            token-uri: https://uaa:8443/oauth/token
```



The used UAA client-id should have `credhub.read` as scope.

Composite Environment Repositories

In some scenarios, you may wish to pull configuration data from multiple environment repositories. To do so, you can enable the `composite` profile in your configuration server's application properties or YAML file. If, for example, you want to pull configuration data from a Subversion repository as well as two Git repositories, you can set the following properties for your configuration server:

```

spring:
  profiles:
    active: composite
  cloud:
    config:
      server:
        composite:
          -
            type: svn
            uri: file:///path/to/svn/repo
          -
            type: git
            uri: file:///path/to/rex/git/repo
          -
            type: git
            uri: file:///path/to/walter/git/repo

```

Using this configuration, precedence is determined by the order in which repositories are listed under the `composite` key. In the above example, the Subversion repository is listed first, so a value found in the Subversion repository will override values found for the same property in one of the Git repositories. A value found in the `rex` Git repository will be used before a value found for the same property in the `walter` Git repository.

If you want to pull configuration data only from repositories that are each of distinct types, you can enable the corresponding profiles, rather than the `composite` profile, in your configuration server's application properties or YAML file. If, for example, you want to pull configuration data from a single Git repository and a single HashiCorp Vault server, you can set the following properties for your configuration server:

```

spring:
  profiles:
    active: git, vault
  cloud:
    config:
      server:
        git:
          uri: file:///path/to/git/repo
          order: 2
        vault:
          host: 127.0.0.1
          port: 8200
          order: 1

```

Using this configuration, precedence can be determined by an `order` property. You can use the `order` property to specify the priority order for all your repositories. The lower the numerical value of the `order` property, the higher priority it has. The priority order of a repository helps resolve any potential conflicts between repositories that contain values for the same properties.



If your composite environment includes a Vault server as in the previous example, you must include a Vault token in every request made to the configuration server. See [Vault Backend](#).



Any type of failure when retrieving values from an environment repository results in a failure for the entire composite environment.



When using a composite environment, it is important that all repositories contain the same labels. If you have an environment similar to those in the preceding examples and you request configuration data with the `master` label but the Subversion repository does not contain a branch called `master`, the entire request fails.

Custom Composite Environment Repositories

In addition to using one of the environment repositories from Spring Cloud, you can also provide your own `EnvironmentRepository` bean to be included as part of a composite environment. To do so, your bean must implement the `EnvironmentRepository` interface. If you want to control the priority of your custom `EnvironmentRepository` within the composite environment, you should also implement the `Ordered` interface and override the `getOrdered` method. If you do not implement the `Ordered` interface, your `EnvironmentRepository` is given the lowest priority.

Property Overrides

The Config Server has an “overrides” feature that lets the operator provide configuration properties to all applications. The overridden properties cannot be accidentally changed by the application with the normal Spring Boot hooks. To declare overrides, add a map of name-value pairs to `spring.cloud.config.server.overrides`, as shown in the following example:

```
spring:
  cloud:
    config:
      server:
        overrides:
          foo: bar
```

The preceding examples causes all applications that are config clients to read `foo=bar`, independent of their own configuration.



A configuration system cannot force an application to use configuration data in any particular way. Consequently, overrides are not enforceable. However, they do provide useful default behavior for Spring Cloud Config clients.



Normally, Spring environment placeholders with `${}` can be escaped (and resolved on the client) by using backslash (`\`) to escape the `$` or the `{`. For example, `\${app.foo:bar}` resolves to `bar`, unless the app provides its own `app.foo`.



In YAML, you do not need to escape the backslash itself. However, in properties files, you do need to escape the backslash, when you configure the overrides on the server.

You can change the priority of all overrides in the client to be more like default values, letting applications supply their own values in environment variables or System properties, by setting the `spring.cloud.config.overrideNone=true` flag (the default is false) in the remote repository.

4.2.2. Health Indicator

Config Server comes with a Health Indicator that checks whether the configured `EnvironmentRepository` is working. By default, it asks the `EnvironmentRepository` for an application named `app`, the `default` profile, and the default label provided by the `EnvironmentRepository` implementation.

You can configure the Health Indicator to check more applications along with custom profiles and custom labels, as shown in the following example:

```
spring:
  cloud:
    config:
      server:
        health:
          repositories:
            myservice:
              label: mylabel
            myservice-dev:
              name: myservice
              profiles: development
```

You can disable the Health Indicator by setting `spring.cloud.config.server.health.enabled=false`.

4.2.3. Security

You can secure your Config Server in any way that makes sense to you (from physical network security to OAuth2 bearer tokens), because Spring Security and Spring Boot offer support for many security arrangements.

To use the default Spring Boot-configured HTTP Basic security, include Spring Security on the classpath (for example, through `spring-boot-starter-security`). The default is a username of `user` and a randomly generated password. A random password is not useful in practice, so we recommend you configure the password (by setting `spring.security.user.password`) and encrypt it (see below for instructions on how to do that).

4.2.4. Encryption and Decryption



To use the encryption and decryption features you need the full-strength JCE installed in your JVM (it is not included by default). You can download the “Java Cryptography Extension (JCE) Unlimited Strength Jurisdiction Policy Files” from Oracle and follow the installation instructions (essentially, you need to replace the two policy files in the JRE `lib/security` directory with the ones that you downloaded).

If the remote property sources contain encrypted content (values starting with `{cipher}`), they are decrypted before sending to clients over HTTP. The main advantage of this setup is that the property values need not be in plain text when they are “at rest” (for example, in a git repository). If a value cannot be decrypted, it is removed from the property source and an additional property is added with the same key but prefixed with `invalid` and a value that means “not applicable” (usually `<n/a>`). This is largely to prevent cipher text being used as a password and accidentally leaking.

If you set up a remote config repository for config client applications, it might contain an `application.yml` similar to the following:

application.yml

```
spring:
  datasource:
    username: dbuser
    password: '{cipher}FKSAJDFGYOS8F7GLHAKERGFHLSAJ'
```

Encrypted values in a `.properties` file must not be wrapped in quotes. Otherwise, the value is not decrypted. The following example shows values that would work:

application.properties

```
spring.datasource.username: dbuser
spring.datasource.password: {cipher}FKSAJDFGYOS8F7GLHAKERGFHLSAJ
```

You can safely push this plain text to a shared git repository, and the secret password remains protected.

The server also exposes `/encrypt` and `/decrypt` endpoints (on the assumption that these are secured and only accessed by authorized agents). If you edit a remote config file, you can use the Config Server to encrypt values by POSTing to the `/encrypt` endpoint, as shown in the following example:

```
$ curl localhost:8888/encrypt -d mysecret
682bc583f4641835fa2db009355293665d2647dade3375c0ee201de2a49f7bda
```



If the value you encrypt has characters in it that need to be URL encoded, you should use the `--data-urlencode` option to `curl` to make sure they are encoded properly.



Be sure not to include any of the curl command statistics in the encrypted value. Outputting the value to a file can help avoid this problem.

The inverse operation is also available through `/decrypt` (provided the server is configured with a symmetric key or a full key pair), as shown in the following example:

```
$ curl localhost:8888/decrypt -d
682bc583f4641835fa2db009355293665d2647dade3375c0ee201de2a49f7bda
mysecret
```



If you testing with curl, then use `--data-urlencode` (instead of `-d`) or set an explicit `Content-Type: text/plain` to make sure curl encodes the data correctly when there are special characters ('+' is particularly tricky).

Take the encrypted value and add the `{cipher}` prefix before you put it in the YAML or properties file and before you commit and push it to a remote (potentially insecure) store.

The `/encrypt` and `/decrypt` endpoints also both accept paths in the form of `/*/{application}/{profiles}`, which can be used to control cryptography on a per-application (name) and per-profile basis when clients call into the main environment resource.



To control the cryptography in this granular way, you must also provide a `@Bean` of type `TextEncryptorLocator` that creates a different encryptor per name and profiles. The one that is provided by default does not do so (all encryptions use the same key).

The `spring` command line client (with Spring Cloud CLI extensions installed) can also be used to encrypt and decrypt, as shown in the following example:

```
$ spring encrypt mysecret --key foo
682bc583f4641835fa2db009355293665d2647dade3375c0ee201de2a49f7bda
$ spring decrypt --key foo
682bc583f4641835fa2db009355293665d2647dade3375c0ee201de2a49f7bda
mysecret
```

To use a key in a file (such as an RSA public key for encryption), prepend the key value with "@" and provide the file path, as shown in the following example:

```
$ spring encrypt mysecret --key @"${HOME}/.ssh/id_rsa.pub
AQAJpGt3eFZQXwt8tsHAVv/QHiY5sI2dRcR+...
```



The `--key` argument is mandatory (despite having a `--` prefix).

4.2.5. Key Management

The Config Server can use a symmetric (shared) key or an asymmetric one (RSA key pair). The asymmetric choice is superior in terms of security, but it is often more convenient to use a symmetric key since it is a single property value to configure in the `bootstrap.properties`.

To configure a symmetric key, you need to set `encrypt.key` to a secret String (or use the `ENCRYPT_KEY` environment variable to keep it out of plain-text configuration files).



You cannot configure an asymmetric key using `encrypt.key`.

To configure an asymmetric key use a keystore (e.g. as created by the `keytool` utility that comes with the JDK). The keystore properties are `encrypt.keyStore.*` with `*` equal to

Property	Description
<code>encrypt.keyStore.location</code>	Contains a <code>Resource</code> location
<code>encrypt.keyStore.password</code>	Holds the password that unlocks the keystore
<code>encrypt.keyStore.alias</code>	Identifies which key in the store to use
<code>encrypt.keyStore.type</code>	The type of KeyStore to create. Defaults to <code>jks</code> .

The encryption is done with the public key, and a private key is needed for decryption. Thus, in principle, you can configure only the public key in the server if you want to only encrypt (and are prepared to decrypt the values yourself locally with the private key). In practice, you might not want to do decrypt locally, because it spreads the key management process around all the clients, instead of concentrating it in the server. On the other hand, it can be a useful option if your config server is relatively insecure and only a handful of clients need the encrypted properties.

4.2.6. Creating a Key Store for Testing

To create a keystore for testing, you can use a command resembling the following:

```
$ keytool -genkeypair -alias mytestkey -keyalg RSA \  
-dname "CN=Web Server,OU=Unit,O=Organization,L=City,S=State,C=US" \  
-keypass changeme -keystore server.jks -storepass letmein
```



When using JDK 11 or above you may get the following warning when using the command above. In this case you probably want to make sure the `keypass` and `storepass` values match.

```
Warning: Different store and key passwords not supported for PKCS12 KeyStores.  
Ignoring user-specified -keypass value.
```

Put the `server.jks` file in the classpath (for instance) and then, in your `bootstrap.yml`, for the Config Server, create the following settings:

```
encrypt:
  keyStore:
    location: classpath:/server.jks
    password: letmein
    alias: mytestkey
    secret: changeme
```

4.2.7. Using Multiple Keys and Key Rotation

In addition to the `{cipher}` prefix in encrypted property values, the Config Server looks for zero or more `{name:value}` prefixes before the start of the (Base64 encoded) cipher text. The keys are passed to a `TextEncryptorLocator`, which can do whatever logic it needs to locate a `TextEncryptor` for the cipher. If you have configured a keystore (`encrypt.keystore.location`), the default locator looks for keys with aliases supplied by the `key` prefix, with a cipher text like resembling the following:

```
foo:
  bar: `{cipher}{key:testkey}...`
```

The locator looks for a key named "testkey". A secret can also be supplied by using a `{secret:...}` value in the prefix. However, if it is not supplied, the default is to use the keystore password (which is what you get when you build a keystore and do not specify a secret). If you do supply a secret, you should also encrypt the secret using a custom `SecretLocator`.

When the keys are being used only to encrypt a few bytes of configuration data (that is, they are not being used elsewhere), key rotation is hardly ever necessary on cryptographic grounds. However, you might occasionally need to change the keys (for example, in the event of a security breach). In that case, all the clients would need to change their source config files (for example, in git) and use a new `{key:...}` prefix in all the ciphers. Note that the clients need to first check that the key alias is available in the Config Server keystore.



If you want to let the Config Server handle all encryption as well as decryption, the `{name:value}` prefixes can also be added as plain text posted to the `/encrypt` endpoint, .

4.2.8. Serving Encrypted Properties

Sometimes you want the clients to decrypt the configuration locally, instead of doing it in the server. In that case, if you provide the `encrypt.*` configuration to locate a key, you can still have `/encrypt` and `/decrypt` endpoints, but you need to explicitly switch off the decryption of outgoing properties by placing `spring.cloud.config.server.encrypt.enabled=false` in `bootstrap.[yml|properties]`. If you do not care about the endpoints, it should work if you do not configure either the key or the enabled flag.

4.3. Serving Alternative Formats

The default JSON format from the environment endpoints is perfect for consumption by Spring

applications, because it maps directly onto the `Environment` abstraction. If you prefer, you can consume the same data as YAML or Java properties by adding a suffix (".yaml", ".yml" or ".properties") to the resource path. This can be useful for consumption by applications that do not care about the structure of the JSON endpoints or the extra metadata they provide (for example, an application that is not using Spring might benefit from the simplicity of this approach).

The YAML and properties representations have an additional flag (provided as a boolean query parameter called `resolvePlaceholders`) to signal that placeholders in the source documents (in the standard Spring `${...}` form) should be resolved in the output before rendering, where possible. This is a useful feature for consumers that do not know about the Spring placeholder conventions.



There are limitations in using the YAML or properties formats, mainly in relation to the loss of metadata. For example, the JSON is structured as an ordered list of property sources, with names that correlate with the source. The YAML and properties forms are coalesced into a single map, even if the origin of the values has multiple sources, and the names of the original source files are lost. Also, the YAML representation is not necessarily a faithful representation of the YAML source in a backing repository either. It is constructed from a list of flat property sources, and assumptions have to be made about the form of the keys.

4.4. Serving Plain Text

Instead of using the `Environment` abstraction (or one of the alternative representations of it in YAML or properties format), your applications might need generic plain-text configuration files that are tailored to their environment. The Config Server provides these through an additional endpoint at `/{application}/{profile}/{label}/{path}`, where `application`, `profile`, and `label` have the same meaning as the regular environment endpoint, but `path` is a path to a file name (such as `log.xml`). The source files for this endpoint are located in the same way as for the environment endpoints. The same search path is used for properties and YAML files. However, instead of aggregating all matching resources, only the first one to match is returned.

After a resource is located, placeholders in the normal format (`${...}`) are resolved by using the effective `Environment` for the supplied application name, profile, and label. In this way, the resource endpoint is tightly integrated with the environment endpoints.



As with the source files for environment configuration, the `profile` is used to resolve the file name. So, if you want a profile-specific file, `/*/development/*/logback.xml` can be resolved by a file called `logback-development.xml` (in preference to `logback.xml`).



If you do not want to supply the `label` and let the server use the default label, you can supply a `useDefaultLabel` request parameter. Consequently, the preceding example for the `default` profile could be `/sample/default/nginx.conf?useDefaultLabel`.

At present, Spring Cloud Config can serve plaintext for git, SVN, native backends, and AWS S3. The support for git, SVN, and native backends is identical. AWS S3 works a bit differently. The following

sections show how each one works:

- [Git, SVN, and Native Backends](#)
- [AWS S3](#)

4.4.1. Git, SVN, and Native Backends

Consider the following example for a GIT or SVN repository or a native backend:

```
application.yml
nginx.conf
```

The `nginx.conf` might resemble the following listing:

```
server {
    listen          80;
    server_name     ${nginx.server.name};
}
```

`application.yml` might resemble the following listing:

```
nginx:
  server:
    name: example.com
---
spring:
  profiles: development
nginx:
  server:
    name: develop.com
```

The `/sample/default/master/nginx.conf` resource might be as follows:

```
server {
    listen          80;
    server_name     example.com;
}
```

`/sample/development/master/nginx.conf` might be as follows:

```
server {
    listen          80;
    server_name    develop.com;
}
```

4.4.2. AWS S3

To enable serving plain text for AWS s3, the Config Server application needs to include a dependency on Spring Cloud AWS. For details on how to set up that dependency, see the [Spring Cloud AWS Reference Guide](#). Then you need to configure Spring Cloud AWS, as described in the [Spring Cloud AWS Reference Guide](#).

4.4.3. Decrypting Plain Text

By default, encrypted values in plain text files are not decrypted. In order to enable decryption for plain text files, set `spring.cloud.config.server.encrypt.enabled=true` and `spring.cloud.config.server.encrypt.plainTextEncrypt=true` in `bootstrap.[yml|properties]`



Decrypting plain text files is only supported for YAML, JSON, and properties file extensions.

If this feature is enabled, and an unsupported file extension is requested, any encrypted values in the file will not be decrypted.

4.5. Embedding the Config Server

The Config Server runs best as a standalone application. However, if need be, you can embed it in another application. To do so, use the `@EnableConfigServer` annotation. An optional property named `spring.cloud.config.server.bootstrap` can be useful in this case. It is a flag to indicate whether the server should configure itself from its own remote repository. By default, the flag is off, because it can delay startup. However, when embedded in another application, it makes sense to initialize the same way as any other application. When setting `spring.cloud.config.server.bootstrap` to `true` you must also use a [composite environment repository configuration](#). For example

```
spring:
  application:
    name: configserver
  profiles:
    active: composite
  cloud:
    config:
      server:
        composite:
          - type: native
            search-locations: ${HOME}/Desktop/config
        bootstrap: true
```



If you use the bootstrap flag, the config server needs to have its name and repository URI configured in `bootstrap.yml`.

To change the location of the server endpoints, you can (optionally) set `spring.cloud.config.server.prefix` (for example, `/config`), to serve the resources under a prefix. The prefix should start but not end with a `/`. It is applied to the `@RequestMapping` in the Config Server (that is, underneath the Spring Boot `server.servletPath` and `server.contextPath` prefixes).

If you want to read the configuration for an application directly from the backend repository (instead of from the config server), you basically want an embedded config server with no endpoints. You can switch off the endpoints entirely by not using the `@EnableConfigServer` annotation (set `spring.cloud.config.server.bootstrap=true`).

4.6. Push Notifications and Spring Cloud Bus

Many source code repository providers (such as Github, Gitlab, Gitea, Gitee, Gogs, or Bitbucket) notify you of changes in a repository through a webhook. You can configure the webhook through the provider's user interface as a URL and a set of events in which you are interested. For instance, [Github](#) uses a POST to the webhook with a JSON body containing a list of commits and a header (`X-Github-Event`) set to `push`. If you add a dependency on the `spring-cloud-config-monitor` library and activate the Spring Cloud Bus in your Config Server, then a `/monitor` endpoint is enabled.

When the webhook is activated, the Config Server sends a `RefreshRemoteApplicationEvent` targeted at the applications it thinks might have changed. The change detection can be strategized. However, by default, it looks for changes in files that match the application name (for example, `foo.properties` is targeted at the `foo` application, while `application.properties` is targeted at all applications). The strategy to use when you want to override the behavior is `PropertyPathNotificationExtractor`, which accepts the request headers and body as parameters and returns a list of file paths that changed.

The default configuration works out of the box with Github, Gitlab, Gitea, Gitee, Gogs or Bitbucket. In addition to the JSON notifications from Github, Gitlab, Gitee, or Bitbucket, you can trigger a change notification by POSTing to `/monitor` with form-encoded body parameters in the pattern of `path={application}`. Doing so broadcasts to applications matching the `{application}` pattern (which

can contain wildcards).



The `RefreshRemoteApplicationEvent` is transmitted only if the `spring-cloud-bus` is activated in both the Config Server and in the client application.



The default configuration also detects filesystem changes in local git repositories. In that case, the webhook is not used. However, as soon as you edit a config file, a refresh is broadcast.

4.7. Spring Cloud Config Client

A Spring Boot application can take immediate advantage of the Spring Config Server (or other external property sources provided by the application developer). It also picks up some additional useful features related to `Environment` change events.

4.7.1. Config First Bootstrap

The default behavior for any application that has the Spring Cloud Config Client on the classpath is as follows: When a config client starts, it binds to the Config Server (through the `spring.cloud.config.uri` bootstrap configuration property) and initializes Spring `Environment` with remote property sources.

The net result of this behavior is that all client applications that want to consume the Config Server need a `bootstrap.yml` (or an environment variable) with the server address set in `spring.cloud.config.uri` (it defaults to "http://localhost:8888").

4.7.2. Discovery First Bootstrap

If you use a `DiscoveryClient` implementation, such as Spring Cloud Netflix and Eureka Service Discovery or Spring Cloud Consul, you can have the Config Server register with the Discovery Service. However, in the default “Config First” mode, clients cannot take advantage of the registration.

If you prefer to use `DiscoveryClient` to locate the Config Server, you can do so by setting `spring.cloud.config.discovery.enabled=true` (the default is `false`). The net result of doing so is that client applications all need a `bootstrap.yml` (or an environment variable) with the appropriate discovery configuration. For example, with Spring Cloud Netflix, you need to define the Eureka server address (for example, in `eureka.client.serviceUrl.defaultZone`). The price for using this option is an extra network round trip on startup, to locate the service registration. The benefit is that, as long as the Discovery Service is a fixed point, the Config Server can change its coordinates. The default service ID is `configserver`, but you can change that on the client by setting `spring.cloud.config.discovery.serviceId` (and on the server, in the usual way for a service, such as by setting `spring.application.name`).

The discovery client implementations all support some kind of metadata map (for example, we have `eureka.instance.metadataMap` for Eureka). Some additional properties of the Config Server may need to be configured in its service registration metadata so that clients can connect correctly. If the Config Server is secured with HTTP Basic, you can configure the credentials as `user` and `password`.

Also, if the Config Server has a context path, you can set `configPath`. For example, the following YAML file is for a Config Server that is a Eureka client:

bootstrap.yml

```
eureka:
  instance:
    ...
  metadataMap:
    user: osufhalskjrtl
    password: lvihlszvaorhvlo5847
    configPath: /config
```

4.7.3. Config Client Fail Fast

In some cases, you may want to fail startup of a service if it cannot connect to the Config Server. If this is the desired behavior, set the bootstrap configuration property `spring.cloud.config.fail-fast=true` to make the client halt with an Exception.

4.7.4. Config Client Retry

If you expect that the config server may occasionally be unavailable when your application starts, you can make it keep trying after a failure. First, you need to set `spring.cloud.config.fail-fast=true`. Then you need to add `spring-retry` and `spring-boot-starter-aop` to your classpath. The default behavior is to retry six times with an initial backoff interval of 1000ms and an exponential multiplier of 1.1 for subsequent backoffs. You can configure these properties (and others) by setting the `spring.cloud.config.retry.*` configuration properties.



To take full control of the retry behavior, add a `@Bean` of type `RetryOperationsInterceptor` with an ID of `configServerRetryInterceptor`. Spring Retry has a `RetryInterceptorBuilder` that supports creating one.

4.7.5. Locating Remote Configuration Resources

The Config Service serves property sources from `/{application}/{profile}/{label}`, where the default bindings in the client app are as follows:

- "name" = `${spring.application.name}`
- "profile" = `${spring.profiles.active}` (actually `Environment.getActiveProfiles()`)
- "label" = "master"



When setting the property `${spring.application.name}` do not prefix your app name with the reserved word `application-` to prevent issues resolving the correct property source.

You can override all of them by setting `spring.cloud.config.*` (where `*` is `name`, `profile` or `label`). The `label` is useful for rolling back to previous versions of configuration. With the default Config Server

implementation, it can be a git label, branch name, or commit ID. Label can also be provided as a comma-separated list. In that case, the items in the list are tried one by one until one succeeds. This behavior can be useful when working on a feature branch. For instance, you might want to align the config label with your branch but make it optional (in that case, use `spring.cloud.config.label=myfeature,develop`).

4.7.6. Specifying Multiple Urls for the Config Server

To ensure high availability when you have multiple instances of Config Server deployed and expect one or more instances to be unavailable from time to time, you can either specify multiple URLs (as a comma-separated list under the `spring.cloud.config.uri` property) or have all your instances register in a Service Registry like Eureka (if using Discovery-First Bootstrap mode). Note that doing so ensures high availability only when the Config Server is not running (that is, when the application has exited) or when a connection timeout has occurred. For example, if the Config Server returns a 500 (Internal Server Error) response or the Config Client receives a 401 from the Config Server (due to bad credentials or other causes), the Config Client does not try to fetch properties from other URLs. An error of that kind indicates a user issue rather than an availability problem.

If you use HTTP basic security on your Config Server, it is currently possible to support per-Config Server auth credentials only if you embed the credentials in each URL you specify under the `spring.cloud.config.uri` property. If you use any other kind of security mechanism, you cannot (currently) support per-Config Server authentication and authorization.

4.7.7. Configuring Timeouts

If you want to configure timeout thresholds:

- Read timeouts can be configured by using the property `spring.cloud.config.request-read-timeout`.
- Connection timeouts can be configured by using the property `spring.cloud.config.request-connect-timeout`.

4.7.8. Security

If you use HTTP Basic security on the server, clients need to know the password (and username if it is not the default). You can specify the username and password through the config server URI or via separate username and password properties, as shown in the following example:

bootstrap.yml

```
spring:
  cloud:
    config:
      uri: https://user:secret@myconfig.mycompany.com
```

The following example shows an alternate way to pass the same information:

bootstrap.yml

```
spring:
  cloud:
    config:
      uri: https://myconfig.mycompany.com
      username: user
      password: secret
```

The `spring.cloud.config.password` and `spring.cloud.config.username` values override anything that is provided in the URI.

If you deploy your apps on Cloud Foundry, the best way to provide the password is through service credentials (such as in the URI, since it does not need to be in a config file). The following example works locally and for a user-provided service on Cloud Foundry named `configserver`:

bootstrap.yml

```
spring:
  cloud:
    config:
      uri:
        ${vcap.services.configserver.credentials.uri:http://user:password@localhost:8888}
```

If you use another form of security, you might need to [provide a RestTemplate](#) to the `ConfigServicePropertySourceLocator` (for example, by grabbing it in the bootstrap context and injecting it).

Health Indicator

The Config Client supplies a Spring Boot Health Indicator that attempts to load configuration from the Config Server. The health indicator can be disabled by setting `health.config.enabled=false`. The response is also cached for performance reasons. The default cache time to live is 5 minutes. To change that value, set the `health.config.time-to-live` property (in milliseconds).

Providing A Custom RestTemplate

In some cases, you might need to customize the requests made to the config server from the client. Typically, doing so involves passing special `Authorization` headers to authenticate requests to the server. To provide a custom `RestTemplate`:

1. Create a new configuration bean with an implementation of `PropertySourceLocator`, as shown in the following example:

```
@Configuration
public class CustomConfigServiceBootstrapConfiguration {
    @Bean
    public ConfigServicePropertySourceLocator configServicePropertySourceLocator() {
        ConfigClientProperties clientProperties = configClientProperties();
        ConfigServicePropertySourceLocator configServicePropertySourceLocator = new
        ConfigServicePropertySourceLocator(clientProperties);

        configServicePropertySourceLocator.setRestTemplate(customRestTemplate(clientProperties
        ));
        return configServicePropertySourceLocator;
    }
}
```



For a simplified approach to adding **Authorization** headers, the `spring.cloud.config.headers.*` property can be used instead.

1. In `resources/META-INF`, create a file called `spring.factories` and specify your custom configuration, as shown in the following example:

spring.factories

```
org.springframework.cloud.bootstrap.BootstrapConfiguration =
com.my.config.client.CustomConfigServiceBootstrapConfiguration
```

Vault

When using Vault as a backend to your config server, the client needs to supply a token for the server to retrieve values from Vault. This token can be provided within the client by setting `spring.cloud.config.token` in `bootstrap.yml`, as shown in the following example:

bootstrap.yml

```
spring:
  cloud:
    config:
      token: YourVaultToken
```

4.7.9. Nested Keys In Vault

Vault supports the ability to nest keys in a value stored in Vault, as shown in the following example:

```
echo -n '{"appA": {"secret": "appAsecret"}, "bar": "baz"}' | vault write secret/myapp -
```

This command writes a JSON object to your Vault. To access these values in Spring, you would use the traditional dot(.) annotation, as shown in the following example

```
@Value("${appA.secret}")  
String name = "World";
```

The preceding code would sets the value of the `name` variable to `appAsecret`.

Chapter 5. Spring Cloud Netflix

2020.0.0-M2

This project provides Netflix OSS integrations for Spring Boot apps through autoconfiguration and binding to the Spring Environment and other Spring programming model idioms. With a few simple annotations you can quickly enable and configure the common patterns inside your application and build large distributed systems with battle-tested Netflix components. The patterns provided include Service Discovery (Eureka), Circuit Breaker (Hystrix), Intelligent Routing (Zuul) and Client Side Load Balancing (Ribbon).

5.1. Service Discovery: Eureka Clients

Service Discovery is one of the key tenets of a microservice-based architecture. Trying to hand-configure each client or some form of convention can be difficult to do and can be brittle. Eureka is the Netflix Service Discovery Server and Client. The server can be configured and deployed to be highly available, with each server replicating state about the registered services to the others.

5.1.1. How to Include Eureka Client

To include the Eureka Client in your project, use the starter with a group ID of `org.springframework.cloud` and an artifact ID of `spring-cloud-starter-netflix-eureka-client`. See the [Spring Cloud Project page](#) for details on setting up your build system with the current Spring Cloud Release Train.

5.1.2. Registering with Eureka

When a client registers with Eureka, it provides meta-data about itself—such as host, port, health indicator URL, home page, and other details. Eureka receives heartbeat messages from each instance belonging to a service. If the heartbeat fails over a configurable timetable, the instance is normally removed from the registry.

The following example shows a minimal Eureka client application:

```

@SpringBootApplication
@RestController
public class Application {

    @RequestMapping("/")
    public String home() {
        return "Hello world";
    }

    public static void main(String[] args) {
        new SpringApplicationBuilder(Application.class).web(true).run(args);
    }
}

```

Note that the preceding example shows a normal [Spring Boot](#) application. By having `spring-cloud-starter-netflix-eureka-client` on the classpath, your application automatically registers with the Eureka Server. Configuration is required to locate the Eureka server, as shown in the following example:

application.yml

```

eureka:
  client:
    serviceUrl:
      defaultZone: http://localhost:8761/eureka/

```

In the preceding example, `defaultZone` is a magic string fallback value that provides the service URL for any client that does not express a preference (in other words, it is a useful default).



The `defaultZone` property is case sensitive and requires camel case because the `serviceUrl` property is a `Map<String, String>`. Therefore, the `defaultZone` property does not follow the normal Spring Boot snake-case convention of `default-zone`.

The default application name (that is, the service ID), virtual host, and non-secure port (taken from the `Environment`) are `${spring.application.name}`, `${spring.application.name}` and `${server.port}`, respectively.

Having `spring-cloud-starter-netflix-eureka-client` on the classpath makes the app into both a Eureka “instance” (that is, it registers itself) and a “client” (it can query the registry to locate other services). The instance behaviour is driven by `eureka.instance.*` configuration keys, but the defaults are fine if you ensure that your application has a value for `spring.application.name` (this is the default for the Eureka service ID or VIP).

See [EurekaInstanceConfigBean](#) and [EurekaClientConfigBean](#) for more details on the configurable options.

To disable the Eureka Discovery Client, you can set `eureka.client.enabled` to `false`. Eureka

Discovery Client will also be disabled when `spring.cloud.discovery.enabled` is set to `false`.

5.1.3. Authenticating with the Eureka Server

HTTP basic authentication is automatically added to your eureka client if one of the `eureka.client.serviceUrl.defaultZone` URLs has credentials embedded in it (curl style, as follows: `user:password@localhost:8761/eureka`). For more complex needs, you can create a `@Bean` of type `DiscoveryClientOptionalArgs` and inject `ClientFilter` instances into it, all of which is applied to the calls from the client to the server.



Because of a limitation in Eureka, it is not possible to support per-server basic auth credentials, so only the first set that are found is used.

5.1.4. Status Page and Health Indicator

The status page and health indicators for a Eureka instance default to `/info` and `/health` respectively, which are the default locations of useful endpoints in a Spring Boot Actuator application. You need to change these, even for an Actuator application if you use a non-default context path or servlet path (such as `server.servletPath=/custom`). The following example shows the default values for the two settings:

application.yml

```
eureka:
  instance:
    statusPageUrlPath: ${server.servletPath}/info
    healthCheckUrlPath: ${server.servletPath}/health
```

These links show up in the metadata that is consumed by clients and are used in some scenarios to decide whether to send requests to your application, so it is helpful if they are accurate.



In Dalston it was also required to set the status and health check URLs when changing that management context path. This requirement was removed beginning in Edgware.

5.1.5. Registering a Secure Application

If your app wants to be contacted over HTTPS, you can set two flags in the `EurekaInstanceConfig`:

- `eureka.instance.[nonSecurePortEnabled]=[false]`
- `eureka.instance.[securePortEnabled]=[true]`

Doing so makes Eureka publish instance information that shows an explicit preference for secure communication. The Spring Cloud `DiscoveryClient` always returns a URI starting with `https` for a service configured this way. Similarly, when a service is configured this way, the Eureka (native) instance information has a secure health check URL.

Because of the way Eureka works internally, it still publishes a non-secure URL for the status and

home pages unless you also override those explicitly. You can use placeholders to configure the eureka instance URLs, as shown in the following example:

application.yml

```
eureka:
  instance:
    statusPageUrl: https://${eureka.hostname}/info
    healthCheckUrl: https://${eureka.hostname}/health
    homePageUrl: https://${eureka.hostname}/
```

(Note that `${eureka.hostname}` is a native placeholder only available in later versions of Eureka. You could achieve the same thing with Spring placeholders as well—for example, by using `${eureka.instance.hostName}`.)



If your application runs behind a proxy, and the SSL termination is in the proxy (for example, if you run in Cloud Foundry or other platforms as a service), then you need to ensure that the proxy “forwarded” headers are intercepted and handled by the application. If the Tomcat container embedded in a Spring Boot application has explicit configuration for the 'X-Forwarded-*' headers, this happens automatically. The links rendered by your app to itself being wrong (the wrong host, port, or protocol) is a sign that you got this configuration wrong.

5.1.6. Eureka’s Health Checks

By default, Eureka uses the client heartbeat to determine if a client is up. Unless specified otherwise, the Discovery Client does not propagate the current health check status of the application, per the Spring Boot Actuator. Consequently, after successful registration, Eureka always announces that the application is in 'UP' state. This behavior can be altered by enabling Eureka health checks, which results in propagating application status to Eureka. As a consequence, every other application does not send traffic to applications in states other than 'UP'. The following example shows how to enable health checks for the client:

application.yml

```
eureka:
  client:
    healthcheck:
      enabled: true
```



`eureka.client.healthcheck.enabled=true` should only be set in `application.yml`. Setting the value in `bootstrap.yml` causes undesirable side effects, such as registering in Eureka with an `UNKNOWN` status.

If you require more control over the health checks, consider implementing your own `com.netflix.appinfo.HealthCheckHandler`.

5.1.7. Eureka Metadata for Instances and Clients

It is worth spending a bit of time understanding how the Eureka metadata works, so you can use it in a way that makes sense in your platform. There is standard metadata for information such as hostname, IP address, port numbers, the status page, and health check. These are published in the service registry and used by clients to contact the services in a straightforward way. Additional metadata can be added to the instance registration in the `eureka.instance.metadataMap`, and this metadata is accessible in the remote clients. In general, additional metadata does not change the behavior of the client, unless the client is made aware of the meaning of the metadata. There are a couple of special cases, described later in this document, where Spring Cloud already assigns meaning to the metadata map.

Using Eureka on Cloud Foundry

Cloud Foundry has a global router so that all instances of the same app have the same hostname (other PaaS solutions with a similar architecture have the same arrangement). This is not necessarily a barrier to using Eureka. However, if you use the router (recommended or even mandatory, depending on the way your platform was set up), you need to explicitly set the hostname and port numbers (secure or non-secure) so that they use the router. You might also want to use instance metadata so that you can distinguish between the instances on the client (for example, in a custom load balancer). By default, the `eureka.instance.instanceId` is `vcap.application.instance_id`, as shown in the following example:

application.yml

```
eureka:
  instance:
    hostname: ${vcap.application.uris[0]}
    nonSecurePort: 80
```

Depending on the way the security rules are set up in your Cloud Foundry instance, you might be able to register and use the IP address of the host VM for direct service-to-service calls. This feature is not yet available on Pivotal Web Services ([PWS](#)).

Using Eureka on AWS

If the application is planned to be deployed to an AWS cloud, the Eureka instance must be configured to be AWS-aware. You can do so by customizing the [EurekaInstanceConfigBean](#) as follows:

```
@Bean
@Profile("!default")
public EurekaInstanceConfigBean eurekaInstanceConfig(InetUtils inetUtils) {
    EurekaInstanceConfigBean b = new EurekaInstanceConfigBean(inetUtils);
    AmazonInfo info = AmazonInfo.Builder.newBuilder().autoBuild("eureka");
    b.setDataCenterInfo(info);
    return b;
}
```

Changing the Eureka Instance ID

A vanilla Netflix Eureka instance is registered with an ID that is equal to its host name (that is, there is only one service per host). Spring Cloud Eureka provides a sensible default, which is defined as follows:

```
${spring.cloud.client.hostname}:${spring.application.name}:${spring.application.instance_id:${server.port}}
```

An example is `myhost:myappname:8080`.

By using Spring Cloud, you can override this value by providing a unique identifier in `eureka.instance.instanceId`, as shown in the following example:

application.yml

```
eureka:
  instance:
    instanceId:
      ${spring.application.name}:${vcap.application.instance_id:${spring.application.instance_id:${random.value}}}
```

With the metadata shown in the preceding example and multiple service instances deployed on localhost, the random value is inserted there to make the instance unique. In Cloud Foundry, the `vcap.application.instance_id` is populated automatically in a Spring Boot application, so the random value is not needed.

5.1.8. Using the EurekaClient

Once you have an application that is a discovery client, you can use it to discover service instances from the [Eureka Server](#). One way to do so is to use the native `com.netflix.discovery.EurekaClient` (as opposed to the Spring Cloud `DiscoveryClient`), as shown in the following example:

```
@Autowired
private EurekaClient discoveryClient;

public String serviceUrl() {
    InstanceInfo instance = discoveryClient.getNextServerFromEureka("STORES", false);
    return instance.getHomePageUrl();
}
```



Do not use the `EurekaClient` in a `@PostConstruct` method or in a `@Scheduled` method (or anywhere where the `ApplicationContext` might not be started yet). It is initialized in a `SmartLifecycle` (with `phase=0`), so the earliest you can rely on it being available is in another `SmartLifecycle` with a higher phase.

EurekaClient without Jersey

By default, `EurekaClient` uses Jersey for HTTP communication. If you wish to avoid dependencies

from Jersey, you can exclude it from your dependencies. Spring Cloud auto-configures a transport client based on Spring `RestTemplate`. The following example shows Jersey being excluded:

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
  <exclusions>
    <exclusion>
      <groupId>com.sun.jersey</groupId>
      <artifactId>jersey-client</artifactId>
    </exclusion>
    <exclusion>
      <groupId>com.sun.jersey</groupId>
      <artifactId>jersey-core</artifactId>
    </exclusion>
    <exclusion>
      <groupId>com.sun.jersey.contribs</groupId>
      <artifactId>jersey-apache-client4</artifactId>
    </exclusion>
  </exclusions>
</dependency>
```

5.1.9. Alternatives to the Native Netflix EurekaClient

You need not use the raw Netflix `EurekaClient`. Also, it is usually more convenient to use it behind a wrapper of some sort. Spring Cloud has support for `Feign` (a REST client builder) and `Spring RestTemplate` through the logical Eureka service identifiers (VIPs) instead of physical URLs. To configure Ribbon with a fixed list of physical servers, you can set `<client>.ribbon.listOfServers` to a comma-separated list of physical addresses (or hostnames), where `<client>` is the ID of the client.

You can also use the `org.springframework.cloud.client.discovery.DiscoveryClient`, which provides a simple API (not specific to Netflix) for discovery clients, as shown in the following example:

```
@Autowired
private DiscoveryClient discoveryClient;

public String serviceUrl() {
    List<ServiceInstance> list = discoveryClient.getInstances("STORES");
    if (list != null && list.size() > 0 ) {
        return list.get(0).getUri();
    }
    return null;
}
```

5.1.10. Why Is It so Slow to Register a Service?

Being an instance also involves a periodic heartbeat to the registry (through the client's `serviceUrl`) with a default duration of 30 seconds. A service is not available for discovery by clients until the

instance, the server, and the client all have the same metadata in their local cache (so it could take 3 heartbeats). You can change the period by setting `eureka.instance.leaseRenewalIntervalInSeconds`. Setting it to a value of less than 30 speeds up the process of getting clients connected to other services. In production, it is probably better to stick with the default, because of internal computations in the server that make assumptions about the lease renewal period.

5.1.11. Zones

If you have deployed Eureka clients to multiple zones, you may prefer that those clients use services within the same zone before trying services in another zone. To set that up, you need to configure your Eureka clients correctly.

First, you need to make sure you have Eureka servers deployed to each zone and that they are peers of each other. See the section on [zones and regions](#) for more information.

Next, you need to tell Eureka which zone your service is in. You can do so by using the `metadataMap` property. For example, if `service 1` is deployed to both `zone 1` and `zone 2`, you need to set the following Eureka properties in `service 1`:

Service 1 in Zone 1

```
eureka.instance.metadataMap.zone = zone1
eureka.client.preferSameZoneEureka = true
```

Service 1 in Zone 2

```
eureka.instance.metadataMap.zone = zone2
eureka.client.preferSameZoneEureka = true
```

5.1.12. Refreshing Eureka Clients

By default, the `EurekaClient` bean is refreshable, meaning the Eureka client properties can be changed and refreshed. When a refresh occurs clients will be unregistered from the Eureka server and there might be a brief moment of time where all instance of a given service are not available. One way to eliminate this from happening is to disable the ability to refresh Eureka clients. To do this set `eureka.client.refresh.enable=false`.

5.1.13. Using Eureka with Spring Cloud LoadBalancer

We offer support for the Spring Cloud LoadBalancer `ZonePreferenceServiceInstanceListSupplier`. The `zone` value from the Eureka instance metadata (`eureka.instance.metadataMap.zone`) is used for setting the value of `spring-cloud-loadbalancer-zone` property that is used to filter service instances by zone.

If that is missing and if the `spring.cloud.loadbalancer.eureka.approximateZoneFromHostname` flag is set to `true`, it can use the domain name from the server hostname as a proxy for the zone.

If there is no other source of zone data, then a guess is made, based on the client configuration (as opposed to the instance configuration). We take `eureka.client.availabilityZones`, which is a map from region name to a list of zones, and pull out the first zone for the instance's own region (that is, the `eureka.client.region`, which defaults to "us-east-1", for compatibility with native Netflix).

5.2. Service Discovery: Eureka Server

This section describes how to set up a Eureka server.

5.2.1. How to Include Eureka Server

To include Eureka Server in your project, use the starter with a group ID of `org.springframework.cloud` and an artifact ID of `spring-cloud-starter-netflix-eureka-server`. See the [Spring Cloud Project page](#) for details on setting up your build system with the current Spring Cloud Release Train.



If your project already uses Thymeleaf as its template engine, the Freemarker templates of the Eureka server may not be loaded correctly. In this case it is necessary to configure the template loader manually:

application.yml

```
spring:
  freemarker:
    template-loader-path: classpath:/templates/
    prefer-file-system-access: false
```

5.2.2. How to Run a Eureka Server

The following example shows a minimal Eureka server:

```
@SpringBootApplication
@EnableEurekaServer
public class Application {

    public static void main(String[] args) {
        new SpringApplicationBuilder(Application.class).web(true).run(args);
    }

}
```

The server has a home page with a UI and HTTP API endpoints for the normal Eureka functionality under `/eureka/`.

The following links have some Eureka background reading: [flux capacitor](#) and [google group discussion](#).

Due to Gradle's dependency resolution rules and the lack of a parent bom feature, depending on `spring-cloud-starter-netflix-eureka-server` can cause failures on application startup. To remedy this issue, add the Spring Boot Gradle plugin and import the Spring cloud starter parent bom as follows:

build.gradle



```
buildscript {
    dependencies {
        classpath("org.springframework.boot:spring-boot-gradle-
plugin:{spring-boot-docs-version}")
    }
}

apply plugin: "spring-boot"

dependencyManagement {
    imports {
        mavenBom "org.springframework.cloud:spring-cloud-
dependencies:{spring-cloud-version}"
    }
}
```

5.2.3. High Availability, Zones and Regions

The Eureka server does not have a back end store, but the service instances in the registry all have to send heartbeats to keep their registrations up to date (so this can be done in memory). Clients also have an in-memory cache of Eureka registrations (so they do not have to go to the registry for every request to a service).

By default, every Eureka server is also a Eureka client and requires (at least one) service URL to locate a peer. If you do not provide it, the service runs and works, but it fills your logs with a lot of noise about not being able to register with the peer.

See also [below for details of Ribbon support](#) on the client side for Zones and Regions.

5.2.4. Standalone Mode

The combination of the two caches (client and server) and the heartbeats make a standalone Eureka server fairly resilient to failure, as long as there is some sort of monitor or elastic runtime (such as Cloud Foundry) keeping it alive. In standalone mode, you might prefer to switch off the client side behavior so that it does not keep trying and failing to reach its peers. The following example shows how to switch off the client-side behavior:

application.yml (Standalone Eureka Server)

```
server:
  port: 8761

eureka:
  instance:
    hostname: localhost
  client:
    registerWithEureka: false
    fetchRegistry: false
    serviceUrl:
      defaultZone: http://${eureka.instance.hostname}:${server.port}/eureka/
```

Notice that the `serviceUrl` is pointing to the same host as the local instance.

5.2.5. Peer Awareness

Eureka can be made even more resilient and available by running multiple instances and asking them to register with each other. In fact, this is the default behavior, so all you need to do to make it work is add a valid `serviceUrl` to a peer, as shown in the following example:

application.yml (Two Peer Aware Eureka Servers)

```
---
spring:
  profiles: peer1
eureka:
  instance:
    hostname: peer1
  client:
    serviceUrl:
      defaultZone: https://peer2/eureka/

---
spring:
  profiles: peer2
eureka:
  instance:
    hostname: peer2
  client:
    serviceUrl:
      defaultZone: https://peer1/eureka/
```

In the preceding example, we have a YAML file that can be used to run the same server on two hosts (`peer1` and `peer2`) by running it in different Spring profiles. You could use this configuration to test the peer awareness on a single host (there is not much value in doing that in production) by manipulating `/etc/hosts` to resolve the host names. In fact, the `eureka.instance.hostname` is not needed if you are running on a machine that knows its own hostname (by default, it is looked up by

using `java.net.InetAddress`).

You can add multiple peers to a system, and, as long as they are all connected to each other by at least one edge, they synchronize the registrations amongst themselves. If the peers are physically separated (inside a data center or between multiple data centers), then the system can, in principle, survive “split-brain” type failures. You can add multiple peers to a system, and as long as they are all directly connected to each other, they will synchronize the registrations amongst themselves.

application.yml (Three Peer Aware Eureka Servers)

```
eureka:
  client:
    serviceUrl:
      defaultZone: https://peer1/eureka/,http://peer2/eureka/,http://peer3/eureka/

---
spring:
  profiles: peer1
eureka:
  instance:
    hostname: peer1

---
spring:
  profiles: peer2
eureka:
  instance:
    hostname: peer2

---
spring:
  profiles: peer3
eureka:
  instance:
    hostname: peer3
```

5.2.6. When to Prefer IP Address

In some cases, it is preferable for Eureka to advertise the IP addresses of services rather than the hostname. Set `eureka.instance.preferIpAddress` to `true` and, when the application registers with eureka, it uses its IP address rather than its hostname.



If the hostname cannot be determined by Java, then the IP address is sent to Eureka. Only explicit way of setting the hostname is by setting `eureka.instance.hostname` property. You can set your hostname at the run-time by using an environment variable — for example, `eureka.instance.hostname=${HOST_NAME}`.

5.2.7. Securing The Eureka Server

You can secure your Eureka server simply by adding Spring Security to your server's classpath via `spring-boot-starter-security`. By default when Spring Security is on the classpath it will require that a valid CSRF token be sent with every request to the app. Eureka clients will not generally possess a valid cross site request forgery (CSRF) token you will need to disable this requirement for the `/eureka/**` endpoints. For example:

```
@EnableWebSecurity
class WebSecurityConfig extends WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.csrf().ignoringAntMatchers("/eureka/**");
        super.configure(http);
    }
}
```

For more information on CSRF see the [Spring Security documentation](#).

A demo Eureka Server can be found in the Spring Cloud Samples [repo](#).

5.2.8. JDK 11 Support

The JAXB modules which the Eureka server depends upon were removed in JDK 11. If you intend to use JDK 11 when running a Eureka server you must include these dependencies in your POM or Gradle file.

```
<dependency>
  <groupId>org.glassfish.jaxb</groupId>
  <artifactId>jaxb-runtime</artifactId>
</dependency>
```

5.3. Configuration properties

To see the list of all Spring Cloud Netflix related configuration properties please check [the Appendix page](#).

Chapter 6. Spring Cloud OpenFeign

2020.0.0-M2

This project provides OpenFeign integrations for Spring Boot apps through autoconfiguration and binding to the Spring Environment and other Spring programming model idioms.

6.1. Declarative REST Client: Feign

[Feign](#) is a declarative web service client. It makes writing web service clients easier. To use Feign create an interface and annotate it. It has pluggable annotation support including Feign annotations and JAX-RS annotations. Feign also supports pluggable encoders and decoders. Spring Cloud adds support for Spring MVC annotations and for using the same [HttpMessageConverters](#) used by default in Spring Web. Spring Cloud integrates Eureka, as well as Spring Cloud LoadBalancer to provide a load-balanced http client when using Feign.

6.1.1. How to Include Feign

To include Feign in your project use the starter with group [org.springframework.cloud](#) and artifact id [spring-cloud-starter-openfeign](#). See the [Spring Cloud Project page](#) for details on setting up your build system with the current Spring Cloud Release Train.

Example spring boot app

```
@SpringBootApplication
@EnableFeignClients
public class Application {

    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }

}
```

```
@FeignClient("stores")
public interface StoreClient {
    @RequestMapping(method = RequestMethod.GET, value = "/stores")
    List<Store> getStores();

    @RequestMapping(method = RequestMethod.GET, value = "/stores")
    Page<Store> getStores(Pageable pageable);

    @RequestMapping(method = RequestMethod.POST, value = "/stores/{storeId}", consumes
= "application/json")
    Store update(@PathVariable("storeId") Long storeId, Store store);
}
```

In the `@FeignClient` annotation the String value ("stores" above) is an arbitrary client name, which is used to create a [Spring Cloud LoadBalancer client](#). You can also specify a URL using the `url` attribute (absolute value or just a hostname). The name of the bean in the application context is the fully qualified name of the interface. To specify your own alias value you can use the `qualifier` value of the `@FeignClient` annotation.

The load-balancer client above will want to discover the physical addresses for the "stores" service. If your application is a Eureka client then it will resolve the service in the Eureka service registry. If you don't want to use Eureka, you can simply configure a list of servers in your external configuration using [SimpleDiscoveryClient](#).

6.1.2. Overriding Feign Defaults

A central concept in Spring Cloud's Feign support is that of the named client. Each feign client is part of an ensemble of components that work together to contact a remote server on demand, and the ensemble has a name that you give it as an application developer using the `@FeignClient` annotation. Spring Cloud creates a new ensemble as an `ApplicationContext` on demand for each named client using `FeignClientsConfiguration`. This contains (amongst other things) an `feign.Decoder`, a `feign.Encoder`, and a `feign.Contract`. It is possible to override the name of that ensemble by using the `contextId` attribute of the `@FeignClient` annotation.

Spring Cloud lets you take full control of the feign client by declaring additional configuration (on top of the `FeignClientsConfiguration`) using `@FeignClient`. Example:

```
@FeignClient(name = "stores", configuration = FooConfiguration.class)
public interface StoreClient {
    //..
}
```

In this case the client is composed from the components already in `FeignClientsConfiguration` together with any in `FooConfiguration` (where the latter will override the former).



`FooConfiguration` does not need to be annotated with `@Configuration`. However, if it is, then take care to exclude it from any `@ComponentScan` that would otherwise include this configuration as it will become the default source for `feign.Decoder`, `feign.Encoder`, `feign.Contract`, etc., when specified. This can be avoided by putting it in a separate, non-overlapping package from any `@ComponentScan` or `@SpringBootApplication`, or it can be explicitly excluded in `@ComponentScan`.



The `serviceId` attribute is now deprecated in favor of the `name` attribute.



Using `contextId` attribute of the `@FeignClient` annotation in addition to changing the name of the `ApplicationContext` ensemble, it will override the alias of the client name and it will be used as part of the name of the configuration bean created for that client.



Previously, using the `url` attribute, did not require the `name` attribute. Using `name` is now required.

Placeholders are supported in the `name` and `url` attributes.

```
@FeignClient(name = "${feign.name}", url = "${feign.url}")
public interface StoreClient {
    //..
}
```

Spring Cloud OpenFeign provides the following beans by default for feign (`BeanType` `beanName`: `ClassName`):

- `Decoder` `feignDecoder`: `ResponseEntityDecoder` (which wraps a `SpringDecoder`)
- `Encoder` `feignEncoder`: `SpringEncoder`
- `Logger` `feignLogger`: `Slf4jLogger`
- `Contract` `feignContract`: `SpringMvcContract`
- `Feign.Builder` `feignBuilder`: `HystrixFeign.Builder`
- `Client` `feignClient`: if Spring Cloud LoadBalancer is in the classpath, `FeignBlockingLoadBalancerClient` is used. If none of them is in the classpath, the default feign client is used.

The `OkHttpClient` and `ApacheHttpClient` feign clients can be used by setting `feign.okhttp.enabled` or `feign.httpClient.enabled` to `true`, respectively, and having them on the classpath. You can customize the HTTP client used by providing a bean of either `org.apache.http.impl.client.CloseableHttpClient` when using Apache or `okhttp3.OkHttpClient` when using OK HTTP.

Spring Cloud OpenFeign *does not* provide the following beans by default for feign, but still looks up beans of these types from the application context to create the feign client:

- `Logger.Level`
- `Retryer`
- `ErrorDecoder`
- `Request.Options`
- `Collection<RequestInterceptor>`
- `SetterFactory`
- `QueryMapEncoder`

Creating a bean of one of those type and placing it in a `@FeignClient` configuration (such as `FooConfiguration` above) allows you to override each one of the beans described. Example:

```
@Configuration
public class FooConfiguration {
    @Bean
    public Contract feignContract() {
        return new feign.Contract.Default();
    }

    @Bean
    public BasicAuthRequestInterceptor basicAuthRequestInterceptor() {
        return new BasicAuthRequestInterceptor("user", "password");
    }
}
```

This replaces the `SpringMvcContract` with `feign.Contract.Default` and adds a `RequestInterceptor` to the collection of `RequestInterceptor`.

`@FeignClient` also can be configured using configuration properties.

application.yml

```
feign:
  client:
    config:
      feignName:
        connectTimeout: 5000
        readTimeout: 5000
        loggerLevel: full
        errorDecoder: com.example.SimpleErrorDecoder
        retryer: com.example.SimpleRetryer
        requestInterceptors:
          - com.example.FooRequestInterceptor
          - com.example.BarRequestInterceptor
        decode404: false
        encoder: com.example.SimpleEncoder
        decoder: com.example.SimpleDecoder
        contract: com.example.SimpleContract
```

Default configurations can be specified in the `@EnableFeignClients` attribute `defaultConfiguration` in a similar manner as described above. The difference is that this configuration will apply to *all* feign clients.

If you prefer using configuration properties to configured all `@FeignClient`, you can create configuration properties with `default` feign name.

application.yml

```
feign:
  client:
    config:
      default:
        connectTimeout: 5000
        readTimeout: 5000
        loggerLevel: basic
```

If we create both `@Configuration` bean and configuration properties, configuration properties will win. It will override `@Configuration` values. But if you want to change the priority to `@Configuration`, you can change `feign.client.default-to-properties` to `false`.



If you need to use `ThreadLocal` bound variables in your `RequestInterceptor`'s you will need to either set the thread isolation strategy for Hystrix to `'SEMAPHORE` or disable Hystrix in Feign.

application.yml

```
# To disable Hystrix in Feign
feign:
  hystrix:
    enabled: false

# To set thread isolation to SEMAPHORE
hystrix:
  command:
    default:
      execution:
        isolation:
          strategy: SEMAPHORE
```

If we want to create multiple feign clients with the same name or url so that they would point to the same server but each with a different custom configuration then we have to use `contextId` attribute of the `@FeignClient` in order to avoid name collision of these configuration beans.

```

@FeignClient(contextId = "fooClient", name = "stores", configuration =
FooConfiguration.class)
public interface FooClient {
    //..
}

```

```

@FeignClient(contextId = "barClient", name = "stores", configuration =
BarConfiguration.class)
public interface BarClient {
    //..
}

```

It is also possible to configure FeignClient not to inherit beans from the parent context. You can do this by overriding the `inheritParentConfiguration()` in a `FeignClientConfigurer` bean to return `false`:

```

@Configuration
public class CustomConfiguration{

    @Bean
    public FeignClientConfigurer feignClientConfigurer() {
        return new FeignClientConfigurer() {

            @Override
            public boolean inheritParentConfiguration() {
                return false;
            }
        };
    }
}

```

6.1.3. Creating Feign Clients Manually

In some cases it might be necessary to customize your Feign Clients in a way that is not possible using the methods above. In this case you can create Clients using the [Feign Builder API](#). Below is an example which creates two Feign Clients with the same interface but configures each one with a separate request interceptor.


```

@Import(FeignClientsConfiguration.class)
class FooController {

    private FooClient fooClient;

    private FooClient adminClient;

    @Autowired
    public FooController(Decoder decoder, Encoder encoder, Client client, Contract
contract) {
        this.fooClient = Feign.builder().client(client)
            .encoder(encoder)
            .decoder(decoder)
            .contract(contract)
            .requestInterceptor(new BasicAuthRequestInterceptor("user", "user"))
            .target(FooClient.class, "https://PROD-SVC");

        this.adminClient = Feign.builder().client(client)
            .encoder(encoder)
            .decoder(decoder)
            .contract(contract)
            .requestInterceptor(new BasicAuthRequestInterceptor("admin", "admin"))
            .target(FooClient.class, "https://PROD-SVC");
    }
}

```



In the above example `FeignClientsConfiguration.class` is the default configuration provided by Spring Cloud Netflix.



`PROD-SVC` is the name of the service the Clients will be making requests to.



The Feign `Contract` object defines what annotations and values are valid on interfaces. The autowired `Contract` bean provides supports for SpringMVC annotations, instead of the default Feign native annotations.

You can also use the `Builder` to configure `FeignClient` not to inherit beans from the parent context. You can do this by overriding calling `inheritParentContext(false)` on the `Builder`.

6.1.4. Feign Hystrix Support

If Hystrix is on the classpath and `feign.hystrix.enabled=true`, Feign will wrap all methods with a circuit breaker. Returning a `com.netflix.hystrix.HystrixCommand` is also available. This lets you use reactive patterns (with a call to `.toObservable()` or `.observe()` or asynchronous use (with a call to `.queue()`).

To disable Hystrix support on a per-client basis create a vanilla `Feign.Builder` with the "prototype" scope, e.g.:

```

@Configuration
public class FooConfiguration {
    @Bean
    @Scope("prototype")
    public Feign.Builder feignBuilder() {
        return Feign.builder();
    }
}

```



Prior to the Spring Cloud Dalston release, if Hystrix was on the classpath Feign would have wrapped all methods in a circuit breaker by default. This default behavior was changed in Spring Cloud Dalston in favor for an opt-in approach.

6.1.5. Feign Hystrix Fallbacks

Hystrix supports the notion of a fallback: a default code path that is executed when they circuit is open or there is an error. To enable fallbacks for a given `@FeignClient` set the `fallback` attribute to the class name that implements the fallback. You also need to declare your implementation as a Spring bean.

```

@FeignClient(name = "hello", fallback = HystrixClientFallback.class)
protected interface HystrixClient {
    @RequestMapping(method = RequestMethod.GET, value = "/hello")
    Hello iFailSometimes();
}

static class HystrixClientFallback implements HystrixClient {
    @Override
    public Hello iFailSometimes() {
        return new Hello("fallback");
    }
}

```

If one needs access to the cause that made the fallback trigger, one can use the `fallbackFactory` attribute inside `@FeignClient`.

```

@FeignClient(name = "hello", fallbackFactory = HystrixClientFallbackFactory.class)
protected interface HystrixClient {
    @RequestMapping(method = RequestMethod.GET, value = "/hello")
    Hello iFailSometimes();
}

@Component
static class HystrixClientFallbackFactory implements FallbackFactory<HystrixClient> {
    @Override
    public HystrixClient create(Throwable cause) {
        return new HystrixClient() {
            @Override
            public Hello iFailSometimes() {
                return new Hello("fallback; reason was: " + cause.getMessage());
            }
        };
    }
}

```



There is a limitation with the implementation of fallbacks in Feign and how Hystrix fallbacks work. Fallbacks are currently not supported for methods that return `com.netflix.hystrix.HystrixCommand` and `rx.Observable`.

6.1.6. Feign and @Primary

When using Feign with Hystrix fallbacks, there are multiple beans in the `ApplicationContext` of the same type. This will cause `@Autowired` to not work because there isn't exactly one bean, or one marked as primary. To work around this, Spring Cloud Netflix marks all Feign instances as `@Primary`, so Spring Framework will know which bean to inject. In some cases, this may not be desirable. To turn off this behavior set the `primary` attribute of `@FeignClient` to false.

```

@FeignClient(name = "hello", primary = false)
public interface HelloClient {
    // methods here
}

```

6.1.7. Feign Inheritance Support

Feign supports boilerplate apis via single-inheritance interfaces. This allows grouping common operations into convenient base interfaces.

UserService.java

```
public interface UserService {  
  
    @RequestMapping(method = RequestMethod.GET, value = "/users/{id}")  
    User getUser(@PathVariable("id") long id);  
}
```

UserResource.java

```
@RestController  
public class UserResource implements UserService {  
  
}
```

UserClient.java

```
package project.user;  
  
@FeignClient("users")  
public interface UserClient extends UserService {  
  
}
```



It is generally not advisable to share an interface between a server and a client. It introduces tight coupling, and also actually doesn't work with Spring MVC in its current form (method parameter mapping is not inherited).

6.1.8. Feign request/response compression

You may consider enabling the request or response GZIP compression for your Feign requests. You can do this by enabling one of the properties:

```
feign.compression.request.enabled=true  
feign.compression.response.enabled=true
```

Feign request compression gives you settings similar to what you may set for your web server:

```
feign.compression.request.enabled=true  
feign.compression.request.mime-types=text/xml,application/xml,application/json  
feign.compression.request.min-request-size=2048
```

These properties allow you to be selective about the compressed media types and minimum request threshold length.

For http clients except OkHttpClient, default gzip decoder can be enabled to decode gzip response in

UTF-8 encoding:

```
feign.compression.response.enabled=true
feign.compression.response.useGzipDecoder=true
```

6.1.9. Feign logging

A logger is created for each Feign client created. By default the name of the logger is the full class name of the interface used to create the Feign client. Feign logging only responds to the **DEBUG** level.

application.yml

```
logging.level.project.user.UserClient: DEBUG
```

The **Logger.Level** object that you may configure per client, tells Feign how much to log. Choices are:

- **NONE**, No logging (**DEFAULT**).
- **BASIC**, Log only the request method and URL and the response status code and execution time.
- **HEADERS**, Log the basic information along with request and response headers.
- **FULL**, Log the headers, body, and metadata for both requests and responses.

For example, the following would set the **Logger.Level** to **FULL**:

```
@Configuration
public class FooConfiguration {
    @Bean
    Logger.Level feignLoggerLevel() {
        return Logger.Level.FULL;
    }
}
```

6.1.10. Feign @QueryMap support

The OpenFeign **@QueryMap** annotation provides support for POJOs to be used as GET parameter maps. Unfortunately, the default OpenFeign QueryMap annotation is incompatible with Spring because it lacks a **value** property.

Spring Cloud OpenFeign provides an equivalent **@SpringQueryMap** annotation, which is used to annotate a POJO or Map parameter as a query parameter map.

For example, the **Params** class defines parameters **param1** and **param2**:

```
// Params.java
public class Params {
    private String param1;
    private String param2;

    // [Getters and setters omitted for brevity]
}
```

The following feign client uses the `Params` class by using the `@SpringQueryMap` annotation:

```
@FeignClient("demo")
public interface DemoTemplate {

    @GetMapping(path = "/demo")
    String demoEndpoint(@SpringQueryMap Params params);
}
```

If you need more control over the generated query parameter map, you can implement a custom `QueryMapEncoder` bean.

6.1.11. HATEOAS support

Spring provides some APIs to create REST representations that follow the [HATEOAS](#) principle, [Spring Hateoas](#) and [Spring Data REST](#).

If your project use the `org.springframework.boot:spring-boot-starter-hateoas` starter or the `org.springframework.boot:spring-boot-starter-data-rest` starter, Feign HATEOAS support is enabled by default.

When HATEOAS support is enabled, Feign clients are allowed to serialize and deserialize HATEOAS representation models: [EntityModel](#), [CollectionModel](#) and [PagedModel](#).

```
@FeignClient("demo")
public interface DemoTemplate {

    @GetMapping(path = "/stores")
    CollectionModel<Store> getStores();
}
```

6.1.12. Spring @MatrixVariable Support

Spring Cloud OpenFeign provides support for the Spring `@MatrixVariable` annotation.

If a map is passed as the method argument, the `@MatrixVariable` path segment is created by joining key-value pairs from the map with a `=`.

If a different object is passed, either the `name` provided in the `@MatrixVariable` annotation (if defined)

or the annotated variable name is joined with the provided method argument using `=`.

IMPORTANT

Even though, on the server side, Spring does not require the users to name the path segment placeholder same as the matrix variable name, since it would be too ambiguous on the client side, Spring Cloud OpenFeign requires that you add a path segment placeholder with a name matching either the `name` provided in the `@MatrixVariable` annotation (if defined) or the annotated variable name.

For example:

```
@GetMapping("/objects/links/{matrixVars}")
Map<String, List<String>> getObjects(@MatrixVariable Map<String, List<String>>
matrixVars);
```

Note that both variable name and the path segment placeholder are called `matrixVars`.

```
@FeignClient("demo")
public interface DemoTemplate {

    @GetMapping(path = "/stores")
    CollectionModel<Store> getStores();
}
```

6.1.13. Reactive Support

As the [OpenFeign project](#) does not currently support reactive clients, such as [Spring WebClient](#), neither does Spring Cloud OpenFeign. We will add support for it here as soon as it becomes available in the core project.

Until that is done, we recommend using [feign-reactive](#) for Spring WebClient support.

Early Initialization Errors

Depending on how you are using your Feign clients you may see initialization errors when starting your application. To work around this problem you can use an `ObjectProvider` when autowiring your client.

```
@Autowired
ObjectProvider<TestFeginClient> testFeginClient;
```

6.2. Configuration properties

To see the list of all Sleuth related configuration properties please check [the Appendix page](#).

Chapter 7. Spring Cloud Bus

Spring Cloud Bus links the nodes of a distributed system with a lightweight message broker. This broker can then be used to broadcast state changes (such as configuration changes) or other management instructions. A key idea is that the bus is like a distributed actuator for a Spring Boot application that is scaled out. However, it can also be used as a communication channel between apps. This project provides starters for either an AMQP broker or Kafka as the transport.



Spring Cloud is released under the non-restrictive Apache 2.0 license. If you would like to contribute to this section of the documentation or if you find an error, please find the source code and issue trackers in the project at [github](#).

7.1. Quick Start

Spring Cloud Bus works by adding Spring Boot autconfiguration if it detects itself on the classpath. To enable the bus, add `spring-cloud-starter-bus-amqp` or `spring-cloud-starter-bus-kafka` to your dependency management. Spring Cloud takes care of the rest. Make sure the broker (RabbitMQ or Kafka) is available and configured. When running on localhost, you need not do anything. If you run remotely, use Spring Cloud Connectors or Spring Boot conventions to define the broker credentials, as shown in the following example for Rabbit:

application.yml

```
spring:
  rabbitmq:
    host: mybroker.com
    port: 5672
    username: user
    password: secret
```

The bus currently supports sending messages to all nodes listening or all nodes for a particular service (as defined by Eureka). The `/bus/*` actuator namespace has some HTTP endpoints. Currently, two are implemented. The first, `/bus/env`, sends key/value pairs to update each node's Spring Environment. The second, `/bus/refresh`, reloads each application's configuration, as though they had all been pinged on their `/refresh` endpoint.



The Spring Cloud Bus starters cover Rabbit and Kafka, because those are the two most common implementations. However, Spring Cloud Stream is quite flexible, and the binder works with `spring-cloud-bus`.

7.2. Bus Endpoints

Spring Cloud Bus provides two endpoints, `/actuator/bus-refresh` and `/actuator/bus-env` that correspond to individual actuator endpoints in Spring Cloud Commons, `/actuator/refresh` and `/actuator/env` respectively.

7.2.1. Bus Refresh Endpoint

The `/actuator/bus-refresh` endpoint clears the `RefreshScope` cache and rebinds `@ConfigurationProperties`. See the [Refresh Scope](#) documentation for more information.

To expose the `/actuator/bus-refresh` endpoint, you need to add following configuration to your application:

```
management.endpoints.web.exposure.include=bus-refresh
```

7.2.2. Bus Env Endpoint

The `/actuator/bus-env` endpoint updates each instances environment with the specified key/value pair across multiple instances.

To expose the `/actuator/bus-env` endpoint, you need to add following configuration to your application:

```
management.endpoints.web.exposure.include=bus-env
```

The `/actuator/bus-env` endpoint accepts `POST` requests with the following shape:

```
{
  "name": "key1",
  "value": "value1"
}
```

7.3. Addressing an Instance

Each instance of the application has a service ID, whose value can be set with `spring.cloud.bus.id` and whose value is expected to be a colon-separated list of identifiers, in order from least specific to most specific. The default value is constructed from the environment as a combination of the `spring.application.name` and `server.port` (or `spring.application.index`, if set). The default value of the ID is constructed in the form of `app:index:id`, where:

- `app` is the `vcap.application.name`, if it exists, or `spring.application.name`
- `index` is the `vcap.application.instance_index`, if it exists, `spring.application.index`, `local.server.port`, `server.port`, or `0` (in that order).
- `id` is the `vcap.application.instance_id`, if it exists, or a random value.

The HTTP endpoints accept a “destination” path parameter, such as `/bus-refresh/customers:9000`, where `destination` is a service ID. If the ID is owned by an instance on the bus, it processes the message, and all other instances ignore it.

7.4. Addressing All Instances of a Service

The “destination” parameter is used in a Spring `PathMatcher` (with the path separator as a colon — `:`) to determine if an instance processes the message. Using the example from earlier, `/bus-env/customers:**` targets all instances of the “customers” service regardless of the rest of the service ID.

7.5. Service ID Must Be Unique

The bus tries twice to eliminate processing an event — once from the original `ApplicationEvent` and once from the queue. To do so, it checks the sending service ID against the current service ID. If multiple instances of a service have the same ID, events are not processed. When running on a local machine, each service is on a different port, and that port is part of the ID. Cloud Foundry supplies an index to differentiate. To ensure that the ID is unique outside Cloud Foundry, set `spring.application.index` to something unique for each instance of a service.

7.6. Customizing the Message Broker

Spring Cloud Bus uses `Spring Cloud Stream` to broadcast the messages. So, to get messages to flow, you need only include the binder implementation of your choice in the classpath. There are convenient starters for the bus with AMQP (RabbitMQ) and Kafka (`spring-cloud-starter-bus-[amqp|kafka]`). Generally speaking, Spring Cloud Stream relies on Spring Boot autoconfiguration conventions for configuring middleware. For instance, the AMQP broker address can be changed with `spring.rabbitmq.*` configuration properties. Spring Cloud Bus has a handful of native configuration properties in `spring.cloud.bus.*` (for example, `spring.cloud.bus.destination` is the name of the topic to use as the external middleware). Normally, the defaults suffice.

To learn more about how to customize the message broker settings, consult the Spring Cloud Stream documentation.

7.7. Tracing Bus Events

Bus events (subclasses of `RemoteApplicationEvent`) can be traced by setting `spring.cloud.bus.trace.enabled=true`. If you do so, the Spring Boot `TraceRepository` (if it is present) shows each event sent and all the acks from each service instance. The following example comes from the `/trace` endpoint:

```

{
  "timestamp": "2015-11-26T10:24:44.411+0000",
  "info": {
    "signal": "spring.cloud.bus.ack",
    "type": "RefreshRemoteApplicationEvent",
    "id": "c4d374b7-58ea-4928-a312-31984def293b",
    "origin": "stores:8081",
    "destination": "*:**"
  }
},
{
  "timestamp": "2015-11-26T10:24:41.864+0000",
  "info": {
    "signal": "spring.cloud.bus.sent",
    "type": "RefreshRemoteApplicationEvent",
    "id": "c4d374b7-58ea-4928-a312-31984def293b",
    "origin": "customers:9000",
    "destination": "*:**"
  }
},
{
  "timestamp": "2015-11-26T10:24:41.862+0000",
  "info": {
    "signal": "spring.cloud.bus.ack",
    "type": "RefreshRemoteApplicationEvent",
    "id": "c4d374b7-58ea-4928-a312-31984def293b",
    "origin": "customers:9000",
    "destination": "*:**"
  }
}
}

```

The preceding trace shows that a `RefreshRemoteApplicationEvent` was sent from `customers:9000`, broadcast to all services, and received (acked) by `customers:9000` and `stores:8081`.

To handle the ack signals yourself, you could add an `@EventListener` for the `AckRemoteApplicationEvent` and `SentApplicationEvent` types to your app (and enable tracing). Alternatively, you could tap into the `TraceRepository` and mine the data from there.



Any Bus application can trace acks. However, sometimes, it is useful to do this in a central service that can do more complex queries on the data or forward it to a specialized tracing service.

7.8. Broadcasting Your Own Events

The Bus can carry any event of type `RemoteApplicationEvent`. The default transport is JSON, and the deserializer needs to know which types are going to be used ahead of time. To register a new type, you must put it in a subpackage of `org.springframework.cloud.bus.event`.

To customise the event name, you can use `@JsonTypeName` on your custom class or rely on the default strategy, which is to use the simple name of the class.



Both the producer and the consumer need access to the class definition.

7.8.1. Registering events in custom packages

If you cannot or do not want to use a subpackage of `org.springframework.cloud.bus.event` for your custom events, you must specify which packages to scan for events of type `RemoteApplicationEvent` by using the `@RemoteApplicationEventScan` annotation. Packages specified with `@RemoteApplicationEventScan` include subpackages.

For example, consider the following custom event, called `MyEvent`:

```
package com.acme;

public class MyEvent extends RemoteApplicationEvent {
    ...
}
```

You can register that event with the deserializer in the following way:

```
package com.acme;

@Configuration
@RemoteApplicationEventScan
public class BusConfiguration {
    ...
}
```

Without specifying a value, the package of the class where `@RemoteApplicationEventScan` is used is registered. In this example, `com.acme` is registered by using the package of `BusConfiguration`.

You can also explicitly specify the packages to scan by using the `value`, `basePackages` or `basePackageClasses` properties on `@RemoteApplicationEventScan`, as shown in the following example:

```
package com.acme;

@Configuration
//@RemoteApplicationEventScan({"com.acme", "foo.bar"})
//@RemoteApplicationEventScan(basePackages = {"com.acme", "foo.bar", "fizz.buzz"})
@RemoteApplicationEventScan(basePackageClasses = BusConfiguration.class)
public class BusConfiguration {
    ...
}
```

All of the preceding examples of `@RemoteApplicationEventScan` are equivalent, in that the `com.acme`

package is registered by explicitly specifying the packages on `@RemoteApplicationEventScan`.



You can specify multiple base packages to scan.

7.9. Configuration properties

To see the list of all Bus related configuration properties please check [the Appendix page](#).

Chapter 8. Spring Cloud Sleuth

Adrian Cole, Spencer Gibb, Marcin Grzeszczak, Dave Syer, Jay Bryant

2020.0.0-M2

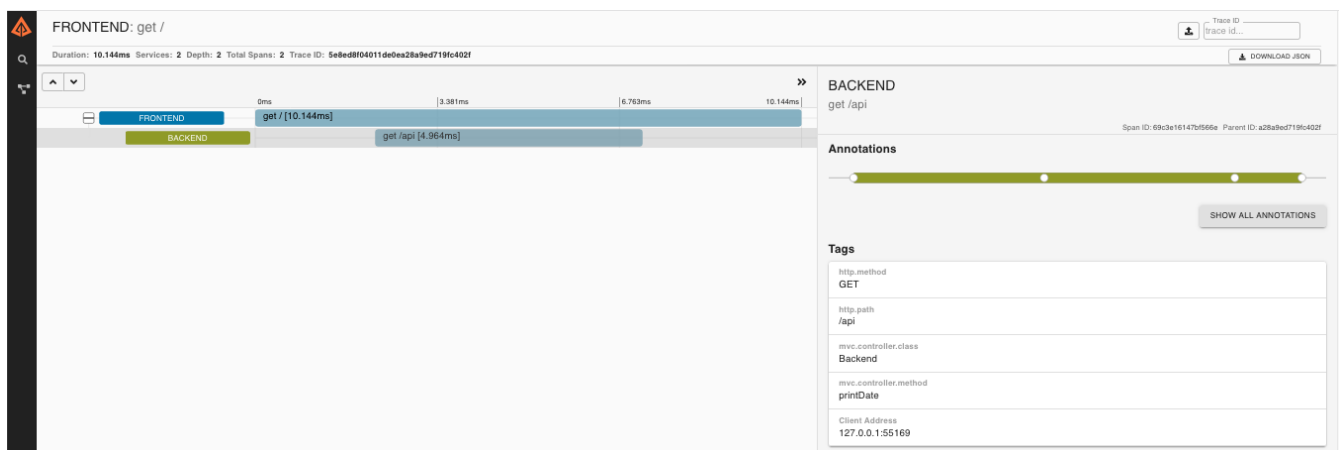
8.1. Overview

Spring Cloud Sleuth provides Spring Boot auto-configuration for distributed tracing. Underneath, Spring Cloud Sleuth is a layer over a Tracer library named [Brave](#).

Sleuth configures everything you need to get started. This includes where trace data (spans) are reported to, how many traces to keep (sampling), if remote fields (baggage) are sent, and which libraries are traced.

We maintain an [example app](#) where two Spring Boot services collaborate on an HTTP request. Sleuth configures these apps, so that timing of these requests are recorded into [Zipkin](#), a distributed tracing system. Tracing UIs visualize latency, such as time in one service vs waiting for other services.

Here's an example of what it looks like:



The [source repository](#) of this example includes demonstrations of many things, including WebFlux and messaging. Most features require only a property or dependency change to work. These snippets showcase the value of Spring Cloud Sleuth: Through auto-configuration, Sleuth make getting started with distributed tracing easy!

To keep things simple, the same example is used throughout documentation using basic HTTP communication.

8.2. Features

Sleuth sets up instrumentation not only to track timing, but also to catch errors so that they can be analyzed or correlated with logs. This works the same way regardless of if the error came from a common instrumented library, such as [RestTemplate](#), or your own code annotated with [@NewSpan](#) or similar.

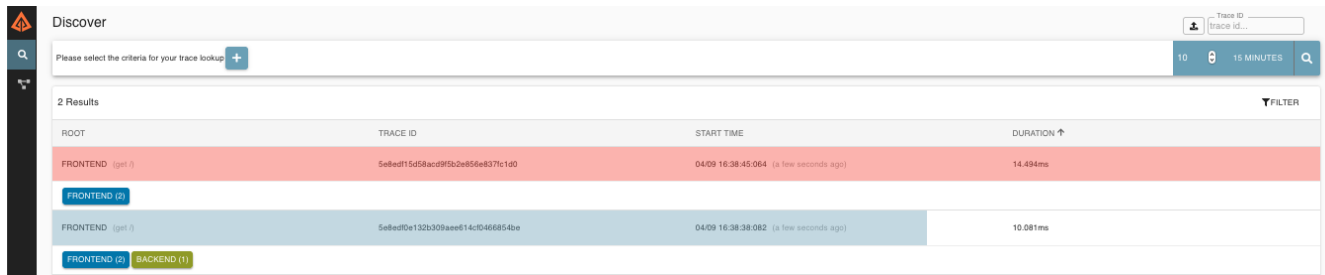
Below, we'll use the word Zipkin to describe the tracing system, and include Zipkin screenshots. However, most services accepting [Zipkin format](#) have similar base features. Sleuth can also be configured to send data in other formats, something detailed later.

8.2.1. Contextualizing errors

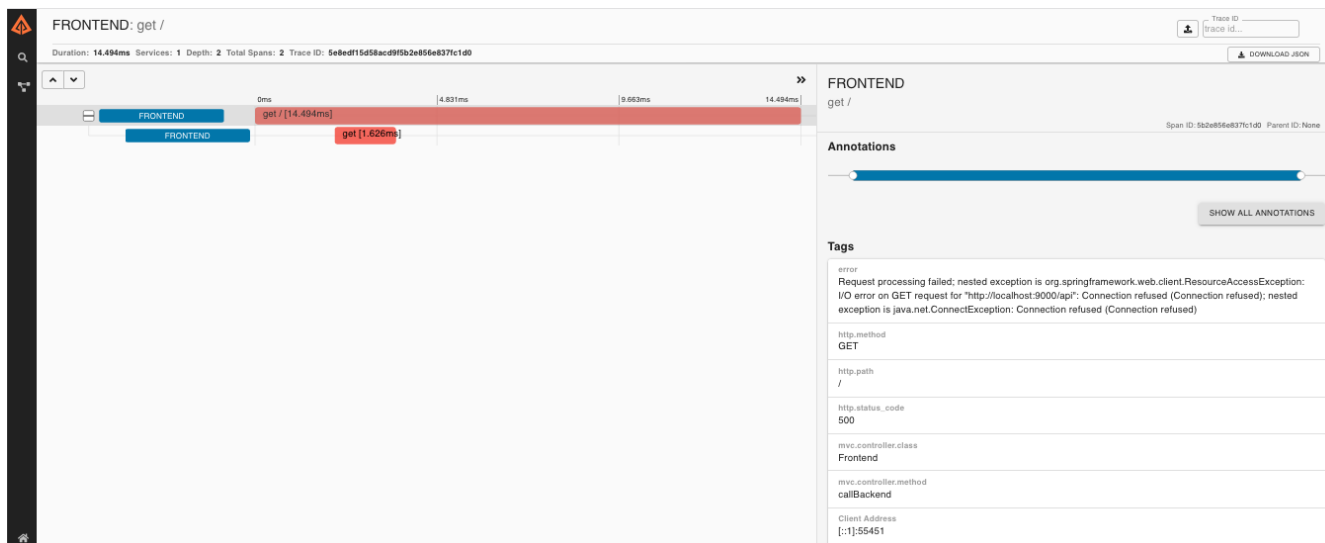
Without distributed tracing, it can be difficult to understand the impact of a an exception. For example, it can be hard to know if a specific request caused the caller to fail or not.

Zipkin reduces time in triage by contextualizing errors and delays.

Requests colored red in the search screen failed:



If you then click on one of the traces, you can understand if the failure happened before the request hit another service or not:



For example, the above error happened in the "backend" service, and caused the "frontend" service to fail.

8.2.2. Log correlation

Sleuth configures the logging context with variables including the service name (`spring.zipkin.service.name`) and the trace ID (`traceId`). These help you connect logs with distributed traces and allow you choice in what tools you use to troubleshoot your services.

Once you find any log with an error, you can look for the trace ID in the message. Paste that into Zipkin to visualize the entire trace, regardless of how many services the first request ended up hitting.

```
backend.log: 2020-04-09 17:45:40.516 ERROR
[backend,5e8eeec48b08e26882aba313eb08f0a4,dcc1df555b5777b3,true] 97203 --- [nio-9000-
exec-1] o.s.c.s.i.web.ExceptionLoggingFilter      : Uncaught exception thrown
frontend.log:2020-04-09 17:45:40.574 ERROR
[frontend,5e8eeec48b08e26882aba313eb08f0a4,82aba313eb08f0a4,true] 97192 --- [nio-8081-
exec-2] o.s.c.s.i.web.ExceptionLoggingFilter      : Uncaught exception thrown
```

Above, you'll notice the trace ID is `5e8eeec48b08e26882aba313eb08f0a4`, for example. This log configuration was automatically setup by Sleuth.

8.2.3. Service Dependency Graph

When you consider distributed tracing tracks requests, it makes sense that trace data can paint a picture of your architecture.

Zipkin includes a tool to build service dependency diagrams from traces, including the count of calls and how many errors exist.

The example application will make a simple diagram like this, but your real environment diagram may be more complex. [image::https://raw.githubusercontent.com/spring-cloud/spring-cloud-sleuth/master/docs/src/main/asciidoc/images/zipkin-dependencies.png](https://raw.githubusercontent.com/spring-cloud/spring-cloud-sleuth/master/docs/src/main/asciidoc/images/zipkin-dependencies.png)[Zipkin Dependencies]

Note: Production environments will generate a lot of data. You will likely need to run a separate service to aggregate the dependency graph. You can learn more [here](#).

8.2.4. Request scoped properties (Baggage)

Distributed tracing works by propagating fields inside and across services that connect the trace together: `traceId` and `spanId` notably. The context that holds these fields can optionally push other fields that need to be consistent regardless of many services are touched. The simple name for these extra fields is "Baggage".

Sleuth allows you to define which baggage are permitted to exist in the trace context, including what header names are used.

The following example shows setting baggage values:

```
Span initialSpan = this.tracer.nextSpan().name("span").start();
BUSINESS_PROCESS.updateValue(initialSpan.context(), "ALM");
COUNTRY_CODE.updateValue(initialSpan.context(), "FO");
```



There is currently no limitation of the count or size of baggage items. Keep in mind that too many can decrease system throughput or increase RPC latency. In extreme cases, too much baggage can crash the application, due to exceeding transport-level message or header capacity.

Baggage versus Tags

Like trace IDs, Baggage is attached to messages or requests, usually as headers. Tags are key value pairs sent in a Span to Zipkin. Baggage values are not added spans by default, which means you can't search based on Baggage unless you opt-in.

To make baggage also tags, use the property `spring.sleuth.baggage.tag-fields` like so:

```
spring:
  sleuth:
    baggage:
      foo: bar
      remoteFields:
        - country-code
        - x-vcap-request-id
      tagFields:
        - country-code
```

8.3. Adding Sleuth to your Project

This section addresses how to add Sleuth to your project with either Maven or Gradle.



To ensure that your application name is properly displayed in Zipkin, set the `spring.application.name` property in `bootstrap.yml`.

8.3.1. Sleuth with Zipkin via HTTP

If you want both Sleuth and Zipkin, add the `spring-cloud-starter-zipkin` dependency.

The following example shows how to do so for Maven:

Maven

```
<dependencyManagement> ①
  <dependencies>
    <dependency>
      <groupId>org.springframework.cloud</groupId>
      <artifactId>spring-cloud-dependencies</artifactId>
      <version>${release.train.version}</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
  </dependencies>
</dependencyManagement>

<dependency> ②
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-zipkin</artifactId>
</dependency>
```

- ① We recommend that you add the dependency management through the Spring BOM so that you need not manage versions yourself.
- ② Add the dependency to `spring-cloud-starter-zipkin`.

The following example shows how to do so for Gradle:

Gradle

```
dependencyManagement { ①
  imports {
    mavenBom "org.springframework.cloud:spring-cloud-
dependencies:${releaseTrainVersion}"
  }
}

dependencies { ②
  compile "org.springframework.cloud:spring-cloud-starter-zipkin"
}
```

- ① We recommend that you add the dependency management through the Spring BOM so that you need not manage versions yourself.
- ② Add the dependency to `spring-cloud-starter-zipkin`.

8.3.2. Sleuth with Zipkin over RabbitMQ or Kafka

If you want to use RabbitMQ or Kafka instead of HTTP, add the `spring-rabbit` or `spring-kafka` dependency. The default destination name is `zipkin`.

If using Kafka, you must set the property `spring.zipkin.sender.type` property accordingly:

```
spring.zipkin.sender.type: kafka
```



`spring-cloud-sleuth-stream` is deprecated and incompatible with these destinations.

If you want Sleuth over RabbitMQ, add the `spring-cloud-starter-zipkin` and `spring-rabbit` dependencies.

The following example shows how to do so for Gradle:

Maven

```
<dependencyManagement> ①
  <dependencies>
    <dependency>
      <groupId>org.springframework.cloud</groupId>
      <artifactId>spring-cloud-dependencies</artifactId>
      <version>${release.train.version}</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
  </dependencies>
</dependencyManagement>

<dependency> ②
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-zipkin</artifactId>
</dependency>
<dependency> ③
  <groupId>org.springframework.amqp</groupId>
  <artifactId>spring-rabbit</artifactId>
</dependency>
```

- ① We recommend that you add the dependency management through the Spring BOM so that you need not manage versions yourself.
- ② Add the dependency to `spring-cloud-starter-zipkin`. That way, all nested dependencies get downloaded.
- ③ To automatically configure RabbitMQ, add the `spring-rabbit` dependency.

```
dependencyManagement { ❶
    imports {
        mavenBom "org.springframework.cloud:spring-cloud-
dependencies:${releaseTrainVersion}"
    }
}

dependencies {
    compile "org.springframework.cloud:spring-cloud-starter-zipkin" ❷
    compile "org.springframework.amqp:spring-rabbit" ❸
}
```

- ❶ We recommend that you add the dependency management through the Spring BOM so that you need not manage versions yourself.
- ❷ Add the dependency to `spring-cloud-starter-zipkin`. That way, all nested dependencies get downloaded.
- ❸ To automatically configure RabbitMQ, add the `spring-rabbit` dependency.

8.3.3. Overriding the auto-configuration of Zipkin

Spring Cloud Sleuth supports sending traces to multiple tracing systems as of version 2.1.0. In order to get this to work, every tracing system needs to have a `Reporter` and `Sender`. If you want to override the provided beans you need to give them a specific name. To do this you can use respectively `ZipkinAutoConfiguration.REPORTER_BEAN_NAME` and `ZipkinAutoConfiguration.SENDER_BEAN_NAME`.

```

@Configuration
protected static class MyConfig {

    @Bean(ZipkinAutoConfiguration.REPORTER_BEAN_NAME)
    Reporter<zipkin2.Span> myReporter() {
        return AsyncReporter.create(mySender());
    }

    @Bean(ZipkinAutoConfiguration.SENDER_BEAN_NAME)
    MySender mySender() {
        return new MySender();
    }

    static class MySender extends Sender {

        private boolean spanSent = false;

        boolean isSpanSent() {
            return this.spanSent;
        }

        @Override
        public Encoding encoding() {
            return Encoding.JSON;
        }

        @Override
        public int messageMaxBytes() {
            return Integer.MAX_VALUE;
        }

        @Override
        public int messageSizeInBytes(List<byte[]> encodedSpans) {
            return encoding().listSizeInBytes(encodedSpans);
        }

        @Override
        public Call<Void> sendSpans(List<byte[]> encodedSpans) {
            this.spanSent = true;
            return Call.create(null);
        }

    }

}

```

8.3.4. Only Sleuth (log correlation)

If you want to use only Spring Cloud Sleuth without the Zipkin integration, add the `spring-cloud-`

`starter-sleuth` module to your project.

The following example shows how to add Sleuth with Maven:

Maven

```
<dependencyManagement> ①
  <dependencies>
    <dependency>
      <groupId>org.springframework.cloud</groupId>
      <artifactId>spring-cloud-dependencies</artifactId>
      <version>${release.train.version}</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
  </dependencies>
</dependencyManagement>

<dependency> ②
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-sleuth</artifactId>
</dependency>
```

- ① We recommend that you add the dependency management through the Spring BOM so that you need not manage versions yourself.
- ② Add the dependency to `spring-cloud-starter-sleuth`.

The following example shows how to add Sleuth with Gradle:

Gradle

```
dependencyManagement { ①
  imports {
    mavenBom "org.springframework.cloud:spring-cloud-
dependencies:${releaseTrainVersion}"
  }
}

dependencies { ②
  compile "org.springframework.cloud:spring-cloud-starter-sleuth"
}
```

- ① We recommend that you add the dependency management through the Spring BOM so that you need not manage versions yourself.
- ② Add the dependency to `spring-cloud-starter-sleuth`.

8.4. How Sleuth works

Spring Cloud Sleuth is a layer over [Brave](#).

Brave is a distributed tracing instrumentation library. Brave typically intercepts production requests to gather timing data, correlate and propagate trace contexts.

Trace data, also called spans, are typically reported to [Zipkin](#). Zipkin is an Open Source tracing system, which includes a UI and various collectors, such as HTTP and messaging.

Many Open Source and commercial products accept [Zipkin format](#). Some options are documented [here](#), but many are not. If you cannot use Zipkin and your product isn't listed, clarify with your support representative and have them update that page. In many cases, products already support Zipkin format, they just don't document it.

Traces connect from service to service using header propagation. The default format is [B3](#). Similar to data formats, you can configure alternate header formats also, provided trace and span IDs are compatible with B3. Most notably, this means the trace ID and span IDs are lower-case hex, not UUIDs. Besides trace identifiers, other properties (Baggage) can also be passed along with the request. Remote Baggage must be predefined, but is flexible otherwise.

Sleuth configures everything you need to get started with tracing. Sleuth configures where trace data (spans) are reported to, how many traces to keep (sampling), if remote fields (baggage) are sent, and which libraries are traced. Sleuth also adds annotation based tracing features and some instrumentation not available otherwise, such as Reactor. If cannot find the configuration you are looking for in the documentation, ask [Gitter](#) before assuming something cannot be done.

8.4.1. Brave Basics

Most instrumentation work is done for you by default. Sleuth provides beans to allow you to change what's traced, and it even provides annotations to avoid using tracing libraries! All of this is explained later in this document.

That said, you might want to know more about how things work underneath. Here are some pointers.

Here are the most core types you might use: * [SpanCustomizer](#) - to change the span currently in progress * [Tracer](#) - to get a start new spans ad-hoc

Here are the most relevant links from the OpenZipkin Brave project: * [\[Brave's core library\]\(github.com/openzipkin/brave/tree/master/brave\)](#) * [\[Baggage \(propagated fields\)\]\(github.com/openzipkin/brave/tree/master/brave#baggage\)](#) * [\[HTTP tracing\]\(github.com/openzipkin/brave/tree/master/instrumentation/http\)](#)

8.5. Sampling

Sampling only applies to tracing backends, such as Zipkin. Trace IDs appear in logs regardless of sample rate. Sampling is a way to prevent overloading the system, by consistently tracing some, but not all requests.

The default rate of 10 traces per second is controlled by the `spring.sleuth.sampler.rate` property and applies when we know Sleuth is used for reasons besides logging. Use a rate above 100 traces per second with extreme caution as it can overload your tracing system.

The sampler can be set by Java Config also, as shown in the following example:

```
@Bean
public Sampler defaultSampler() {
    return Sampler.ALWAYS_SAMPLE;
}
```



You can set the HTTP header `b3` to `1`, or, when doing messaging, you can set the `spanFlags` header to `1`. Doing so forces the current request to be sampled regardless of configuration.

8.6. Baggage

Baggage are fields that are propagated with the trace, optionally out of process. You can use properties to define fields that have no special configuration such as name mapping:

- `spring.sleuth.remote-fields` is a list of header names to accept and propagate to remote services.
- `spring.sleuth.local-fields` is a list of names to propagate locally

No prefixing applies with these keys. What you set is literally what is used.

A name set in either of these properties will result in a `BaggageField` of the same name.

In order to automatically set the baggage values to Slf4j's MDC, you have to set the `spring.sleuth.baggage.correlation-fields` property with a list of whitelisted local or remote keys. E.g. `spring.sleuth.baggage.correlation-fields=country-code` will set the value of the `country-code` baggage into MDC.



Remember that adding entries to MDC can drastically decrease the performance of your application!

If you want to add the baggage entries as tags, to make it possible to search for spans via the baggage entries, you can set the value of `spring.sleuth.baggage.tag-fields` with a list of whitelisted baggage keys. To disable the feature you have to pass the `spring.sleuth.propagation.tag.enabled=false` property.

8.6.1. Java configuration

If you need to do anything more advanced than above, do not define properties and instead use a `@Bean` config for the baggage fields you use. * `BaggagePropagationCustomizer` sets up baggage fields * Add a `SingleBaggageField` to control header names for a `BaggageField`. * `CorrelationScopeCustomizer` sets up MDC fields * Add a `SingleCorrelationField` to change the MDC name of a `BaggageField` or if updates flush.

8.7. Instrumentation

Spring Cloud Sleuth automatically instruments all your Spring applications, so you should not have to do anything to activate it. The instrumentation is added by using a variety of technologies according to the stack that is available. For example, for a servlet web application, we use a `Filter`, and, for Spring Integration, we use `ChannelInterceptors`.

You can customize the keys used in span tags. To limit the volume of span data, an HTTP request is, by default, tagged only with a handful of metadata, such as the status code, the host, and the URL. You can add request headers by configuring `spring.sleuth.keys.http.headers` (a list of header names).



Tags are collected and exported only if there is a `Sampler` that allows it. By default, there is no such `Sampler`, to ensure that there is no danger of accidentally collecting too much data without configuring something).

8.8. Span lifecycle

You can do the following operations on the Span by means of `brave.Tracer`:

- **start**: When you start a span, its name is assigned and the start timestamp is recorded.
- **close**: The span gets finished (the end time of the span is recorded) and, if the span is sampled, it is eligible for collection (for example, to Zipkin).
- **continue**: A new instance of span is created. It is a copy of the one that it continues.
- **detach**: The span does not get stopped or closed. It only gets removed from the current thread.
- **create with explicit parent**: You can create a new span and set an explicit parent for it.



Spring Cloud Sleuth creates an instance of `Tracer` for you. In order to use it, you can autowire it.

8.8.1. Creating and finishing spans

You can manually create spans by using the `Tracer`, as shown in the following example:

```
// Start a span. If there was a span present in this thread it will become
// the `newSpan`'s parent.
Span newSpan = this.tracer.nextSpan().name("calculateTax");
try (Tracer.SpanInScope ws = this.tracer.withSpanInScope(newSpan.start())) {
    // ...
    // You can tag a span
    newSpan.tag("taxValue", taxValue);
    // ...
    // You can log an event on a span
    newSpan.annotate("taxCalculated");
}
finally {
    // Once done remember to finish the span. This will allow collecting
    // the span to send it to Zipkin
    newSpan.finish();
}
```

In the preceding example, we could see how to create a new instance of the span. If there is already a span in this thread, it becomes the parent of the new span.



Always clean after you create a span. Also, always finish any span that you want to send to Zipkin.



If your span contains a name greater than 50 chars, that name is truncated to 50 chars. Your names have to be explicit and concrete. Big names lead to latency issues and sometimes even exceptions.

8.8.2. Continuing Spans

Sometimes, you do not want to create a new span but you want to continue one. An example of such a situation might be as follows:

- **AOP:** If there was already a span created before an aspect was reached, you might not want to create a new span.

To continue a span, you can use `brave.Tracer`, as shown in the following example:

```

// let's assume that we're in a thread Y and we've received
// the `initialSpan` from thread X
Span continuedSpan = this.tracer.toSpan(newSpan.context());
try {
    // ...
    // You can tag a span
    continuedSpan.tag("taxValue", taxValue);
    // ...
    // You can log an event on a span
    continuedSpan.annotate("taxCalculated");
}
finally {
    // Once done remember to flush the span. That means that
    // it will get reported but the span itself is not yet finished
    continuedSpan.flush();
}

```

8.8.3. Creating a Span with an explicit Parent

You might want to start a new span and provide an explicit parent of that span. Assume that the parent of a span is in one thread and you want to start a new span in another thread. In Brave, whenever you call `nextSpan()`, it creates a span in reference to the span that is currently in scope. You can put the span in scope and then call `nextSpan()`, as shown in the following example:

```

// let's assume that we're in a thread Y and we've received
// the `initialSpan` from thread X. `initialSpan` will be the parent
// of the `newSpan`
Span newSpan = null;
try (Tracer.SpanInScope ws = this.tracer.withSpanInScope(initialSpan)) {
    newSpan = this.tracer.nextSpan().name("calculateCommission");
    // ...
    // You can tag a span
    newSpan.tag("commissionValue", commissionValue);
    // ...
    // You can log an event on a span
    newSpan.annotate("commissionCalculated");
}
finally {
    // Once done remember to finish the span. This will allow collecting
    // the span to send it to Zipkin. The tags and events set on the
    // newSpan will not be present on the parent
    if (newSpan != null) {
        newSpan.finish();
    }
}
}

```



After creating such a span, you must finish it. Otherwise it is not reported (for example, to Zipkin).

8.9. Naming spans

Picking a span name is not a trivial task. A span name should depict an operation name. The name should be low cardinality, so it should not include identifiers.

Since there is a lot of instrumentation going on, some span names are artificial:

- `controller-method-name` when received by a Controller with a method name of `controllerMethodName`
- `async` for asynchronous operations done with wrapped `Callable` and `Runnable` interfaces.
- Methods annotated with `@Scheduled` return the simple name of the class.

Fortunately, for asynchronous processing, you can provide explicit naming.

8.9.1. `@SpanName` Annotation

You can name the span explicitly by using the `@SpanName` annotation, as shown in the following example:

```
@SpanName("calculateTax")
class TaxCountingRunnable implements Runnable {

    @Override
    public void run() {
        // perform logic
    }

}
```

In this case, when processed in the following manner, the span is named `calculateTax`:

```
Runnable runnable = new TraceRunnable(this.tracing, spanNamer,
    new TaxCountingRunnable());
Future<?> future = executorService.submit(runnable);
// ... some additional logic ...
future.get();
```

8.9.2. `toString()` method

It is pretty rare to create separate classes for `Runnable` or `Callable`. Typically, one creates an anonymous instance of those classes. You cannot annotate such classes. To overcome that limitation, if there is no `@SpanName` annotation present, we check whether the class has a custom implementation of the `toString()` method.

Running such code leads to creating a span named `calculateTax`, as shown in the following example:

```
Runnable runnable = new TraceRunnable(this.tracing, spanNamer, new Runnable() {
    @Override
    public void run() {
        // perform logic
    }

    @Override
    public String toString() {
        return "calculateTax";
    }
});
Future<?> future = executorService.submit(runnable);
// ... some additional logic ...
future.get();
```

8.10. Managing Spans with Annotations

You can manage spans with a variety of annotations.

8.10.1. Rationale

There are a number of good reasons to manage spans with annotations, including:

- API-agnostic means to collaborate with a span. Use of annotations lets users add to a span with no library dependency on a span api. Doing so lets Sleuth change its core API to create less impact to user code.
- Reduced surface area for basic span operations. Without this feature, you must use the span api, which has lifecycle commands that could be used incorrectly. By only exposing scope, tag, and log functionality, you can collaborate without accidentally breaking span lifecycle.
- Collaboration with runtime generated code. With libraries such as Spring Data and Feign, the implementations of interfaces are generated at runtime. Consequently, span wrapping of objects was tedious. Now you can provide annotations over interfaces and the arguments of those interfaces.

8.10.2. Creating New Spans

If you do not want to create local spans manually, you can use the `@NewSpan` annotation. Also, we provide the `@SpanTag` annotation to add tags in an automated fashion.

Now we can consider some examples of usage.

```
@NewSpan
void testMethod();
```

Annotating the method without any parameter leads to creating a new span whose name equals the annotated method name.

```
@NewSpan("customNameOnTestMethod4")
void testMethod4();
```

If you provide the value in the annotation (either directly or by setting the `name` parameter), the created span has the provided value as the name.

```
// method declaration
@NewSpan(name = "customNameOnTestMethod5")
void testMethod5(@SpanTag("testTag") String param);

// and method execution
this.testBean.testMethod5("test");
```

You can combine both the name and a tag. Let's focus on the latter. In this case, the value of the annotated method's parameter runtime value becomes the value of the tag. In our sample, the tag key is `testTag`, and the tag value is `test`.

```
@NewSpan(name = "customNameOnTestMethod3")
@Override
public void testMethod3() {
}
```

You can place the `@NewSpan` annotation on both the class and an interface. If you override the interface's method and provide a different value for the `@NewSpan` annotation, the most concrete one wins (in this case `customNameOnTestMethod3` is set).

8.10.3. Continuing Spans

If you want to add tags and annotations to an existing span, you can use the `@ContinueSpan` annotation, as shown in the following example:

```
// method declaration
@ContinueSpan(log = "testMethod11")
void testMethod11(@SpanTag("testTag11") String param);

// method execution
this.testBean.testMethod11("test");
this.testBean.testMethod13();
```

(Note that, in contrast with the `@NewSpan` annotation, you can also add logs with the `log` parameter.)

That way, the span gets continued and:

- Log entries named `testMethod11.before` and `testMethod11.after` are created.
- If an exception is thrown, a log entry named `testMethod11.afterFailure` is also created.
- A tag with a key of `testTag11` and a value of `test` is created.

8.10.4. Advanced Tag Setting

There are 3 different ways to add tags to a span. All of them are controlled by the `SpanTag` annotation. The precedence is as follows:

1. Try with a bean of `TagValueResolver` type and a provided name.
2. If the bean name has not been provided, try to evaluate an expression. We search for a `TagValueExpressionResolver` bean. The default implementation uses SPEL expression resolution. **IMPORTANT** You can only reference properties from the SPEL expression. Method execution is not allowed due to security constraints.
3. If we do not find any expression to evaluate, return the `toString()` value of the parameter.

Custom extractor

The value of the tag for the following method is computed by an implementation of `TagValueResolver` interface. Its class name has to be passed as the value of the `resolver` attribute.

Consider the following annotated method:

```
@NewSpan
public void getAnnotationForTagValueResolver(
    @SpanTag(key = "test", resolver = TagValueResolver.class) String test) {
}
```

Now further consider the following `TagValueResolver` bean implementation:

```
@Bean(name = "myCustomTagValueResolver")
public TagValueResolver tagValueResolver() {
    return parameter -> "Value from myCustomTagValueResolver";
}
```

The two preceding examples lead to setting a tag value equal to `Value from myCustomTagValueResolver`.

Resolving Expressions for a Value

Consider the following annotated method:

```
@NewSpan
public void getAnnotationForTagValueExpression(@SpanTag(key = "test",
    expression = "'hello' + ' characters'") String test) {
}
```

No custom implementation of a `TagValueExpressionResolver` leads to evaluation of the SPEL expression, and a tag with a value of `4 characters` is set on the span. If you want to use some other expression resolution mechanism, you can create your own implementation of the bean.

Using the `toString()` method

Consider the following annotated method:

```
@NewSpan
public void getAnnotationForArgumentToString(@SpanTag("test") Long param) {
}
```

Running the preceding method with a value of `15` leads to setting a tag with a String value of `"15"`.

8.11. Customizations

The `Tracer` object is fully managed by sleuth, so you rarely need to affect it. That said, Sleuth supports a number of `Customizer` types, that allow you to configure anything not already done by Sleuth with auto-configuration or properties.

If you define one of the following as a `Bean`, Sleuth will invoke it to customize behaviour:

- `RpcTracingCustomizer` - for RPC tagging and sampling policy
- `HttpTracingCustomizer` - for HTTP tagging and sampling policy
- `MessagingTracingCustomizer` - for messaging tagging and sampling policy
- `CurrentTraceContextCustomizer` - to integrate decorators such as correlation.
- `BaggagePropagationCustomizer` - for propagating baggage fields in process and over headers
- `CorrelationScopeDecoratorCustomizer` - for scope decorations such as MDC (logging) field correlation

8.11.1. HTTP

Data Policy

The default span data policy for HTTP requests is described in Brave: github.com/openzipkin/brave/tree/master/instrumentation/http#span-data-policy

To add different data to the span, you need to register a bean of type `brave.http.HttpRequestParser` or `brave.http.HttpResponseParser` based on when the data is collected.

The bean names correspond to the request or response side, and whether it is a client or server. For example, `sleuthHttpClientRequestParser` changes what is collected before a client request is sent to the server.

For your convenience `@HttpClientRequestParser`, `@HttpClientResponseParser` and corresponding server annotations can be used to inject the proper beans or to reference the bean names via their static String `NAME` fields.

Here's an example adding the HTTP url in addition to defaults:

```
@Configuration
class Config {
    @Bean(name = { HttpClientRequestParser.NAME, HttpServerRequestParser.NAME })
    HttpRequestParser sleuthHttpServerRequestParser() {
        return (req, context, span) -> {
            HttpRequestParser.DEFAULT.parse(req, context, span);
            String url = req.url();
            if (url != null) {
                span.tag("http.url", url);
            }
        };
    }
}
```

Sampling

If client /server sampling is required, just register a bean of type `brave.sampler.SamplerFunction<HttpRequest>` and name the bean `sleuthHttpClientSampler` for client sampler and `sleuthHttpServerSampler` for server sampler.

For your convenience the `@HttpClientSampler` and `@HttpServerSampler` annotations can be used to inject the proper beans or to reference the bean names via their static String `NAME` fields.

Check out Brave's code to see an example of how to make a path-based sampler github.com/openzipkin/brave/tree/master/instrumentation/http#sampling-policy

If you want to completely rewrite the `HttpTracing` bean you can use the `SkipPatternProvider` interface to retrieve the URL `Pattern` for spans that should be not sampled. Below you can see an example of usage of `SkipPatternProvider` inside a server side, `Sampler<HttpRequest>`.

```

@Configuration
class Config {
    @Bean(name = HttpServerSampler.NAME)
    SamplerFunction<HttpRequest> myHttpSampler(SkipPatternProvider provider) {
        Pattern pattern = provider.skipPattern();
        return request -> {
            String url = request.path();
            boolean shouldSkip = pattern.matcher(url).matches();
            if (shouldSkip) {
                return false;
            }
            return null;
        };
    }
}

```

8.11.2. TracingFilter

You can also modify the behavior of the `TracingFilter`, which is the component that is responsible for processing the input HTTP request and adding tags basing on the HTTP response. You can customize the tags or modify the response headers by registering your own instance of the `TracingFilter` bean.

In the following example, we register the `TracingFilter` bean, add the `ZIPKIN-TRACE-ID` response header containing the current Span's trace id, and add a tag with key `custom` and a value `tag` to the span.

```

@Component
@Order(TraceWebServletAutoConfiguration.TRACING_FILTER_ORDER + 1)
class MyFilter extends GenericFilterBean {

    private final Tracer tracer;

    MyFilter(Tracer tracer) {
        this.tracer = tracer;
    }

    @Override
    public void doFilter(ServletRequest request, ServletResponse response,
        FilterChain chain) throws IOException, ServletException {
        Span currentSpan = this.tracer.currentSpan();
        if (currentSpan == null) {
            chain.doFilter(request, response);
            return;
        }
        // for readability we're returning trace id in a hex form
        ((HttpServletResponse) response).addHeader("ZIPKIN-TRACE-ID",
            currentSpan.context().traceIdString());
        // we can also add some custom tags
        currentSpan.tag("custom", "tag");
        chain.doFilter(request, response);
    }
}

```

8.11.3. Messaging

Sleuth automatically configures the `MessagingTracing` bean which serves as a foundation for Messaging instrumentation such as Kafka or JMS.

If a customization of producer / consumer sampling of messaging traces is required, just register a bean of type `brave.sampler.SamplerFunction<MessagingRequest>` and name the bean `sleuthProducerSampler` for producer sampler and `sleuthConsumerSampler` for consumer sampler.

For your convenience the `@ProducerSampler` and `@ConsumerSampler` annotations can be used to inject the proper beans or to reference the bean names via their static String `NAME` fields.

Ex. Here's a sampler that traces 100 consumer requests per second, except for the "alerts" channel. Other requests will use a global rate provided by the `Tracing` component.

```

@Configuration
class Config {
}

```

For more, see github.com/openzipkin/brave/tree/master/instrumentation/messaging#sampling

8.11.4. RPC

Sleuth automatically configures the `RpcTracing` bean which serves as a foundation for RPC instrumentation such as gRPC or Dubbo.

If a customization of client / server sampling of the RPC traces is required, just register a bean of type `brave.sampler.SamplerFunction<RpcRequest>` and name the bean `sleuthRpcClientSampler` for client sampler and `sleuthRpcServerSampler` for server sampler.

For your convenience the `@RpcClientSampler` and `@RpcServerSampler` annotations can be used to inject the proper beans or to reference the bean names via their static String `NAME` fields.

Ex. Here's a sampler that traces 100 "GetUserToken" server requests per second. This doesn't start new traces for requests to the health check service. Other requests will use the global sampling configuration.

```
@Configuration
class Config {
    @Bean(name = RpcServerSampler.NAME)
    SamplerFunction<RpcRequest> myRpcSampler() {
        Matcher<RpcRequest> userAuth = and(serviceEquals("users.UserService"),
            methodEquals("GetUserToken"));
        return RpcRuleSampler.newBuilder()
            .putRule(serviceEquals("grpc.health.v1.Health"), Sampler.NEVER_SAMPLE)
            .putRule(userAuth, RateLimitingSampler.create(100)).build();
    }
}
```

For more, see github.com/openzipkin/brave/tree/master/instrumentation/rpc#sampling-policy

8.11.5. Custom service name

By default, Sleuth assumes that, when you send a span to Zipkin, you want the span's service name to be equal to the value of the `spring.application.name` property. That is not always the case, though. There are situations in which you want to explicitly provide a different service name for all spans coming from your application. To achieve that, you can pass the following property to your application to override that value (the example is for a service named `myService`):

```
spring.zipkin.service.name: myService
```

8.11.6. Customization of Reported Spans

Before reporting spans (for example, to Zipkin) you may want to modify that span in some way. You can do so by implementing a `SpanHandler`.

In Sleuth, we generate spans with a fixed name. Some users want to modify the name depending on

values of tags. You can implement the `SpanHandler` interface to alter that name.

The following example shows how to register two beans that implement `SpanHandler`:

```
@Bean
SpanHandler handlerOne() {
    return new SpanHandler() {
        @Override
        public boolean end(TraceContext traceContext, MutableSpan span,
            Cause cause) {
            span.name("foo");
            return true; // keep this span
        }
    };
}

@Bean
SpanHandler handlerTwo() {
    return new SpanHandler() {
        @Override
        public boolean end(TraceContext traceContext, MutableSpan span,
            Cause cause) {
            span.name(span.name() + " bar");
            return true; // keep this span
        }
    };
}
```

The preceding example results in changing the name of the reported span to `foo bar`, just before it gets reported (for example, to Zipkin).

8.11.7. Host Locator



This section is about defining **host** from service discovery. It is **NOT** about finding Zipkin through service discovery.

To define the host that corresponds to a particular span, we need to resolve the host name and port. The default approach is to take these values from server properties. If those are not set, we try to retrieve the host name from the network interfaces.

If you have the discovery client enabled and prefer to retrieve the host address from the registered instance in a service registry, you have to set the `spring.zipkin.locator.discovery.enabled` property (it is applicable for both HTTP-based and Stream-based span reporting), as follows:

```
spring.zipkin.locator.discovery.enabled: true
```

8.12. Sending Spans to Zipkin

By default, if you add `spring-cloud-starter-zipkin` as a dependency to your project, when the span is closed, it is sent to Zipkin over HTTP. The communication is asynchronous. You can configure the URL by setting the `spring.zipkin.baseUrl` property, as follows:

```
spring.zipkin.baseUrl: https://192.168.99.100:9411/
```

If you want to find Zipkin through service discovery, you can pass the Zipkin's service ID inside the URL, as shown in the following example for `zipkinserver` service ID:

```
spring.zipkin.baseUrl: https://zipkinserver/
```

To disable this feature just set `spring.zipkin.discoveryClientEnabled` to `false`.

When the Discovery Client feature is enabled, Sleuth uses `LoadBalancerClient` to find the URL of the Zipkin Server. It means that you can set up the load balancing configuration e.g. via Ribbon.

```
zipkinserver:
  ribbon:
    ListOfServers: host1,host2
```

If you have `web`, `rabbit`, `activemq` or `kafka` together on the classpath, you might need to pick the means by which you would like to send spans to zipkin. To do so, set `web`, `rabbit`, `activemq` or `kafka` to the `spring.zipkin.sender.type` property. The following example shows setting the sender type for `web`:

```
spring.zipkin.sender.type: web
```

To customize the `RestTemplate` that sends spans to Zipkin via HTTP, you can register the `ZipkinRestTemplateCustomizer` bean.

```
@Configuration
class MyConfig {
    @Bean ZipkinRestTemplateCustomizer myCustomizer() {
        return new ZipkinRestTemplateCustomizer() {
            @Override
            void customize(RestTemplate restTemplate) {
                // customize the RestTemplate
            }
        };
    }
}
```

If, however, you would like to control the full process of creating the `RestTemplate` object, you will have to create a bean of `zipkin2.reporter.Sender` type.

```
@Bean Sender myRestTemplateSender(ZipkinProperties zipkin,
    ZipkinRestTemplateCustomizer zipkinRestTemplateCustomizer) {
    RestTemplate restTemplate = mySuperCustomRestTemplate();
    zipkinRestTemplateCustomizer.customize(restTemplate);
    return myCustomSender(zipkin, restTemplate);
}
```

8.13. Integrations

8.13.1. OpenTracing

Spring Cloud Sleuth is compatible with [OpenTracing](#). If you have OpenTracing on the classpath, we automatically register the OpenTracing `Tracer` bean. If you wish to disable this, set `spring.sleuth.opentracing.enabled` to `false`

8.13.2. Runnable and Callable

If you wrap your logic in `Runnable` or `Callable`, you can wrap those classes in their Sleuth representative, as shown in the following example for `Runnable`:

```
Runnable runnable = new Runnable() {
    @Override
    public void run() {
        // do some work
    }

    @Override
    public String toString() {
        return "spanNameFromToStringMethod";
    }
};
// Manual `TraceRunnable` creation with explicit "calculateTax" Span name
Runnable traceRunnable = new TraceRunnable(this.tracing, spanNamer, runnable,
    "calculateTax");
// Wrapping `Runnable` with `Tracing`. That way the current span will be available
// in the thread of `Runnable`
Runnable traceRunnableFromTracer = this.tracing.currentTraceContext()
    .wrap(runnable);
```

The following example shows how to do so for `Callable`:

```

Callable<String> callable = new Callable<String>() {
    @Override
    public String call() throws Exception {
        return someLogic();
    }

    @Override
    public String toString() {
        return "spanNameFromToStringMethod";
    }
};
// Manual `TraceCallable` creation with explicit "calculateTax" Span name
Callable<String> traceCallable = new TraceCallable<>(this.tracing, spanNamer,
    callable, "calculateTax");
// Wrapping `Callable` with `Tracing`. That way the current span will be available
// in the thread of `Callable`
Callable<String> traceCallableFromTracer = this.tracing.currentTraceContext()
    .wrap(callable);

```

That way, you ensure that a new span is created and closed for each execution.

8.13.3. Spring Cloud CircuitBreaker

If you have Spring Cloud CircuitBreaker on the classpath, we will wrap the passed command `Supplier` and the fallback `Function` in its trace representations. In order to disable this instrumentation set `spring.sleuth.circuitbreaker.enabled` to `false`.

8.13.4. RxJava

We registering a custom `RxJavaSchedulersHook` that wraps all `Action0` instances in their Sleuth representative, which is called `TraceAction`. The hook either starts or continues a span, depending on whether tracing was already going on before the Action was scheduled. To disable the custom `RxJavaSchedulersHook`, set the `spring.sleuth.rxjava.schedulers.hook.enabled` to `false`.

You can define a list of regular expressions for thread names for which you do not want spans to be created. To do so, provide a comma-separated list of regular expressions in the `spring.sleuth.rxjava.schedulers.ignoredthreads` property.



The suggest approach to reactive programming and Sleuth is to use the Reactor support.

8.13.5. HTTP integration

Features from this section can be disabled by setting the `spring.sleuth.web.enabled` property with value equal to `false`.

HTTP Filter

Through the `TracingFilter`, all sampled incoming requests result in creation of a Span. That Span's name is `http: + the path to which the request was sent`. For example, if the request was sent to `/this/that` then the name will be `http:/this/that`. You can configure which URIs you would like to skip by setting the `spring.sleuth.web.skipPattern` property. If you have `ManagementServerProperties` on classpath, its value of `contextPath` gets appended to the provided skip pattern. If you want to reuse the Sleuth's default skip patterns and just append your own, pass those patterns by using the `spring.sleuth.web.additionalSkipPattern`.

By default, all the spring boot actuator endpoints are automatically added to the skip pattern. If you want to disable this behaviour set `spring.sleuth.web.ignore-auto-configured-skip-patterns` to `true`.

To change the order of tracing filter registration, please set the `spring.sleuth.web.filter-order` property.

To disable the filter that logs uncaught exceptions you can disable the `spring.sleuth.web.exception-throwing-filter-enabled` property.

HandlerInterceptor

Since we want the span names to be precise, we use a `TraceHandlerInterceptor` that either wraps an existing `HandlerInterceptor` or is added directly to the list of existing `HandlerInterceptors`. The `TraceHandlerInterceptor` adds a special request attribute to the given `HttpServletRequest`. If the `TracingFilter` does not see this attribute, it creates a “fallback” span, which is an additional span created on the server side so that the trace is presented properly in the UI. If that happens, there is probably missing instrumentation. In that case, please file an issue in Spring Cloud Sleuth.

Async Servlet support

If your controller returns a `Callable` or a `WebAsyncTask`, Spring Cloud Sleuth continues the existing span instead of creating a new one.

WebFlux support

Through `TraceWebFilter`, all sampled incoming requests result in creation of a Span. That Span's name is `http: + the path to which the request was sent`. For example, if the request was sent to `/this/that`, the name is `http:/this/that`. You can configure which URIs you would like to skip by using the `spring.sleuth.web.skipPattern` property. If you have `ManagementServerProperties` on the classpath, its value of `contextPath` gets appended to the provided skip pattern. If you want to reuse Sleuth's default skip patterns and append your own, pass those patterns by using the `spring.sleuth.web.additionalSkipPattern`.

To change the order of tracing filter registration, please set the `spring.sleuth.web.filter-order` property.

Dubbo RPC support

Via the integration with Brave, Spring Cloud Sleuth supports `Dubbo`. It's enough to add the `brave-instrumentation-dubbo` dependency:

```
<dependency>
  <groupId>io.zipkin.brave</groupId>
  <artifactId>brave-instrumentation-dubbo</artifactId>
</dependency>
```

You need to also set a `dubbo.properties` file with the following contents:

```
dubbo.provider.filter=tracing
dubbo.consumer.filter=tracing
```

You can read more about Brave - Dubbo integration [here](#). An example of Spring Cloud Sleuth and Dubbo can be found [here](#).

8.13.6. HTTP Client Integration

Synchronous Rest Template

We inject a `RestTemplate` interceptor to ensure that all the tracing information is passed to the requests. Each time a call is made, a new Span is created. It gets closed upon receiving the response. To block the synchronous `RestTemplate` features, set `spring.sleuth.web.client.enabled` to `false`.



You have to register `RestTemplate` as a bean so that the interceptors get injected. If you create a `RestTemplate` instance with a `new` keyword, the instrumentation does NOT work.

Asynchronous Rest Template



Starting with Sleuth `2.0.0`, we no longer register a bean of `AsyncRestTemplate` type. It is up to you to create such a bean. Then we instrument it.

To block the `AsyncRestTemplate` features, set `spring.sleuth.web.async.client.enabled` to `false`. To disable creation of the default `TraceAsyncClientHttpRequestFactoryWrapper`, set `spring.sleuth.web.async.client.factory.enabled` to `false`. If you do not want to create `AsyncRestClient` at all, set `spring.sleuth.web.async.client.template.enabled` to `false`.

Multiple Asynchronous Rest Templates

Sometimes you need to use multiple implementations of the Asynchronous Rest Template. In the following snippet, you can see an example of how to set up such a custom `AsyncRestTemplate`:

```

@Configuration
@EnableAutoConfiguration
static class Config {

    @Bean(name = "customAsyncRestTemplate")
    public AsyncRestTemplate traceAsyncRestTemplate() {
        return new AsyncRestTemplate(asyncClientFactory(),
            clientHttpRequestFactory());
    }

    private ClientHttpRequestFactory clientHttpRequestFactory() {
        ClientHttpRequestFactory clientHttpRequestFactory = new
CustomClientHttpRequestFactory();
        // CUSTOMIZE HERE
        return clientHttpRequestFactory;
    }

    private AsyncClientHttpRequestFactory asyncClientFactory() {
        AsyncClientHttpRequestFactory factory = new
CustomAsyncClientHttpRequestFactory();
        // CUSTOMIZE HERE
        return factory;
    }
}

```

WebClient

We inject a `ExchangeFilterFunction` implementation that creates a span and, through on-success and on-error callbacks, takes care of closing client-side spans.

To block this feature, set `spring.sleuth.web.client.enabled` to `false`.



You have to register `WebClient` as a bean so that the tracing instrumentation gets applied. If you create a `WebClient` instance with a `new` keyword, the instrumentation does NOT work.

Traverson

If you use the `Traverson` library, you can inject a `RestTemplate` as a bean into your `Traverson` object. Since `RestTemplate` is already intercepted, you get full support for tracing in your client. The following pseudo code shows how to do that:

```
@Autowired RestTemplate restTemplate;

Traverson traverson = new Traverson(URI.create("https://some/address"),
    MediaType.APPLICATION_JSON,
    MediaType.APPLICATION_JSON_UTF8).setRestOperations(restTemplate);
// use Traverson
```

Apache `HttpClientBuilder` and `HttpAsyncClientBuilder`

We instrument the `HttpClientBuilder` and `HttpAsyncClientBuilder` so that tracing context gets injected to the sent requests.

To block these features, set `spring.sleuth.web.client.enabled` to `false`.

Netty `HttpClient`

We instrument the Netty's `HttpClient`.

To block this feature, set `spring.sleuth.web.client.enabled` to `false`.



You have to register `HttpClient` as a bean so that the instrumentation happens. If you create a `HttpClient` instance with a `new` keyword, the instrumentation does NOT work.

`UserInfoRestTemplateCustomizer`

We instrument the Spring Security's `UserInfoRestTemplateCustomizer`.

To block this feature, set `spring.sleuth.web.client.enabled` to `false`.

8.13.7. Feign

By default, Spring Cloud Sleuth provides integration with Feign through `TraceFeignClientAutoConfiguration`. You can disable it entirely by setting `spring.sleuth.feign.enabled` to `false`. If you do so, no Feign-related instrumentation take place.

Part of Feign instrumentation is done through a `FeignBeanPostProcessor`. You can disable it by setting `spring.sleuth.feign.processor.enabled` to `false`. If you set it to `false`, Spring Cloud Sleuth does not instrument any of your custom Feign components. However, all the default instrumentation is still there.

8.13.8. gRPC

Spring Cloud Sleuth provides instrumentation for `gRPC` through `TraceGrpcAutoConfiguration`. You can disable it entirely by setting `spring.sleuth.grpc.enabled` to `false`.

Variant 1

Dependencies



The gRPC integration relies on two external libraries to instrument clients and servers and both of those libraries must be on the class path to enable the instrumentation.

Maven:

```
<dependency>
  <groupId>io.github.lognet</groupId>
  <artifactId>grpc-spring-boot-starter</artifactId>
</dependency>
<dependency>
  <groupId>io.zipkin.brave</groupId>
  <artifactId>brave-instrumentation-grpc</artifactId>
</dependency>
```

Gradle:

```
compile("io.github.lognet:grpc-spring-boot-starter")
compile("io.zipkin.brave:brave-instrumentation-grpc")
```

Server Instrumentation

Spring Cloud Sleuth leverages `grpc-spring-boot-starter` to register Brave's gRPC server interceptor with all services annotated with `@GRpcService`.

Client Instrumentation

gRPC clients leverage a `ManagedChannelBuilder` to construct a `ManagedChannel` used to communicate to the gRPC server. The native `ManagedChannelBuilder` provides static methods as entry points for construction of `ManagedChannel` instances, however, this mechanism is outside the influence of the Spring application context.



Spring Cloud Sleuth provides a `SpringAwareManagedChannelBuilder` that can be customized through the Spring application context and injected by gRPC clients. **This builder must be used when creating `ManagedChannel` instances.**

Sleuth creates a `TracingManagedChannelBuilderCustomizer` which injects Brave's client interceptor into the `SpringAwareManagedChannelBuilder`.

Variant 2

`Grpc Spring Boot Starter` automatically detects the presence of Spring Cloud Sleuth and Brave's instrumentation for gRPC and registers the necessary client and/or server tooling.

8.13.9. Asynchronous Communication

@Async Annotated methods

In Spring Cloud Sleuth, we instrument async-related components so that the tracing information is passed between threads. You can disable this behavior by setting the value of `spring.sleuth.async.enabled` to `false`.

If you annotate your method with `@Async`, we automatically create a new Span with the following characteristics:

- If the method is annotated with `@SpanName`, the value of the annotation is the Span's name.
- If the method is not annotated with `@SpanName`, the Span name is the annotated method name.
- The span is tagged with the method's class name and method name.

@Scheduled Annotated Methods

In Spring Cloud Sleuth, we instrument scheduled method execution so that the tracing information is passed between threads. You can disable this behavior by setting the value of `spring.sleuth.scheduled.enabled` to `false`.

If you annotate your method with `@Scheduled`, we automatically create a new span with the following characteristics:

- The span name is the annotated method name.
- The span is tagged with the method's class name and method name.

If you want to skip span creation for some `@Scheduled` annotated classes, you can set the `spring.sleuth.scheduled.skipPattern` with a regular expression that matches the fully qualified name of the `@Scheduled` annotated class.

Executor, ExecutorService, and ScheduledExecutorService

We provide `LazyTraceExecutor`, `TraceableExecutorService`, and `TraceableScheduledExecutorService`. Those implementations create spans each time a new task is submitted, invoked, or scheduled.

The following example shows how to pass tracing information with `TraceableExecutorService` when working with `CompletableFuture`:

```
CompletableFuture<Long> completableFuture = CompletableFuture.supplyAsync(() -> {
    // perform some logic
    return 1_000_000L;
}, new TraceableExecutorService(beanFactory, executorService,
    // 'calculateTax' explicitly names the span - this param is optional
    "calculateTax"));
```



Sleuth does not work with `parallelStream()` out of the box. If you want to have the tracing information propagated through the stream, you have to use the approach with `supplyAsync(...)`, as shown earlier.

If there are beans that implement the `Executor` interface that you would like to exclude from span creation, you can use the `spring.sleuth.async.ignored-beans` property where you can provide a list of bean names.

Customization of Executors

Sometimes, you need to set up a custom instance of the `AsyncExecutor`. The following example shows how to set up such a custom `Executor`:

```
@Configuration
@EnableAutoConfiguration
@EnableAsync
// add the infrastructure role to ensure that the bean gets auto-proxied
@Role(BeanDefinition.ROLE_INFRASTRUCTURE)
static class CustomExecutorConfig extends AsyncConfigurerSupport {

    @Autowired
    BeanFactory beanFactory;

    @Override
    public Executor getAsyncExecutor() {
        ThreadPoolTaskExecutor executor = new ThreadPoolTaskExecutor();
        // CUSTOMIZE HERE
        executor.setCorePoolSize(7);
        executor.setMaxPoolSize(42);
        executor.setQueueCapacity(11);
        executor.setThreadNamePrefix("MyExecutor-");
        // DON'T FORGET TO INITIALIZE
        executor.initialize();
        return new LazyTraceExecutor(this.beanFactory, executor);
    }
}
```



To ensure that your configuration gets post processed, remember to add the `@Role(BeanDefinition.ROLE_INFRASTRUCTURE)` on your `@Configuration` class

8.13.10. Messaging

Features from this section can be disabled by setting the `spring.sleuth.messaging.enabled` property with value equal to `false`.

Spring Integration and Spring Cloud Stream

Spring Cloud Sleuth integrates with [Spring Integration](#). It creates spans for publish and subscribe events. To disable Spring Integration instrumentation, set `spring.sleuth.integration.enabled` to `false`.

You can provide the `spring.sleuth.integration.patterns` pattern to explicitly provide the names of channels that you want to include for tracing. By default, all channels but `hystrixStreamOutput` channel are included.



When using the `Executor` to build a Spring Integration `IntegrationFlow`, you must use the untraced version of the `Executor`. Decorating the Spring Integration Executor Channel with `TraceableExecutorService` causes the spans to be improperly closed.

If you want to customize the way tracing context is read from and written to message headers, it's enough for you to register beans of types:

- `Propagation.Setter<MessageHeaderAccessor, String>` - for writing headers to the message
- `Propagation.Getter<MessageHeaderAccessor, String>` - for reading headers from the message

Spring RabbitMq

We instrument the `RabbitTemplate` so that tracing headers get injected into the message.

To block this feature, set `spring.sleuth.messaging.rabbit.enabled` to `false`.

Spring Kafka

We instrument the Spring Kafka's `ProducerFactory` and `ConsumerFactory` so that tracing headers get injected into the created Spring Kafka's `Producer` and `Consumer`.

To block this feature, set `spring.sleuth.messaging.kafka.enabled` to `false`.

Spring Kafka Streams

We instrument the `KafkaStreams KafkaClientSupplier` so that tracing headers get injected into the `Producer` and `Consumer`'s. A `KafkaStreamsTracing` bean allows for further instrumentation through additional `TransformerSupplier` and `ProcessorSupplier` methods.

To block this feature, set `spring.sleuth.messaging.kafka.streams.enabled` to `false`.

Spring JMS

We instrument the `JmsTemplate` so that tracing headers get injected into the message. We also support `@JmsListener` annotated methods on the consumer side.

To block this feature, set `spring.sleuth.messaging.jms.enabled` to `false`.



We don't support baggage propagation for JMS

Spring Cloud AWS Messaging SQS

We instrument `@SqsListener` which is provided by `org.springframework.cloud:spring-cloud-aws-messaging` so that tracing headers get extracted from the message and a trace gets put into the context.

To block this feature, set `spring.sleuth.messaging.sqs.enabled` to `false`.

8.13.11. Redis

We set `tracing` property to Lettuce `ClientResources` instance to enable Brave tracing built in Lettuce. To disable Redis support, set the `spring.sleuth.redis.enabled` property to `false`.

8.13.12. Quartz

We instrument quartz jobs by adding Job/Trigger listeners to the Quartz Scheduler.

To turn off this feature, set the `spring.sleuth.quartz.enabled` property to `false`.

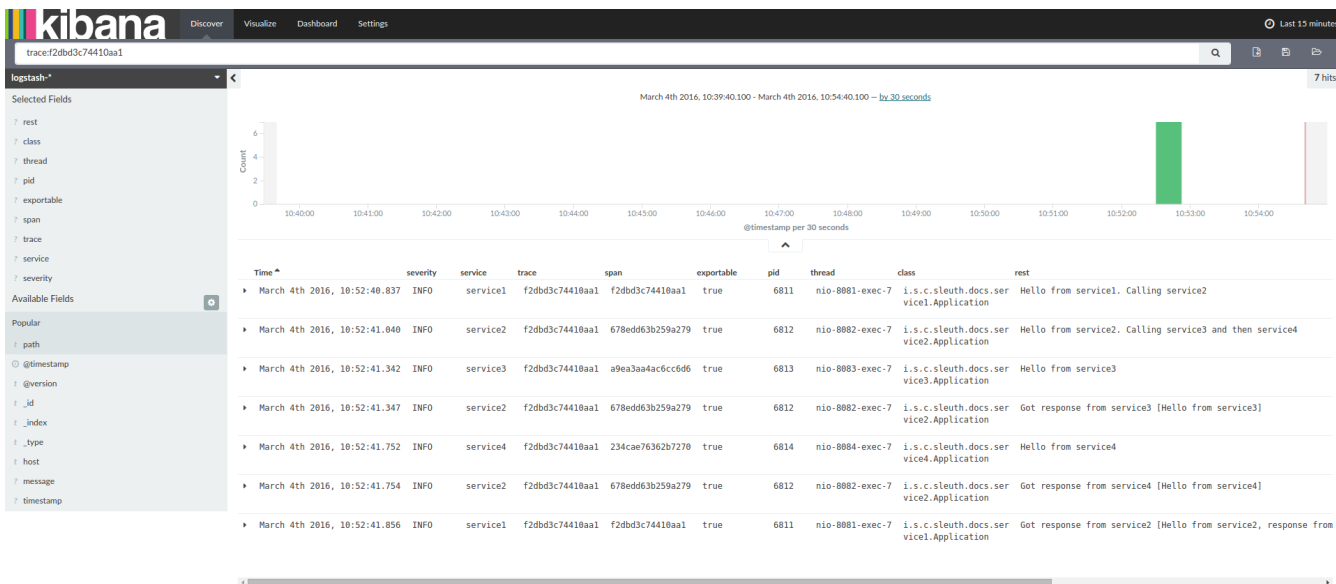
8.13.13. Project Reactor

For projects depending on Project Reactor such as Spring Cloud Gateway, we suggest turning the `spring.sleuth.reactor.decorate-on-each` option to `false`. That way an increased performance gain should be observed in comparison to the standard instrumentation mechanism. What this option does is it will wrap decorate `onLast` operator instead of `onEach` which will result in creation of far fewer objects. The downside of this is that when Project Reactor will change threads, the trace propagation will continue without issues, however anything relying on the `ThreadLocal` such as e.g. MDC entries can be buggy.

8.14. Log integration

Sleuth configures the logging context with variables including the service name (`spring.zipkin.service.name`) and the trace ID (`traceId`). These help you connect logs with distributed traces and allow you choice in what tools you use to troubleshoot your services.

If you use a log aggregating tool (such as [Kibana](#), [Splunk](#), and others), you can order the events that took place. An example from Kibana would resemble the following image:



If you want to use [Logstash](#), the following listing shows the Grok pattern for Logstash:

```

filter {
  # pattern matching logback pattern
  grok {
    match => { "message" =>
"%{TIMESTAMP_ISO8601:timestamp}\s+{%{LOGLEVEL:severity}}\s+\[%{DATA:service},%{DATA:trace},%{DATA:span}\]\s+{%{DATA:pid}}\s+---\s+\[%{DATA:thread}\]\s+{%{DATA:class}}\s+:\s+{%{GREEDYDATA:rest}}" }
  }
  date {
    match => ["timestamp", "ISO8601"]
  }
  mutate {
    remove_field => ["timestamp"]
  }
}

```



If you want to use Grok together with the logs from Cloud Foundry, you have to use the following pattern:

```

filter {
  # pattern matching logback pattern
  grok {
    match => { "message" =>
"(?m)OUT\s+{%TIMESTAMP_ISO8601:timestamp}\s+{%LOGLEVEL:severity}\s+\[%{DATA:service},%
{DATA:trace},%{DATA:span}\}\s+{%DATA:pid}\s+---
\s+\[%{DATA:thread}\}\s+{%DATA:class}\s+:\s+{%GREEDYDATA:rest}" }
  }
  date {
    match => ["timestamp", "ISO8601"]
  }
  mutate {
    remove_field => ["timestamp"]
  }
}

```

8.14.1. JSON Logback with Logstash

Often, you do not want to store your logs in a text file but in a JSON file that Logstash can immediately pick. To do so, you have to do the following (for readability, we pass the dependencies in the `groupId:artifactId:version` notation).

Dependencies Setup

1. Ensure that Logback is on the classpath (`ch.qos.logback:logback-core`).
2. Add Logstash Logback encode. For example, to use version `4.6`, add `net.logstash.logback:logstash-logback-encoder:4.6`.

Logback Setup

Consider the following example of a Logback configuration file (logback-spring.xml).

```

<?xml version="1.0" encoding="UTF-8"?>
<configuration>
  <include resource="org/springframework/boot/logging/logback/defaults.xml"/>
  <springProperty scope="context" name="springAppName"
source="spring.application.name"/>
  <!-- Example for logging into the build folder of your project -->
  <property name="LOG_FILE" value="${BUILD_FOLDER:-build}/${springAppName}"/>

  <!-- You can override this to have a custom pattern -->
  <property name="CONSOLE_LOG_PATTERN"
value="%clr(%d{yyyy-MM-dd HH:mm:ss.SSS}){faint}
%clr(${LOG_LEVEL_PATTERN:-%5p}) %clr(${PID:- }){magenta} %clr(---){faint}
%clr([%15.15t]){faint} %clr(%-40.40logger{39}){cyan} %clr(:){faint}
%n%n${LOG_EXCEPTION_CONVERSION_WORD:-%wEx}"/>

  <!-- Appender to log to console -->
  <appender name="console" class="ch.qos.logback.core.ConsoleAppender">

```

```

<filter class="ch.qos.logback.classic.filter.ThresholdFilter">
  <!-- Minimum logging level to be presented in the console logs-->
  <level>DEBUG</level>
</filter>
<encoder>
  <pattern>${CONSOLE_LOG_PATTERN}</pattern>
  <charset>utf8</charset>
</encoder>
</appender>

<!-- Appender to log to file -->
<appender name="flatfile" class="ch.qos.logback.core.rolling.RollingFileAppender">
  <file>${LOG_FILE}</file>
  <rollingPolicy class="ch.qos.logback.core.rolling.TimeBasedRollingPolicy">
    <fileNamePattern>${LOG_FILE}.%d{yyyy-MM-dd}.gz</fileNamePattern>
    <maxHistory>7</maxHistory>
  </rollingPolicy>
  <encoder>
    <pattern>${CONSOLE_LOG_PATTERN}</pattern>
    <charset>utf8</charset>
  </encoder>
</appender>

<!-- Appender to log to file in a JSON format -->
<appender name="logstash" class="ch.qos.logback.core.rolling.RollingFileAppender">
  <file>${LOG_FILE}.json</file>
  <rollingPolicy class="ch.qos.logback.core.rolling.TimeBasedRollingPolicy">
    <fileNamePattern>${LOG_FILE}.json.%d{yyyy-MM-dd}.gz</fileNamePattern>
    <maxHistory>7</maxHistory>
  </rollingPolicy>
  <encoder
class="net.logstash.logback.encoder.LoggingEventCompositeJsonEncoder">
    <providers>
      <timestamp>
        <timeZone>UTC</timeZone>
      </timestamp>
      <pattern>
        <pattern>
          {
            "timestamp": "@timestamp",
            "severity": "%level",
            "service": "${springAppName:-}",
            "trace": "%X{traceId:-}",
            "span": "%X{spanId:-}",
            "pid": "${PID:-}",
            "thread": "%thread",
            "class": "%logger{40}",
            "rest": "%message"
          }
        </pattern>
      </pattern>
    </providers>

```

```
</encoder>
</appender>
<root level="INFO">
  <appender-ref ref="console"/>
  <!-- uncomment this to have also JSON logs -->
  <!--<appender-ref ref="logstash"/>-->
  <!--<appender-ref ref="flatfile"/>-->
</root>
</configuration>
```

That Logback configuration file:

- Logs information from the application in a JSON format to a `build/${spring.application.name}.json` file.
- Has commented out two additional appenders: console and standard log file.
- Has the same logging pattern as the one presented in the previous section.



If you use a custom `logback-spring.xml`, you must pass the `spring.application.name` in the `bootstrap` rather than the `application` property file. Otherwise, your custom logback file does not properly read the property.

8.15. Configuration properties

To see the list of all Sleuth related configuration properties please check [the Appendix page](#).

Chapter 9. Spring Cloud Consul

2020.0.0-M2

This project provides Consul integrations for Spring Boot apps through autoconfiguration and binding to the Spring Environment and other Spring programming model idioms. With a few simple annotations you can quickly enable and configure the common patterns inside your application and build large distributed systems with Consul based components. The patterns provided include Service Discovery, Control Bus and Configuration. Intelligent Routing (Zuul) and Client Side Load Balancing (Ribbon), Circuit Breaker (Hystrix) are provided by integration with Spring Cloud Netflix.

9.1. Install Consul

Please see the [installation documentation](#) for instructions on how to install Consul.

9.2. Consul Agent

A Consul Agent client must be available to all Spring Cloud Consul applications. By default, the Agent client is expected to be at `localhost:8500`. See the [Agent documentation](#) for specifics on how to start an Agent client and how to connect to a cluster of Consul Agent Servers. For development, after you have installed consul, you may start a Consul Agent using the following command:

```
./src/main/bash/local_run_consul.sh
```

This will start an agent in server mode on port 8500, with the ui available at [localhost:8500](#)

9.3. Service Discovery with Consul

Service Discovery is one of the key tenets of a microservice based architecture. Trying to hand configure each client or some form of convention can be very difficult to do and can be very brittle. Consul provides Service Discovery services via an [HTTP API](#) and [DNS](#). Spring Cloud Consul leverages the HTTP API for service registration and discovery. This does not prevent non-Spring Cloud applications from leveraging the DNS interface. Consul Agents servers are run in a [cluster](#) that communicates via a [gossip protocol](#) and uses the [Raft consensus protocol](#).

9.3.1. How to activate

To activate Consul Service Discovery use the starter with group `org.springframework.cloud` and artifact id `spring-cloud-starter-consul-discovery`. See the [Spring Cloud Project page](#) for details on setting up your build system with the current Spring Cloud Release Train.

9.3.2. Registering with Consul

When a client registers with Consul, it provides meta-data about itself such as host and port, id, name and tags. An HTTP [Check](#) is created by default that Consul hits the `/health` endpoint every 10

seconds. If the health check fails, the service instance is marked as critical.

Example Consul client:

```
@SpringBootApplication
@RestController
public class Application {

    @RequestMapping("/")
    public String home() {
        return "Hello world";
    }

    public static void main(String[] args) {
        new SpringApplicationBuilder(Application.class).web(true).run(args);
    }
}
```

(i.e. utterly normal Spring Boot app). If the Consul client is located somewhere other than `localhost:8500`, the configuration is required to locate the client. Example:

application.yml

```
spring:
  cloud:
    consul:
      host: localhost
      port: 8500
```



If you use [Spring Cloud Consul Config](#), the above values will need to be placed in `bootstrap.yml` instead of `application.yml`.

The default service name, instance id and port, taken from the `Environment`, are `spring.application.name`, the Spring Context ID and `server.port` respectively.

To disable the Consul Discovery Client you can set `spring.cloud.consul.discovery.enabled` to `false`. Consul Discovery Client will also be disabled when `spring.cloud.discovery.enabled` is set to `false`.

To disable the service registration you can set `spring.cloud.consul.discovery.register` to `false`.

Registering Management as a Separate Service

When management server port is set to something different than the application port, by setting `management.server.port` property, management service will be registered as a separate service than the application service. For example:

application.yml

```
spring:
  application:
    name: myApp
management:
  server:
    port: 4452
```

Above configuration will register following 2 services:

- Application Service:

```
ID: myApp
Name: myApp
```

- Management Service:

```
ID: myApp-management
Name: myApp-management
```

Management service will inherit its `instanceId` and `serviceName` from the application service. For example:

application.yml

```
spring:
  application:
    name: myApp
management:
  server:
    port: 4452
spring:
  cloud:
    consul:
      discovery:
        instance-id: custom-service-id
        serviceName: myprefix-${spring.application.name}
```

Above configuration will register following 2 services:

- Application Service:

```
ID: custom-service-id
Name: myprefix-myApp
```


- Management Service:

```
ID: custom-service-id-management
Name: myprefix-myApp-management
```

Further customization is possible via following properties:

```
/** Port to register the management service under (defaults to management port) */
spring.cloud.consul.discovery.management-port

/** Suffix to use when registering management service (defaults to "management" */
spring.cloud.consul.discovery.management-suffix

/** Tags to use when registering management service (defaults to "management" */
spring.cloud.consul.discovery.management-tags
```

9.3.3. HTTP Health Check

The health check for a Consul instance defaults to `/health`, which is the default locations of a useful endpoint in a Spring Boot Actuator application. You need to change these, even for an Actuator application if you use a non-default context path or servlet path (e.g. `server.servletPath=/foo`) or management endpoint path (e.g. `management.server.servlet.context-path=/admin`). The interval that Consul uses to check the health endpoint may also be configured. "10s" and "1m" represent 10 seconds and 1 minute respectively. Example:

application.yml

```
spring:
  cloud:
    consul:
      discovery:
        healthCheckPath: ${management.server.servlet.context-path}/health
        healthCheckInterval: 15s
```

You can disable the health check by setting `management.health.consul.enabled=false`.

Metadata

Consul supports metadata on services. Spring Cloud's `ServiceInstance` has a `Map<String, String> metadata` field which is populated from a services `meta` field. To populate the `meta` field set values on `spring.cloud.consul.discovery.metadata` or `spring.cloud.consul.discovery.management-metadata` properties.

application.yml

```
spring:
  cloud:
    consul:
      discovery:
        metadata:
          myfield: myvalue
          anotherfield: anothervalue
```

The above configuration will result in a service who's meta field contains `myfield→myvalue` and `anotherfield→anothervalue`.

Generated Metadata

The Consul Auto Registration will generate a few entries automatically.

Table 4. Auto Generated Metadata

Key	Value
'group'	Property <code>spring.cloud.consul.discovery.instance-group</code> . This values is only generated if <code>instance-group</code> is not empty.'
'secure'	True if property <code>spring.cloud.consul.discovery.scheme</code> equals 'https', otherwise false.
Property <code>spring.cloud.consul.discovery.default-zone-metadata-name</code> , defaults to 'zone'	Property <code>spring.cloud.consul.discovery.instance-zone</code> . This values is only generated if <code>instance-zone</code> is not empty.'



Older versions of Spring Cloud Consul populated the `ServiceInstance.getMetadata()` method from Spring Cloud Commons by parsing the `spring.cloud.consul.discovery.tags` property. This is no longer supported, please migrate to using the `spring.cloud.consul.discovery.metadata` map.

Making the Consul Instance ID Unique

By default a consul instance is registered with an ID that is equal to its Spring Application Context ID. By default, the Spring Application Context ID is `${spring.application.name}:comma,separated,profiles:${server.port}`. For most cases, this will allow multiple instances of one service to run on one machine. If further uniqueness is required, Using Spring Cloud you can override this by providing a unique identifier in `spring.cloud.consul.discovery.instanceId`. For example:

application.yml

```
spring:
  cloud:
    consul:
      discovery:
        instanceId:
          ${spring.application.name}:${vcap.application.instance_id:${spring.application.instance_id:${random.value}}}
```

With this metadata, and multiple service instances deployed on localhost, the random value will kick in there to make the instance unique. In Cloudfoundry the `vcap.application.instance_id` will be populated automatically in a Spring Boot application, so the random value will not be needed.

Applying Headers to Health Check Requests

Headers can be applied to health check requests. For example, if you're trying to register a [Spring Cloud Config](#) server that uses [Vault Backend](#):

application.yml

```
spring:
  cloud:
    consul:
      discovery:
        health-check-headers:
          X-Config-Token: 6442e58b-d1ea-182e-cfa5-cf9cddef0722
```

According to the HTTP standard, each header can have more than one values, in which case, an array can be supplied:

application.yml

```
spring:
  cloud:
    consul:
      discovery:
        health-check-headers:
          X-Config-Token:
            - "6442e58b-d1ea-182e-cfa5-cf9cddef0722"
            - "Some other value"
```

9.3.4. Looking up services

Using Load-balancer

Spring Cloud has support for [Feign](#) (a REST client builder) and also [Spring RestTemplate](#) for looking up services using the logical service names/ids instead of physical URLs. Both Feign and the discovery-aware RestTemplate utilize [Ribbon](#) for client-side load balancing.

If you want to access service STORES using the RestTemplate simply declare:

```
@LoadBalanced
@Bean
public RestTemplate loadbalancedRestTemplate() {
    return new RestTemplate();
}
```

and use it like this (notice how we use the STORES service name/id from Consul instead of a fully qualified domainname):

```
@Autowired
RestTemplate restTemplate;

public String getFirstProduct() {
    return this.restTemplate.getForObject("https://STORES/products/1", String.class);
}
```

If you have Consul clusters in multiple datacenters and you want to access a service in another datacenter a service name/id alone is not enough. In that case you use property `spring.cloud.consul.discovery.datacenters.STORES=dc-west` where `STORES` is the service name/id and `dc-west` is the datacenter where the STORES service lives.



Spring Cloud now also offers support for [Spring Cloud LoadBalancer](#).

As Spring Cloud Ribbon is now under maintenance, we suggest you set `spring.cloud.loadbalancer.ribbon.enabled` to `false`, so that `BlockingLoadBalancerClient` is used instead of `RibbonLoadBalancerClient`.

Using the DiscoveryClient

You can also use the `org.springframework.cloud.client.discovery.DiscoveryClient` which provides a simple API for discovery clients that is not specific to Netflix, e.g.

```
@Autowired
private DiscoveryClient discoveryClient;

public String serviceUrl() {
    List<ServiceInstance> list = discoveryClient.getInstances("STORES");
    if (list != null && list.size() > 0 ) {
        return list.get(0).getUri();
    }
    return null;
}
```

9.3.5. Consul Catalog Watch

The Consul Catalog Watch takes advantage of the ability of consul to [watch services](#). The Catalog Watch makes a blocking Consul HTTP API call to determine if any services have changed. If there is new service data a Heartbeat Event is published.

To change the frequency of when the Config Watch is called change `spring.cloud.consul.config.discovery.catalog-services-watch-delay`. The default value is 1000, which is in milliseconds. The delay is the amount of time after the end of the previous invocation and the start of the next.

To disable the Catalog Watch set `spring.cloud.consul.discovery.catalogServicesWatch.enabled=false`.

The watch uses a Spring `TaskScheduler` to schedule the call to consul. By default it is a `ThreadPoolTaskScheduler` with a `poolSize` of 1. To change the `TaskScheduler`, create a bean of type `TaskScheduler` named with the `ConsulDiscoveryClientConfiguration.CATALOG_WATCH_TASK_SCHEDULER_NAME` constant.

9.4. Distributed Configuration with Consul

Consul provides a [Key/Value Store](#) for storing configuration and other metadata. Spring Cloud Consul Config is an alternative to the [Config Server and Client](#). Configuration is loaded into the Spring Environment during the special "bootstrap" phase. Configuration is stored in the `/config` folder by default. Multiple `PropertySource` instances are created based on the application's name and the active profiles that mimicks the Spring Cloud Config order of resolving properties. For example, an application with the name "testApp" and with the "dev" profile will have the following property sources created:

```
config/testApp,dev/  
config/testApp/  
config/application,dev/  
config/application/
```

The most specific property source is at the top, with the least specific at the bottom. Properties in the `config/application` folder are applicable to all applications using consul for configuration. Properties in the `config/testApp` folder are only available to the instances of the service named "testApp".

Configuration is currently read on startup of the application. Sending a HTTP POST to `/refresh` will cause the configuration to be reloaded. [Config Watch](#) will also automatically detect changes and reload the application context.

9.4.1. How to activate

To get started with Consul Configuration use the starter with group `org.springframework.cloud` and artifact id `spring-cloud-starter-consul-config`. See the [Spring Cloud Project page](#) for details on setting up your build system with the current Spring Cloud Release Train.

This will enable auto-configuration that will setup Spring Cloud Consul Config.

9.4.2. Customizing

Consul Config may be customized using the following properties:

bootstrap.yml

```
spring:
  cloud:
    consul:
      config:
        enabled: true
        prefix: configuration
        defaultContext: apps
        profileSeparator: '::'
```

- **enabled** setting this value to "false" disables Consul Config
- **prefix** sets the base folder for configuration values
- **defaultContext** sets the folder name used by all applications
- **profileSeparator** sets the value of the separator used to separate the profile name in property sources with profiles

9.4.3. Config Watch

The Consul Config Watch takes advantage of the ability of consul to [watch a key prefix](#). The Config Watch makes a blocking Consul HTTP API call to determine if any relevant configuration data has changed for the current application. If there is new configuration data a Refresh Event is published. This is equivalent to calling the `/refresh` actuator endpoint.

To change the frequency of when the Config Watch is called change `spring.cloud.consul.config.watch.delay`. The default value is 1000, which is in milliseconds. The delay is the amount of time after the end of the previous invocation and the start of the next.

To disable the Config Watch set `spring.cloud.consul.config.watch.enabled=false`.

The watch uses a Spring `TaskScheduler` to schedule the call to consul. By default it is a `ThreadPoolTaskScheduler` with a `poolSize` of 1. To change the `TaskScheduler`, create a bean of type `TaskScheduler` named with the `ConsulConfigAutoConfiguration.CONFIG_WATCH_TASK_SCHEDULER_NAME` constant.

9.4.4. YAML or Properties with Config

It may be more convenient to store a blob of properties in YAML or Properties format as opposed to individual key/value pairs. Set the `spring.cloud.consul.config.format` property to `YAML` or `PROPERTIES`. For example to use YAML:

bootstrap.yml

```
spring:
  cloud:
    consul:
      config:
        format: YAML
```

YAML must be set in the appropriate `data` key in consul. Using the defaults above the keys would look like:

```
config/testApp,dev/data
config/testApp/data
config/application,dev/data
config/application/data
```

You could store a YAML document in any of the keys listed above.

You can change the data key using `spring.cloud.consul.config.data-key`.

9.4.5. git2consul with Config

git2consul is a Consul community project that loads files from a git repository to individual keys into Consul. By default the names of the keys are names of the files. YAML and Properties files are supported with file extensions of `.yaml` and `.properties` respectively. Set the `spring.cloud.consul.config.format` property to `FILES`. For example:

bootstrap.yml

```
spring:
  cloud:
    consul:
      config:
        format: FILES
```

Given the following keys in `/config`, the `development` profile and an application name of `foo`:

```
.gitignore
application.yml
bar.properties
foo-development.properties
foo-production.yml
foo.properties
master.ref
```

the following property sources would be created:

```
config/foo-development.properties
config/foo.properties
config/application.yml
```

The value of each key needs to be a properly formatted YAML or Properties file.

9.4.6. Fail Fast

It may be convenient in certain circumstances (like local development or certain test scenarios) to not fail if consul isn't available for configuration. Setting `spring.cloud.consul.config.failFast=false` in `bootstrap.yml` will cause the configuration module to log a warning rather than throw an exception. This will allow the application to continue startup normally.

9.5. Consul Retry

If you expect that the consul agent may occasionally be unavailable when your app starts, you can ask it to keep trying after a failure. You need to add `spring-retry` and `spring-boot-starter-aop` to your classpath. The default behaviour is to retry 6 times with an initial backoff interval of 1000ms and an exponential multiplier of 1.1 for subsequent backoffs. You can configure these properties (and others) using `spring.cloud.consul.retry.*` configuration properties. This works with both Spring Cloud Consul Config and Discovery registration.



To take full control of the retry add a `@Bean` of type `RetryOperationsInterceptor` with id "consulRetryInterceptor". Spring Retry has a `RetryInterceptorBuilder` that makes it easy to create one.

9.6. Spring Cloud Bus with Consul

9.6.1. How to activate

To get started with the Consul Bus use the starter with group `org.springframework.cloud` and artifact id `spring-cloud-starter-consul-bus`. See the [Spring Cloud Project page](#) for details on setting up your build system with the current Spring Cloud Release Train.

See the [Spring Cloud Bus](#) documentation for the available actuator endpoints and howto send custom messages.

9.7. Circuit Breaker with Hystrix

Applications can use the Hystrix Circuit Breaker provided by the Spring Cloud Netflix project by including this starter in the projects pom.xml: `spring-cloud-starter-hystrix`. Hystrix doesn't depend on the Netflix Discovery Client. The `@EnableHystrix` annotation should be placed on a configuration class (usually the main class). Then methods can be annotated with `@HystrixCommand` to be protected by a circuit breaker. See [the documentation](#) for more details.

9.8. Hystrix metrics aggregation with Turbine and Consul

Turbine (provided by the Spring Cloud Netflix project), aggregates multiple instances Hystrix metrics streams, so the dashboard can display an aggregate view. Turbine uses the `DiscoveryClient` interface to lookup relevant instances. To use Turbine with Spring Cloud Consul, configure the Turbine application in a manner similar to the following examples:

pom.xml

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-netflix-turbine</artifactId>
</dependency>
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-consul-discovery</artifactId>
</dependency>
```

Notice that the Turbine dependency is not a starter. The turbine starter includes support for Netflix Eureka.

application.yml

```
spring.application.name: turbine
applications: consulhystrixclient
turbine:
  aggregator:
    clusterConfig: ${applications}
    appConfig: ${applications}
```

The `clusterConfig` and `appConfig` sections must match, so it's useful to put the comma-separated list of service ID's into a separate configuration property.

Turbine.java

```
@EnableTurbine
@SpringBootApplication
public class Turbine {
  public static void main(String[] args) {
    SpringApplication.run(DemoturbinecommonsApplication.class, args);
  }
}
```

9.9. Configuration Properties

To see the list of all Consul related configuration properties please check [the Appendix page](#).

Chapter 10. Spring Cloud Zookeeper

This project provides Zookeeper integrations for Spring Boot applications through autoconfiguration and binding to the Spring Environment and other Spring programming model idioms. With a few annotations, you can quickly enable and configure the common patterns inside your application and build large distributed systems with Zookeeper based components. The provided patterns include Service Discovery and Configuration. The project also provides client-side load-balancing via integration with Spring Cloud LoadBalancer.

10.1. Install Zookeeper

See the [installation documentation](#) for instructions on how to install Zookeeper.

Spring Cloud Zookeeper uses Apache Curator behind the scenes. While Zookeeper 3.5.x is still considered "beta" by the Zookeeper development team, the reality is that it is used in production by many users. However, Zookeeper 3.4.x is also used in production. Prior to Apache Curator 4.0, both versions of Zookeeper were supported via two versions of Apache Curator. Starting with Curator 4.0 both versions of Zookeeper are supported via the same Curator libraries.

In case you are integrating with version 3.4 you need to change the Zookeeper dependency that comes shipped with `curator`, and thus `spring-cloud-zookeeper`. To do so simply exclude that dependency and add the 3.4.x version like shown below.

maven

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-zookeeper-all</artifactId>
  <exclusions>
    <exclusion>
      <groupId>org.apache.zookeeper</groupId>
      <artifactId>zookeeper</artifactId>
    </exclusion>
  </exclusions>
</dependency>
<dependency>
  <groupId>org.apache.zookeeper</groupId>
  <artifactId>zookeeper</artifactId>
  <version>3.4.12</version>
  <exclusions>
    <exclusion>
      <groupId>org.slf4j</groupId>
      <artifactId>slf4j-log4j12</artifactId>
    </exclusion>
  </exclusions>
</dependency>
```

gradle

```
compile('org.springframework.cloud:spring-cloud-starter-zookeeper-all') {
    exclude group: 'org.apache.zookeeper', module: 'zookeeper'
}
compile('org.apache.zookeeper:zookeeper:3.4.12') {
    exclude group: 'org.slf4j', module: 'slf4j-log4j12'
}
```

10.2. Service Discovery with Zookeeper

Service Discovery is one of the key tenets of a microservice based architecture. Trying to hand-configure each client or some form of convention can be difficult to do and can be brittle. [Curator](#) (A Java library for Zookeeper) provides Service Discovery through a [Service Discovery Extension](#). Spring Cloud Zookeeper uses this extension for service registration and discovery.

10.2.1. Activating

Including a dependency on `org.springframework.cloud:spring-cloud-starter-zookeeper-discovery` enables autoconfiguration that sets up Spring Cloud Zookeeper Discovery.



For web functionality, you still need to include `org.springframework.boot:spring-boot-starter-web`.



When working with version 3.4 of Zookeeper you need to change the way you include the dependency as described [here](#).

10.2.2. Registering with Zookeeper

When a client registers with Zookeeper, it provides metadata (such as host and port, ID, and name) about itself.

The following example shows a Zookeeper client:

```

@SpringBootApplication
@RestController
public class Application {

    @RequestMapping("/")
    public String home() {
        return "Hello world";
    }

    public static void main(String[] args) {
        new SpringApplicationBuilder(Application.class).web(true).run(args);
    }
}

```



The preceding example is a normal Spring Boot application.

If Zookeeper is located somewhere other than `localhost:2181`, the configuration must provide the location of the server, as shown in the following example:

application.yml

```

spring:
  cloud:
    zookeeper:
      connect-string: localhost:2181

```



If you use [Spring Cloud Zookeeper Config](#), the values shown in the preceding example need to be in `bootstrap.yml` instead of `application.yml`.

The default service name, instance ID, and port (taken from the `Environment`) are `spring.application.name`, the Spring Context ID, and `server.port`, respectively.

Having `spring-cloud-starter-zookeeper-discovery` on the classpath makes the app into both a Zookeeper “service” (that is, it registers itself) and a “client” (that is, it can query Zookeeper to locate other services).

If you would like to disable the Zookeeper Discovery Client, you can set `spring.cloud.zookeeper.discovery.enabled` to `false`.

10.2.3. Using the `DiscoveryClient`

Spring Cloud has support for [Feign](#) (a REST client builder), [Spring RestTemplate](#) and [Spring WebFlux](#), using logical service names instead of physical URLs.

You can also use the `org.springframework.cloud.client.discovery.DiscoveryClient`, which provides a simple API for discovery clients that is not specific to Netflix, as shown in the following example:

```

@Autowired
private DiscoveryClient discoveryClient;

public String serviceUrl() {
    List<ServiceInstance> list = discoveryClient.getInstances("STORES");
    if (list != null && list.size() > 0 ) {
        return list.get(0).getUri().toString();
    }
    return null;
}

```

10.3. Using Spring Cloud Zookeeper with Spring Cloud Netflix Components

Spring Cloud Netflix supplies useful tools that work regardless of which `DiscoveryClient` implementation you use. Feign and Spring Cloud LoadBalancer also work with Spring Cloud Zookeeper.

10.3.1. Spring Cloud LoadBalancer with Zookeeper

Spring Cloud Zookeeper provides an implementation of Spring Cloud LoadBalancer `ServiceInstanceListSupplier`. When you use the `spring-cloud-starter-zookeeper-discovery`, Spring Cloud LoadBalancer is autoconfigured to use the `ZookeeperServiceInstanceListSupplier` by default.

10.4. Spring Cloud Zookeeper and Service Registry

Spring Cloud Zookeeper implements the `ServiceRegistry` interface, letting developers register arbitrary services in a programmatic way.

The `ServiceInstanceRegistration` class offers a `builder()` method to create a `Registration` object that can be used by the `ServiceRegistry`, as shown in the following example:

```

@Autowired
private ZookeeperServiceRegistry serviceRegistry;

public void registerThings() {
    ZookeeperRegistration registration = ServiceInstanceRegistration.builder()
        .defaultUriSpec()
        .address("anyUrl")
        .port(10)
        .name("/a/b/c/d/anotherService")
        .build();
    this.serviceRegistry.register(registration);
}

```

10.4.1. Instance Status

Netflix Eureka supports having instances that are `OUT_OF_SERVICE` registered with the server. These instances are not returned as active service instances. This is useful for behaviors such as blue/green deployments. (Note that the Curator Service Discovery recipe does not support this behavior.) Taking advantage of the flexible payload has let Spring Cloud Zookeeper implement `OUT_OF_SERVICE` by updating some specific metadata and then filtering on that metadata in the Spring Cloud LoadBalancer `ZookeeperServiceInstanceListSupplier`. The `ZookeeperServiceInstanceListSupplier` filters out all non-null instance statuses that do not equal `UP`. If the instance status field is empty, it is considered to be `UP` for backwards compatibility. To change the status of an instance, make a `POST` with `OUT_OF_SERVICE` to the `ServiceRegistry` instance status actuator endpoint, as shown in the following example:

```
$ http POST http://localhost:8081/service-registry status=OUT_OF_SERVICE
```



The preceding example uses the `http` command from httpie.org.

10.5. Zookeeper Dependencies

The following topics cover how to work with Spring Cloud Zookeeper dependencies:

- [Using the Zookeeper Dependencies](#)
- [Activating Zookeeper Dependencies](#)
- [Setting up Zookeeper Dependencies](#)
- [Configuring Spring Cloud Zookeeper Dependencies](#)

10.5.1. Using the Zookeeper Dependencies

Spring Cloud Zookeeper gives you a possibility to provide dependencies of your application as properties. As dependencies, you can understand other applications that are registered in Zookeeper and which you would like to call through [Feign](#) (a REST client builder), [Spring RestTemplate](#) and [Spring WebFlux](#).

You can also use the Zookeeper Dependency Watchers functionality to control and monitor the state of your dependencies.

10.5.2. Activating Zookeeper Dependencies

Including a dependency on `org.springframework.cloud:spring-cloud-starter-zookeeper-discovery` enables autoconfiguration that sets up Spring Cloud Zookeeper Dependencies. Even if you provide the dependencies in your properties, you can turn off the dependencies. To do so, set the `spring.cloud.zookeeper.dependency.enabled` property to `false` (it defaults to `true`).

10.5.3. Setting up Zookeeper Dependencies

Consider the following example of dependency representation:

application.yml

```
spring.application.name: yourServiceName
spring.cloud.zookeeper:
  dependencies:
    newsletter:
      path: /path/where/newsletter/has/registered/in/zookeeper
      loadBalancerType: ROUND_ROBIN
      contentTypeTemplate: application/vnd.newsletter.$version+json
      version: v1
      headers:
        header1:
          - value1
        header2:
          - value2
      required: false
      stubs: org.springframework:foo:stubs
    mailing:
      path: /path/where/mailing/has/registered/in/zookeeper
      loadBalancerType: ROUND_ROBIN
      contentTypeTemplate: application/vnd.mailing.$version+json
      version: v1
      required: true
```

The next few sections go through each part of the dependency one by one. The root property name is `spring.cloud.zookeeper.dependencies`.

Aliases

Below the root property you have to represent each dependency as an alias. This is due to the constraints of Spring Cloud LoadBalancer, which requires that the application ID be placed in the URL. Consequently, you cannot pass any complex path, such as `/myApp/myRoute/name`). The alias is the name you use instead of the `serviceId` for `DiscoveryClient`, `Feign`, or `RestTemplate`.

In the previous examples, the aliases are `newsletter` and `mailing`. The following example shows Feign usage with a `newsletter` alias:

```
@FeignClient("newsletter")
public interface NewsletterService {
    @RequestMapping(method = RequestMethod.GET, value = "/newsletter")
    String getNewsletters();
}
```

Path

The path is represented by the `path` YAML property and is the path under which the dependency is registered under Zookeeper. As described in the [previous section](#), Spring Cloud LoadBalancer operates on URLs. As a result, this path is not compliant with its requirement. That is why Spring Cloud Zookeeper maps the alias to the proper path.

Load Balancer Type

The load balancer type is represented by `loadBalancerType` YAML property.

If you know what kind of load-balancing strategy has to be applied when calling this particular dependency, you can provide it in the YAML file, and it is automatically applied. You can choose one of the following load balancing strategies:

- **STICKY**: Once chosen, the instance is always called.
- **RANDOM**: Picks an instance randomly.
- **ROUND_ROBIN**: Iterates over instances over and over again.

Content-Type Template and Version

The `Content-Type` template and version are represented by the `contentTypeTemplate` and `version` YAML properties.

If you version your API in the `Content-Type` header, you do not want to add this header to each of your requests. Also, if you want to call a new version of the API, you do not want to roam around your code to bump up the API version. That is why you can provide a `contentTypeTemplate` with a special `$version` placeholder. That placeholder will be filled by the value of the `version` YAML property. Consider the following example of a `contentTypeTemplate`:

```
application/vnd.newsletter.$version+json
```

Further consider the following `version`:

```
v1
```

The combination of `contentTypeTemplate` and `version` results in the creation of a `Content-Type` header for each request, as follows:

```
application/vnd.newsletter.v1+json
```

Default Headers

Default headers are represented by the `headers` map in YAML.

Sometimes, each call to a dependency requires setting up of some default headers. To not do that in code, you can set them up in the YAML file, as shown in the following example `headers` section:


```
headers:  
  Accept:  
    - text/html  
    - application/xhtml+xml  
  Cache-Control:  
    - no-cache
```

That `headers` section results in adding the `Accept` and `Cache-Control` headers with appropriate list of values in your HTTP request.

Required Dependencies

Required dependencies are represented by `required` property in YAML.

If one of your dependencies is required to be up when your application boots, you can set the `required: true` property in the YAML file.

If your application cannot localize the required dependency during boot time, it throws an exception, and the Spring Context fails to set up. In other words, your application cannot start if the required dependency is not registered in Zookeeper.

You can read more about Spring Cloud Zookeeper Presence Checker [later in this document](#).

Stubs

You can provide a colon-separated path to the JAR containing stubs of the dependency, as shown in the following example:

```
stubs: org.springframework:myApp:stubs
```

where:

- `org.springframework` is the `groupId`.
- `myApp` is the `artifactId`.
- `stubs` is the classifier. (Note that `stubs` is the default value.)

Because `stubs` is the default classifier, the preceding example is equal to the following example:

```
stubs: org.springframework:myApp
```

10.5.4. Configuring Spring Cloud Zookeeper Dependencies

You can set the following properties to enable or disable parts of Zookeeper Dependencies functionalities:

- `spring.cloud.zookeeper.dependencies`: If you do not set this property, you cannot use Zookeeper Dependencies.
- `spring.cloud.zookeeper.dependency.loadbalancer.enabled` (enabled by default): Turns on Zookeeper-specific custom load-balancing strategies, including

`ZookeeperServiceInstanceListSupplier` and dependency-based load-balanced `RestTemplate` setup.

- `spring.cloud.zookeeper.dependency.headers.enabled` (enabled by default): This property registers a `FeignBlockingLoadBalancerClient` that automatically appends appropriate headers and content types with their versions, as presented in the Dependency configuration. Without this setting, those two parameters do not work.
- `spring.cloud.zookeeper.dependency.resttemplate.enabled` (enabled by default): When enabled, this property modifies the request headers of a `@LoadBalanced`-annotated `RestTemplate` such that it passes headers and content type with the version set in dependency configuration. Without this setting, those two parameters do not work.

10.6. Spring Cloud Zookeeper Dependency Watcher

The Dependency Watcher mechanism lets you register listeners to your dependencies. The functionality is, in fact, an implementation of the `Observer` pattern. When a dependency changes, its state (to either UP or DOWN), some custom logic can be applied.

10.6.1. Activating

Spring Cloud Zookeeper Dependencies functionality needs to be enabled for you to use the Dependency Watcher mechanism.

10.6.2. Registering a Listener

To register a listener, you must implement an interface called `org.springframework.cloud.zookeeper.discovery.watcher.DependencyWatcherListener` and register it as a bean. The interface gives you one method:

```
void stateChanged(String dependencyName, DependencyState newState);
```

If you want to register a listener for a particular dependency, the `dependencyName` would be the discriminator for your concrete implementation. `newState` provides you with information about whether your dependency has changed to `CONNECTED` or `DISCONNECTED`.

10.6.3. Using the Presence Checker

Bound with the Dependency Watcher is the functionality called Presence Checker. It lets you provide custom behavior when your application boots, to react according to the state of your dependencies.

The default implementation of the abstract `org.springframework.cloud.zookeeper.discovery.watcher.presence.DependencyPresenceOnStartupVerifier` class is the `org.springframework.cloud.zookeeper.discovery.watcher.presence.DefaultDependencyPresenceOnStartupVerifier`, which works in the following way.

1. If the dependency is marked as `required` and is not in Zookeeper, when your application boots, it throws an exception and shuts down.

2. If the dependency is not required, the `org.springframework.cloud.zookeeper.discovery.watcher.presence.LogMissingDependencyChecker` logs that the dependency is missing at the **WARN** level.

Because the `DefaultDependencyPresenceOnStartupVerifier` is registered only when there is no bean of type `DependencyPresenceOnStartupVerifier`, this functionality can be overridden.

10.7. Distributed Configuration with Zookeeper

Zookeeper provides a [hierarchical namespace](#) that lets clients store arbitrary data, such as configuration data. Spring Cloud Zookeeper Config is an alternative to the [Config Server and Client](#). Configuration is loaded into the Spring Environment during the special “bootstrap” phase. Configuration is stored in the `/config` namespace by default. Multiple `PropertySource` instances are created, based on the application’s name and the active profiles, to mimic the Spring Cloud Config order of resolving properties. For example, an application with a name of `testApp` and with the `dev` profile has the following property sources created for it:

- `config/testApp,dev`
- `config/testApp`
- `config/application,dev`
- `config/application`

The most specific property source is at the top, with the least specific at the bottom. Properties in the `config/application` namespace apply to all applications that use zookeeper for configuration. Properties in the `config/testApp` namespace are available only to the instances of the service named `testApp`.

Configuration is currently read on startup of the application. Sending a HTTP **POST** request to `/refresh` causes the configuration to be reloaded. Watching the configuration namespace (which Zookeeper supports) is not currently implemented.

10.7.1. Activating

Including a dependency on `org.springframework.cloud:spring-cloud-starter-zookeeper-config` enables autoconfiguration that sets up Spring Cloud Zookeeper Config.



When working with version 3.4 of Zookeeper you need to change the way you include the dependency as described [here](#).

10.7.2. Customizing

Zookeeper Config may be customized by setting the following properties:

bootstrap.yml

```
spring:
  cloud:
    zookeeper:
      config:
        enabled: true
        root: configuration
        defaultContext: apps
        profileSeparator: '::'
```

- **enabled**: Setting this value to **false** disables Zookeeper Config.
- **root**: Sets the base namespace for configuration values.
- **defaultContext**: Sets the name used by all applications.
- **profileSeparator**: Sets the value of the separator used to separate the profile name in property sources with profiles.

10.7.3. Access Control Lists (ACLs)

You can add authentication information for Zookeeper ACLs by calling the **addAuthInfo** method of a **CuratorFramework** bean. One way to accomplish this is to provide your own **CuratorFramework** bean, as shown in the following example:

```
@BootstrapConfiguration
public class CustomCuratorFrameworkConfig {

    @Bean
    public CuratorFramework curatorFramework() {
        CuratorFramework curator = new CuratorFramework();
        curator.addAuthInfo("digest", "user:password".getBytes());
        return curator;
    }
}
```

Consult [the ZookeeperAutoConfiguration class](#) to see how the **CuratorFramework** bean's default configuration.

Alternatively, you can add your credentials from a class that depends on the existing **CuratorFramework** bean, as shown in the following example:

```
@BootstrapConfiguration
public class DefaultCuratorFrameworkConfig {

    public ZookeeperConfig(CuratorFramework curator) {
        curator.addAuthInfo("digest", "user:password".getBytes());
    }

}
```

The creation of this bean must occur during the bootstrapping phase. You can register configuration classes to run during this phase by annotating them with `@BootstrapConfiguration` and including them in a comma-separated list that you set as the value of the `org.springframework.cloud.bootstrap.BootstrapConfiguration` property in the `resources/META-INF/spring.factories` file, as shown in the following example:

resources/META-INF/spring.factories

```
org.springframework.cloud.bootstrap.BootstrapConfiguration=\
my.project.CustomCuratorFrameworkConfig,\
my.project.DefaultCuratorFrameworkConfig
```

Chapter 11. Spring Boot Cloud CLI

Spring Boot CLI provides [Spring Boot](#) command line features for [Spring Cloud](#). You can write Groovy scripts to run Spring Cloud component applications (e.g. `@EnableEurekaServer`). You can also easily do things like encryption and decryption to support Spring Cloud Config clients with secret configuration values. With the Launcher CLI you can launch services like Eureka, Zipkin, Config Server conveniently all at once from the command line (very useful at development time).



Spring Cloud is released under the non-restrictive Apache 2.0 license. If you would like to contribute to this section of the documentation or if you find an error, please find the source code and issue trackers in the project at [github](#).

11.1. Installation

To install, make sure you have [Spring Boot CLI](#) (2.0.0 or better):

```
$ spring version
Spring CLI v2.2.0.BUILD-SNAPSHOT
```

E.g. for SDKMan users

```
$ sdk install springboot 2.2.0.BUILD-SNAPSHOT
$ sdk use springboot 2.2.0.BUILD-SNAPSHOT
```

and install the Spring Cloud plugin

```
$ mvn install
$ spring install org.springframework.cloud:spring-cloud-cli:2.2.0.BUILD-SNAPSHOT
```



Prerequisites: to use the encryption and decryption features you need the full-strength JCE installed in your JVM (it's not there by default). You can download the "Java Cryptography Extension (JCE) Unlimited Strength Jurisdiction Policy Files" from Oracle, and follow instructions for installation (essentially replace the 2 policy files in the JRE lib/security directory with the ones that you downloaded).

11.2. Running Spring Cloud Services in Development

The Launcher CLI can be used to run common services like Eureka, Config Server etc. from the command line. To list the available services you can do `spring cloud --list`, and to launch a default set of services just `spring cloud`. To choose the services to deploy, just list them on the command line, e.g.

```
$ spring cloud eureka configserver h2 kafka stubrunner zipkin
```

Summary of supported deployables:

Service	Name	Address	Description
eureka	Eureka Server	localhost:8761	Eureka server for service registration and discovery. All the other services show up in its catalog by default.
configserver	Config Server	localhost:8888	Spring Cloud Config Server running in the "native" profile and serving configuration from the local directory ./launcher
h2	H2 Database	localhost:9095 (console), jdbc:h2:tcp://localhost:9096/{data}	Relation database service. Use a file path for {data} (e.g. ./target/test) when you connect. Remember that you can add ;MODE=MYSQL or ;MODE=POSTGRESQL to connect with compatibility to other server types.
kafka	Kafka Broker	localhost:9091 (actuator endpoints), localhost:9092	
dataflow	Dataflow Server	localhost:9393	Spring Cloud Dataflow server with UI at /admin-ui. Connect the Dataflow shell to target at root path.
zipkin	Zipkin Server	localhost:9411	Zipkin Server with UI for visualizing traces. Stores span data in memory and accepts them via HTTP POST of JSON data.

Service	Name	Address	Description
stubrunner	Stub Runner Boot	localhost:8750	Downloads WireMock stubs, starts WireMock and feeds the started servers with stored stubs. Pass <code>stubrunner.ids</code> to pass stub coordinates and then go to localhost:8750/stubs .

Each of these apps can be configured using a local YAML file with the same name (in the current working directory or a subdirectory called "config" or in `~/.spring-cloud`). E.g. in `configserver.yml` you might want to do something like this to locate a local git repository for the backend:

configserver.yml

```
spring:
  profiles:
    active: git
  cloud:
    config:
      server:
        git:
          uri: file://${user.home}/dev/demo/config-repo
```

E.g. in Stub Runner app you could fetch stubs from your local `.m2` in the following way.

stubrunner.yml

```
stubrunner:
  workOffline: true
  ids:
    - com.example:beer-api-producer:+:9876
```

11.2.1. Adding Additional Applications

Additional applications can be added to `./config/cloud.yml` (not `./config.yml` because that would replace the defaults), e.g. with

config/cloud.yml

```
spring:
  cloud:
    launcher:
      deployables:
        source:
          coordinates: maven://com.example:source:0.0.1-SNAPSHOT
          port: 7000
        sink:
          coordinates: maven://com.example:sink:0.0.1-SNAPSHOT
          port: 7001
```

when you list the apps:

```
$ spring cloud --list
source sink configserver dataflow eureka h2 kafka stubrunner zipkin
```

(notice the additional apps at the start of the list).

11.3. Writing Groovy Scripts and Running Applications

Spring Cloud CLI has support for most of the Spring Cloud declarative features, such as the `@Enable*` class of annotations. For example, here is a fully functional Eureka server

app.groovy

```
@EnableEurekaServer
class Eureka {}
```

which you can run from the command line like this

```
$ spring run app.groovy
```

To include additional dependencies, often it suffices just to add the appropriate feature-enabling annotation, e.g. `@EnableConfigServer`, `@EnableOAuth2Sso` or `@EnableEurekaClient`. To manually include a dependency you can use a `@Grab` with the special "Spring Boot" short style artifact co-ordinates, i.e. with just the artifact ID (no need for group or version information), e.g. to set up a client app to listen on AMQP for management events from the Spring Cloud Bus:

app.groovy

```
@Grab('spring-cloud-starter-bus-amqp')
@RestController
class Service {
    @RequestMapping('/')
    def home() { [message: 'Hello'] }
}
```

11.4. Encryption and Decryption

The Spring Cloud CLI comes with an "encrypt" and a "decrypt" command. Both accept arguments in the same form with a key specified as a mandatory "--key", e.g.

```
$ spring encrypt mysecret --key foo
682bc583f4641835fa2db009355293665d2647dade3375c0ee201de2a49f7bda
$ spring decrypt --key foo
682bc583f4641835fa2db009355293665d2647dade3375c0ee201de2a49f7bda
mysecret
```

To use a key in a file (e.g. an RSA public key for encryption) prepend the key value with "@" and provide the file path, e.g.

```
$ spring encrypt mysecret --key @${HOME}/.ssh/id_rsa.pub
AQAJpGt3eFZQXwt8tsHAVv/QHiY5sI2dRcR+...
```

Chapter 12. Spring Cloud Security

Spring Cloud Security offers a set of primitives for building secure applications and services with minimum fuss. A declarative model which can be heavily configured externally (or centrally) lends itself to the implementation of large systems of co-operating, remote components, usually with a central identity management service. It is also extremely easy to use in a service platform like Cloud Foundry. Building on Spring Boot and Spring Security OAuth2 we can quickly create systems that implement common patterns like single sign on, token relay and token exchange.



Spring Cloud is released under the non-restrictive Apache 2.0 license. If you would like to contribute to this section of the documentation or if you find an error, please find the source code and issue trackers in the project at [github](#).

12.1. Quickstart

12.1.1. OAuth2 Single Sign On

Here's a Spring Cloud "Hello World" app with HTTP Basic authentication and a single user account:

app.groovy

```
@Grab('spring-boot-starter-security')
@Controller
class Application {

    @RequestMapping('/')
    String home() {
        'Hello World'
    }
}
```

You can run it with `spring run app.groovy` and watch the logs for the password (username is "user"). So far this is just the default for a Spring Boot app.

Here's a Spring Cloud app with OAuth2 SSO:

app.groovy

```
@Controller
@EnableOAuth2Sso
class Application {

    @RequestMapping('/')
    String home() {
        'Hello World'
    }
}
```

Spot the difference? This app will actually behave exactly the same as the previous one, because it doesn't know it's OAuth2 credentials yet.

You can register an app in github quite easily, so try that if you want a production app on your own domain. If you are happy to test on localhost:8080, then set up these properties in your application configuration:

application.yml

```
security:
  oauth2:
    client:
      clientId: bd1c0a783ccdd1c9b9e4
      clientSecret: 1a9030fbca47a5b2c28e92f19050bb77824b5ad1
      accessTokenUri: https://github.com/login/oauth/access_token
      userAuthorizationUri: https://github.com/login/oauth/authorize
      clientAuthenticationScheme: form
    resource:
      userInfoUri: https://api.github.com/user
      preferTokenInfo: false
```

run the app above and it will redirect to github for authorization. If you are already signed into github you won't even notice that it has authenticated. These credentials will only work if your app is running on port 8080.

To limit the scope that the client asks for when it obtains an access token you can set `security.oauth2.client.scope` (comma separated or an array in YAML). By default the scope is empty and it is up to to Authorization Server to decide what the defaults should be, usually depending on the settings in the client registration that it holds.



The examples above are all Groovy scripts. If you want to write the same code in Java (or Groovy) you need to add Spring Security OAuth2 to the classpath (e.g. see the [sample here](#)).

12.1.2. OAuth2 Protected Resource

You want to protect an API resource with an OAuth2 token? Here's a simple example (paired with the client above):

app.groovy

```
@Grab('spring-cloud-starter-security')
@RestController
@EnableResourceServer
class Application {

    @RequestMapping('/')
    def home() {
        [message: 'Hello World']
    }
}
```

and

application.yml

```
security:
  oauth2:
    resource:
      userInfoUri: https://api.github.com/user
      preferTokenInfo: false
```

12.2. More Detail

12.2.1. Single Sign On



All of the OAuth2 SSO and resource server features moved to Spring Boot in version 1.3. You can find documentation in the [Spring Boot user guide](#).

12.2.2. Token Relay

A Token Relay is where an OAuth2 consumer acts as a Client and forwards the incoming token to outgoing resource requests. The consumer can be a pure Client (like an SSO application) or a Resource Server.

Client Token Relay in Spring Cloud Gateway

If your app also has a [Spring Cloud Gateway](#) embedded reverse proxy then you can ask it to forward OAuth2 access tokens downstream to the services it is proxying. Thus the SSO app above can be enhanced simply like this:

App.java

```
@Autowired
private TokenRelayGatewayFilterFactory filterFactory;

@Bean
public RouteLocator customRouteLocator(RouteLocatorBuilder builder) {
    return builder.routes()
        .route("resource", r -> r.path("/resource")
            .filters(f -> f.filter(filterFactory.apply()))
            .uri("http://localhost:9000"))
        .build();
}
```

or this

application.yaml

```
spring:
  cloud:
    gateway:
      routes:
        - id: resource
          uri: http://localhost:9000
          predicates:
            - Path=/resource
          filters:
            - TokenRelay=
```

and it will (in addition to logging the user in and grabbing a token) pass the authentication token downstream to the services (in this case */resource*).

To enable this for Spring Cloud Gateway add the following dependencies

- `org.springframework.boot:spring-boot-starter-oauth2-client`
- `org.springframework.cloud:spring-cloud-starter-security`

How does it work? The [filter](#) extracts an access token from the currently authenticated user, and puts it in a request header for the downstream requests.

For a full working sample see [this project](#).



The default implementation of `ReactiveOAuth2AuthorizedClientService` used by `TokenRelayGatewayFilterFactory` uses an in-memory data store. You will need to provide your own implementation `ReactiveOAuth2AuthorizedClientService` if you need a more robust solution.

Client Token Relay

If your app is a user facing OAuth2 client (i.e. has declared `@EnableOAuth2Sso` or `@EnableOAuth2Client`) then it has an `OAuth2ClientContext` in request scope from Spring Boot. You can create your own `OAuth2RestTemplate` from this context and an autowired `OAuth2ProtectedResourceDetails`, and then the context will always forward the access token downstream, also refreshing the access token automatically if it expires. (These are features of Spring Security and Spring Boot.)



Spring Boot (1.4.1) does not create an `OAuth2ProtectedResourceDetails` automatically if you are using `client_credentials` tokens. In that case you need to create your own `ClientCredentialsResourceDetails` and configure it with `@ConfigurationProperties("security.oauth2.client")`.

Client Token Relay in Zuul Proxy

If your app also has a [Spring Cloud Zuul](#) embedded reverse proxy (using `@EnableZuulProxy`) then you can ask it to forward OAuth2 access tokens downstream to the services it is proxying. Thus the SSO app above can be enhanced simply like this:

app.groovy

```
@Controller
@EnableOAuth2Sso
@EnableZuulProxy
class Application {

}
```

and it will (in addition to logging the user in and grabbing a token) pass the authentication token downstream to the `/proxy/*` services. If those services are implemented with `@EnableResourceServer` then they will get a valid token in the correct header.

How does it work? The `@EnableOAuth2Sso` annotation pulls in `spring-cloud-starter-security` (which you could do manually in a traditional app), and that in turn triggers some autoconfiguration for a `ZuulFilter`, which itself is activated because Zuul is on the classpath (via `@EnableZuulProxy`). The `filter` just extracts an access token from the currently authenticated user, and puts it in a request header for the downstream requests.



Spring Boot does not create an `OAuth2RestOperations` automatically which is needed for `refresh_token`. In that case you need to create your own `OAuth2RestOperations` so `OAuth2TokenRelayFilter` can refresh the token if needed.

Resource Server Token Relay

If your app has `@EnableResourceServer` you might want to relay the incoming token downstream to other services. If you use a `RestTemplate` to contact the downstream services then this is just a matter of how to create the template with the right context.

If your service uses `UserInfoTokenServices` to authenticate incoming tokens (i.e. it is using the

`security.oauth2.user-info-uri` configuration), then you can simply create an `OAuth2RestTemplate` using an autowired `OAuth2ClientContext` (it will be populated by the authentication process before it hits the backend code). Equivalently (with Spring Boot 1.4), you could inject a `UserInfoRestTemplateFactory` and grab its `OAuth2RestTemplate` in your configuration. For example:

MyConfiguration.java

```
@Bean
public OAuth2RestTemplate restTemplate(UserInfoRestTemplateFactory factory) {
    return factory.getUserInfoRestTemplate();
}
```

This rest template will then have the same `OAuth2ClientContext` (request-scoped) that is used by the authentication filter, so you can use it to send requests with the same access token.

If your app is not using `UserInfoTokenServices` but is still a client (i.e. it declares `@EnableOAuth2Client` or `@EnableOAuth2Sso`), then with Spring Security Cloud any `OAuth2RestOperations` that the user creates from an `@Autowired OAuth2Context` will also forward tokens. This feature is implemented by default as an MVC handler interceptor, so it only works in Spring MVC. If you are not using MVC you could use a custom filter or AOP interceptor wrapping an `AccessTokenContextRelay` to provide the same feature.

Here's a basic example showing the use of an autowired rest template created elsewhere ("foo.com" is a Resource Server accepting the same tokens as the surrounding app):

MyController.java

```
@Autowired
private OAuth2RestOperations restTemplate;

@RequestMapping("/relay")
public String relay() {
    ResponseEntity<String> response =
        restTemplate.getForEntity("https://foo.com/bar", String.class);
    return "Success! (" + response.getBody() + ")";
}
```

If you don't want to forward tokens (and that is a valid choice, since you might want to act as yourself, rather than the client that sent you the token), then you only need to create your own `OAuth2Context` instead of autowiring the default one.

Feign clients will also pick up an interceptor that uses the `OAuth2ClientContext` if it is available, so they should also do a token relay anywhere where a `RestTemplate` would.

12.3. Configuring Authentication Downstream of a Zuul Proxy

You can control the authorization behaviour downstream of an `@EnableZuulProxy` through the

`proxy.auth.*` settings. Example:

application.yml

```
proxy:
  auth:
    routes:
      customers: oauth2
      stores: passthru
      recommendations: none
```

In this example the "customers" service gets an OAuth2 token relay, the "stores" service gets a passthrough (the authorization header is just passed downstream), and the "recommendations" service has its authorization header removed. The default behaviour is to do a token relay if there is a token available, and passthru otherwise.

See [ProxyAuthenticationProperties](#) for full details.

Chapter 13. Spring Cloud for Cloud Foundry

Spring Cloud for Cloudfoundry makes it easy to run [Spring Cloud](#) apps in [Cloud Foundry](#) (the Platform as a Service). Cloud Foundry has the notion of a "service", which is middleware that you "bind" to an app, essentially providing it with an environment variable containing credentials (e.g. the location and username to use for the service).

The `spring-cloud-cloudfoundry-commons` module configures the Reactor-based Cloud Foundry Java client, v 3.0, and can be used standalone.

The `spring-cloud-cloudfoundry-web` project provides basic support for some enhanced features of webapps in Cloud Foundry: binding automatically to single-sign-on services and optionally enabling sticky routing for discovery.

The `spring-cloud-cloudfoundry-discovery` project provides an implementation of Spring Cloud Commons `DiscoveryClient` so you can `@EnableDiscoveryClient` and provide your credentials as `spring.cloud.cloudfoundry.discovery.[username,password]` (also `*.url` if you are not connecting to [Pivotal Web Services](#)) and then you can use the `DiscoveryClient` directly or via a `LoadBalancerClient`.

The first time you use it the discovery client might be slow owing to the fact that it has to get an access token from Cloud Foundry.

13.1. Discovery

Here's a Spring Cloud app with Cloud Foundry discovery:

app.groovy

```
@Grab('org.springframework.cloud:spring-cloud-cloudfoundry')
@RestController
@EnableDiscoveryClient
class Application {

    @Autowired
    DiscoveryClient client

    @RequestMapping('/')
    String home() {
        'Hello from ' + client.getLocalServiceInstance()
    }
}
```

If you run it without any service bindings:

```
$ spring jar app.jar app.groovy
$ cf push -p app.jar
```

It will show its app name in the home page.

The `DiscoveryClient` can list all the apps in a space, according to the credentials it is authenticated with, where the space defaults to the one the client is running in (if any). If neither org nor space are configured, they default per the user's profile in Cloud Foundry.

13.2. Single Sign On



All of the OAuth2 SSO and resource server features moved to Spring Boot in version 1.3. You can find documentation in the [Spring Boot user guide](#).

This project provides automatic binding from CloudFoundry service credentials to the Spring Boot features. If you have a CloudFoundry service called "sso", for instance, with credentials containing "client_id", "client_secret" and "auth_domain", it will bind automatically to the Spring OAuth2 client that you enable with `@EnableOAuth2Sso` (from Spring Boot). The name of the service can be parameterized using `spring.oauth2.sso.serviceId`.

13.3. Configuration

To see the list of all Spring Cloud Cloud Foundry related configuration properties please check [the Appendix page](#).

Chapter 14. Spring Cloud Contract Reference Documentation

Adam Dudczak, Mathias Düsterhöft, Marcin Grzejszczak, Dennis Kieselhorst, Jakub Kubryński, Karol Lassak, Olga Maciaszek-Sharma, Mariusz Smykuła, Dave Syer, Jay Bryant

Legal

3.0.0-M2

Copyright © 2012-2020

Copies of this document may be made for your own use and for distribution to others, provided that you do not charge any fee for such copies and further provided that each copy contains this Copyright Notice, whether distributed in print or electronically.

14.1. Getting Started

If you are getting started with Spring Cloud Contract, or Spring in general, start by reading this section. It answers the basic “what?”, “how?” and “why?” questions. It includes an introduction to Spring Cloud Contract, along with installation instructions. We then walk you through building your first Spring Cloud Contract application, discussing some core principles as we go.

14.1.1. Introducing Spring Cloud Contract

Spring Cloud Contract moves TDD to the level of software architecture. It lets you perform consumer-driven and producer-driven contract testing.

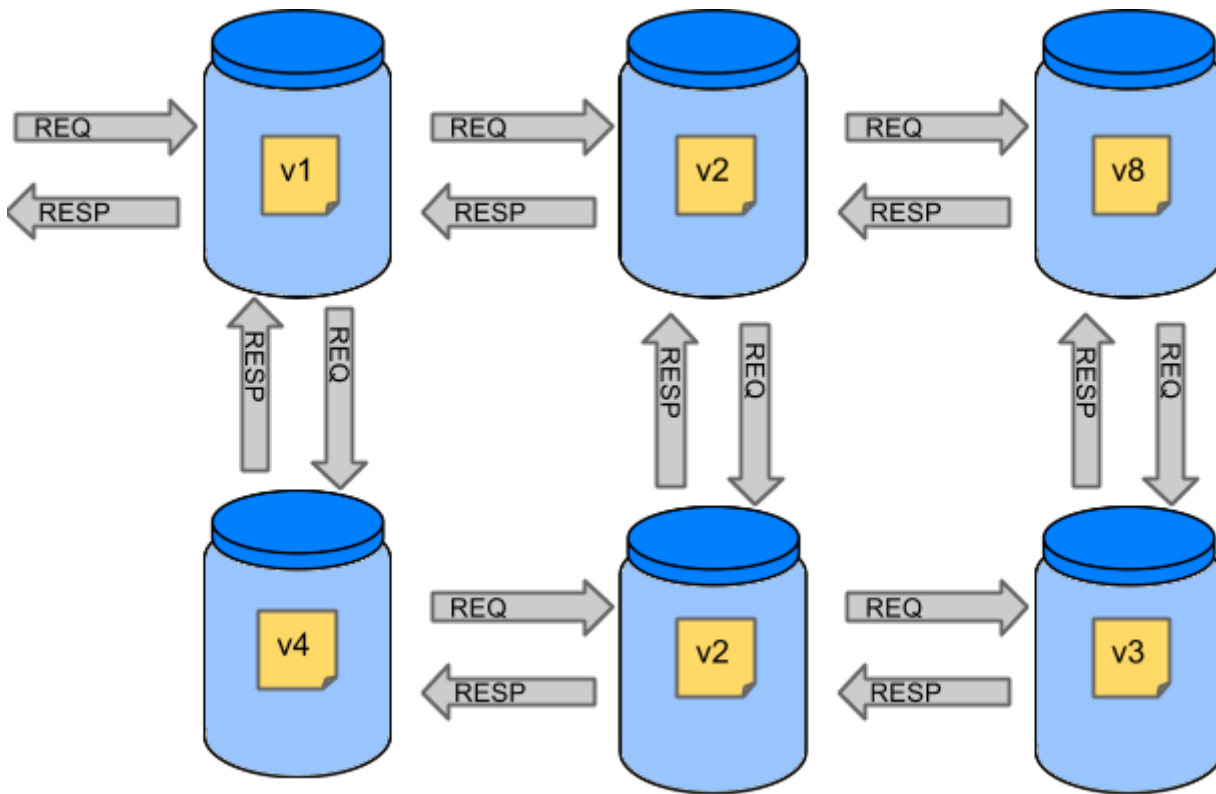
History

Before becoming Spring Cloud Contract, this project was called [Accurest](#). It was created by [Marcin Grzejszczak](#) and [Jakub Kubrynski](#) from ([Codearte](#)).

The [0.1.0](#) release took place on 26 Jan 2015 and it became stable with [1.0.0](#) release on 29 Feb 2016.

Why Do You Need It?

Assume that we have a system that consists of multiple microservices, as the following image shows:



Testing Issues

If we want to test the application in the top left corner of the image in the preceding section to determine whether it can communicate with other services, we could do one of two things:

- Deploy all microservices and perform end-to-end tests.
- Mock other microservices in unit and integration tests.

Both have their advantages but also a lot of disadvantages.

Deploy all microservices and perform end to end tests

Advantages:

- Simulates production.
- Tests real communication between services.

Disadvantages:

- To test one microservice, we have to deploy six microservices, a couple of databases, and other items.
- The environment where the tests run is locked for a single suite of tests (nobody else would be able to run the tests in the meantime).
- They take a long time to run.
- The feedback comes very late in the process.
- They are extremely hard to debug.

Mock other microservices in unit and integration tests

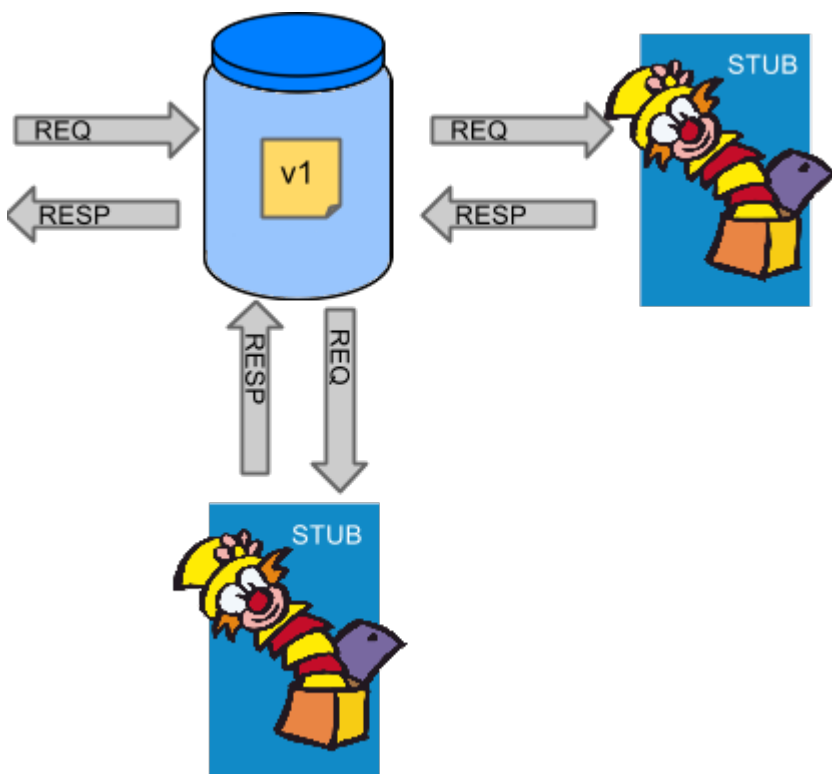
Advantages:

- They provide very fast feedback.
- They have no infrastructure requirements.

Disadvantages:

- The implementor of the service creates stubs that might have nothing to do with reality.
- You can go to production with passing tests and failing production.

To solve the aforementioned issues, Spring Cloud Contract was created. The main idea is to give you very fast feedback, without the need to set up the whole world of microservices. If you work on stubs, then the only applications you need are those that your application directly uses. The following image shows the relationship of stubs to an application:



Spring Cloud Contract gives you the certainty that the stubs that you use were created by the service that you call. Also, if you can use them, it means that they were tested against the producer's side. In short, you can trust those stubs.

Purposes

The main purposes of Spring Cloud Contract are:

- To ensure that HTTP and Messaging stubs (used when developing the client) do exactly what the actual server-side implementation does.
- To promote the ATDD (acceptance test-driven development) method and the microservices architectural style.
- To provide a way to publish changes in contracts that are immediately visible on both sides.

- To generate boilerplate test code to be used on the server side.

By default, Spring Cloud Contract integrates with [Wiremock](#) as the HTTP server stub.



Spring Cloud Contract's purpose is NOT to start writing business features in the contracts. Assume that we have a business use case of fraud check. If a user can be a fraud for 100 different reasons, we would assume that you would create two contracts, one for the positive case and one for the negative case. Contract tests are used to test contracts between applications and not to simulate full behavior.

What Is a Contract?

As consumers of services, we need to define what exactly we want to achieve. We need to formulate our expectations. That is why we write contracts. In other words, a contract is an agreement on how the API or message communication should look. Consider the following example:

Assume that you want to send a request that contains the ID of a client company and the amount it wants to borrow from us. You also want to send it to the `/fraudcheck` URL via the `PUT` method. The following listing shows a contract to check whether a client should be marked as a fraud in both Groovy and YAML:

groovy

```
/*
 * Copyright 2013-2020 the original author or authors.
 *
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 *     https://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */

package contracts

org.springframework.cloud.contract.spec.Contract.make {
    request { // (1)
        method 'PUT' // (2)
        url '/fraudcheck' // (3)
        body([ // (4)
            "client.id": $(regex('[0-9]{10}')),
            loanAmount : 99999
        ])
        headers { // (5)
```

```

        contentType('application/json')
    }
}
response { // (6)
    status OK() // (7)
    body([ // (8)
        fraudCheckStatus : "FRAUD",
        "rejection.reason": "Amount too high"
    ])
    headers { // (9)
        contentType('application/json')
    }
}
}
}

```

/*

From the Consumer perspective, when shooting a request in the integration test:

- (1) - If the consumer sends a request
- (2) - With the "PUT" method
- (3) - to the URL "/fraudcheck"
- (4) - with the JSON body that
 - * has a field `client.id` that matches a regular expression `[0-9]{10}`
 - * has a field `loanAmount` that is equal to `99999`
- (5) - with header `Content-Type` equal to `application/json`
- (6) - then the response will be sent with
- (7) - status equal `200`
- (8) - and JSON body equal to


```
{ "fraudCheckStatus": "FRAUD", "rejectionReason": "Amount too high" }
```
- (9) - with header `Content-Type` equal to `application/json`

From the Producer perspective, in the autogenerated producer-side test:

- (1) - A request will be sent to the producer
 - (2) - With the "PUT" method
 - (3) - to the URL "/fraudcheck"
 - (4) - with the JSON body that
 - * has a field `client.id` that will have a generated value that matches a regular expression `[0-9]{10}`
 - * has a field `loanAmount` that is equal to `99999`
 - (5) - with header `Content-Type` equal to `application/json`
 - (6) - then the test will assert if the response has been sent with
 - (7) - status equal `200`
 - (8) - and JSON body equal to


```
{ "fraudCheckStatus": "FRAUD", "rejectionReason": "Amount too high" }
```
 - (9) - with header `Content-Type` matching `application/json.*`
- */

yaml

```
request: # (1)
```



```
method: PUT # (2)
url: /yamlfraudcheck # (3)
body: # (4)
  "client.id": 1234567890
  loanAmount: 99999
headers: # (5)
  Content-Type: application/json
matchers:
  body:
    - path: $.['client.id'] # (6)
      type: by_regex
      value: "[0-9]{10}"
response: # (7)
  status: 200 # (8)
  body: # (9)
    fraudCheckStatus: "FRAUD"
    "rejection.reason": "Amount too high"
  headers: # (10)
    Content-Type: application/json
```

```
#From the Consumer perspective, when shooting a request in the integration test:
#
```

```
#(1) - If the consumer sends a request
#(2) - With the "PUT" method
#(3) - to the URL "/yamlfraudcheck"
#(4) - with the JSON body that
# * has a field `client.id`
# * has a field `loanAmount` that is equal to `99999`
#(5) - with header `Content-Type` equal to `application/json`
#(6) - and a `client.id` json entry matches the regular expression `[0-9]{10}`
#(7) - then the response will be sent with
#(8) - status equal `200`
#(9) - and JSON body equal to
# { "fraudCheckStatus": "FRAUD", "rejectionReason": "Amount too high" }
#(10) - with header `Content-Type` equal to `application/json`
#
```

```
#From the Producer perspective, in the autogenerated producer-side test:
#
```

```
#(1) - A request will be sent to the producer
#(2) - With the "PUT" method
#(3) - to the URL "/yamlfraudcheck"
#(4) - with the JSON body that
# * has a field `client.id` `1234567890`
# * has a field `loanAmount` that is equal to `99999`
#(5) - with header `Content-Type` equal to `application/json`
#(7) - then the test will assert if the response has been sent with
#(8) - status equal `200`
#(9) - and JSON body equal to
# { "fraudCheckStatus": "FRAUD", "rejectionReason": "Amount too high" }
#(10) - with header `Content-Type` equal to `application/json`
```

14.1.2. A Three-second Tour

This very brief tour walks through using Spring Cloud Contract. It consists of the following topics:

- [On the Producer Side](#)
- [On the Consumer Side](#)

You can find a somewhat longer tour [here](#).

The following UML diagram shows the relationship of the parts within Spring Cloud Contract:

[getting started three second] | *getting-started-three-second.png*

On the Producer Side

To start working with Spring Cloud Contract, you can add files with REST or messaging contracts expressed in either Groovy DSL or YAML to the contracts directory, which is set by the `contractsDslDir` property. By default, it is `$rootDir/src/test/resources/contracts`.

Then you can add the Spring Cloud Contract Verifier dependency and plugin to your build file, as the following example shows:

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-contract-verifier</artifactId>
  <scope>test</scope>
</dependency>
```

The following listing shows how to add the plugin, which should go in the `build/plugins` portion of the file:

```
<plugin>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-contract-maven-plugin</artifactId>
  <version>${spring-cloud-contract.version}</version>
  <extensions>true</extensions>
</plugin>
```

Running `./mvnw clean install` automatically generates tests that verify the application compliance with the added contracts. By default, the tests get generated under `org.springframework.cloud.contract.verifier.tests..`

As the implementation of the functionalities described by the contracts is not yet present, the tests fail.

To make them pass, you must add the correct implementation of either handling HTTP requests or messages. Also, you must add a base test class for auto-generated tests to the project. This class is extended by all the auto-generated tests, and it should contain all the setup information necessary to run them (for example `RestAssuredMockMvc` controller setup or messaging test setup).

The following example, from `pom.xml`, shows how to specify the base test class:

```
<build>
  <plugins>
    <plugin>
      <groupId>org.springframework.cloud</groupId>
      <artifactId>spring-cloud-contract-maven-plugin</artifactId>
      <version>2.1.2.RELEASE</version>
      <extensions>true</extensions>
      <configuration>

<baseClassForTests>com.example.contractTest.BaseTestClass</baseClassForTests> ①
      </configuration>
    </plugin>
    <plugin>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-maven-plugin</artifactId>
    </plugin>
  </plugins>
</build>
```

① The `baseClassForTests` element lets you specify your base test class. It must be a child of a `configuration` element within `spring-cloud-contract-maven-plugin`.

Once the implementation and the test base class are in place, the tests pass, and both the application and the stub artifacts are built and installed in the local Maven repository. You can now merge the changes, and you can publish both the application and the stub artifacts in an online repository.

On the Consumer Side

You can use `Spring Cloud Contract Stub Runner` in the integration tests to get a running WireMock instance or messaging route that simulates the actual service.

To do so, add the dependency to `Spring Cloud Contract Stub Runner`, as the following example shows:

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-contract-stub-runner</artifactId>
  <scope>test</scope>
</dependency>
```

You can get the Producer-side stubs installed in your Maven repository in either of two ways:

- By checking out the Producer side repository and adding contracts and generating the stubs by running the following commands:

```
$ cd local-http-server-repo
$ ./mvnw clean install -DskipTests
```



The tests are being skipped because the producer-side contract implementation is not in place yet, so the automatically-generated contract tests fail.

- By getting already-existing producer service stubs from a remote repository. To do so, pass the stub artifact IDs and artifact repository URL as **Spring Cloud Contract Stub Runner** properties, as the following example shows:

```
stubrunner:
  ids: 'com.example:http-server-dsl::stubs:8080'
  repositoryRoot: https://repo.spring.io/libs-snapshot
```

Now you can annotate your test class with **@AutoConfigureStubRunner**. In the annotation, provide the **group-id** and **artifact-id** values for **Spring Cloud Contract Stub Runner** to run the collaborators' stubs for you, as the following example shows:

```
@RunWith(SpringRunner.class)
@SpringBootTest(webEnvironment=WebEnvironment.NONE)
@AutoConfigureStubRunner(ids = {"com.example:http-server-dsl::stubs:6565"},
  stubsMode = StubRunnerProperties.StubsMode.LOCAL)
public class LoanApplicationServiceTests {
```



Use the **REMOTE** **stubsMode** when downloading stubs from an online repository and **LOCAL** for offline work.

Now, in your integration test, you can receive stubbed versions of HTTP responses or messages that are expected to be emitted by the collaborator service.

14.1.3. Developing Your First Spring Cloud Contract-based Application

This brief tour walks through using Spring Cloud Contract. It consists of the following topics:

- [On the Producer Side](#)
- [On the Consumer Side](#)

You can find an even more brief tour [here](#).

For the sake of this example, the **Stub Storage** is Nexus/Artifactory.

The following UML diagram shows the relationship of the parts of Spring Cloud Contract:

[Getting started first application] | *getting-started-three-second.png*

On the Producer Side

To start working with **Spring Cloud Contract**, you can add Spring Cloud Contract Verifier dependency and plugin to your build file, as the following example shows:

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-contract-verifier</artifactId>
  <scope>test</scope>
</dependency>
```

The following listing shows how to add the plugin, which should go in the build/plugins portion of the file:

```
<plugin>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-contract-maven-plugin</artifactId>
  <version>${spring-cloud-contract.version}</version>
  <extensions>>true</extensions>
</plugin>
```

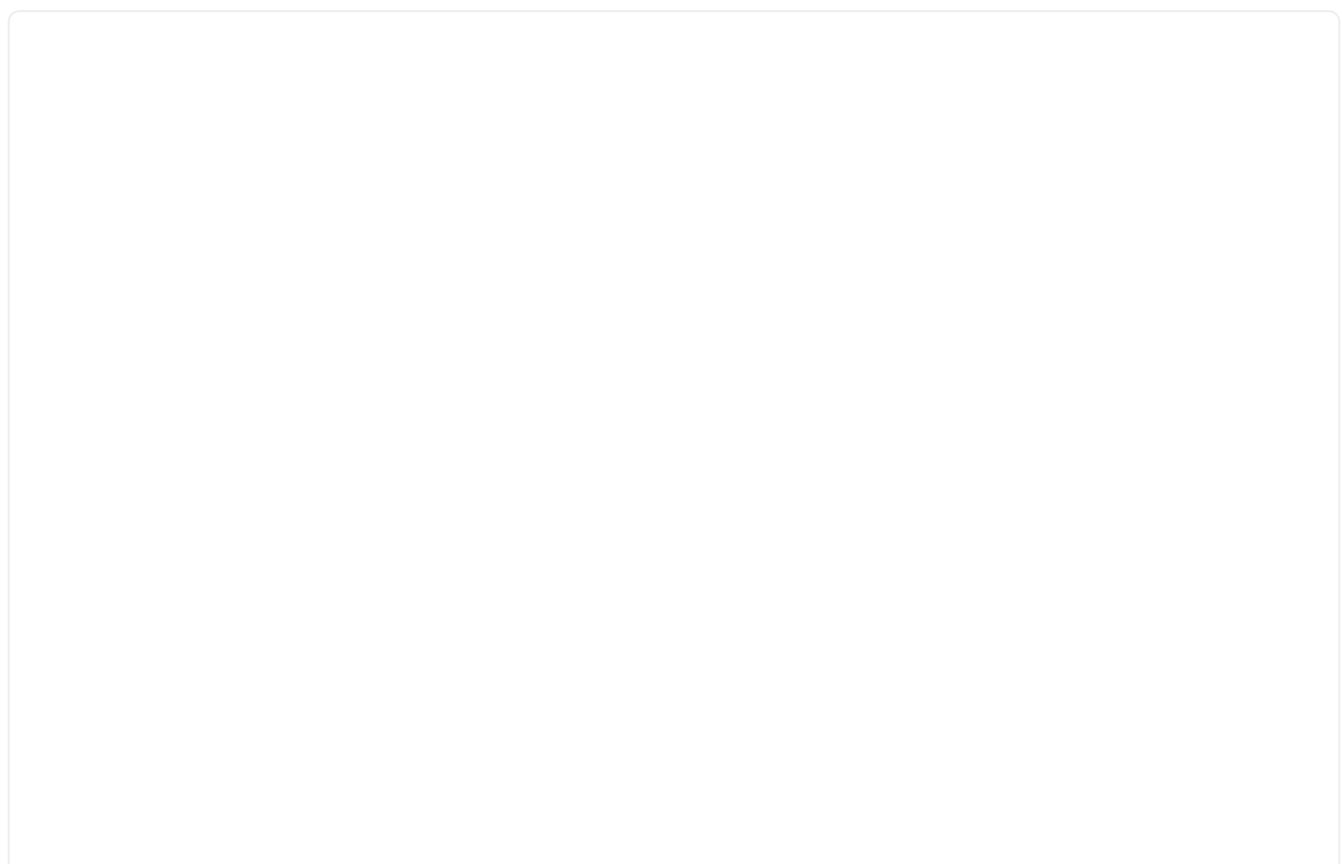
The easiest way to get started is to go to [the Spring Initializr](#) and add “Web” and “Contract Verifier” as dependencies. Doing so pulls in the previously mentioned dependencies and everything else you need in the `pom.xml` file (except for setting the base test class, which we cover later in this section). The following image shows the settings to use in [the Spring Initializr](#):



The screenshot shows the Spring Initializr interface. The left sidebar contains navigation links for Project, Language, Spring Boot, Project Metadata, and Dependencies. The main content area is divided into two tabs: Maven Project (selected) and Gradle Project. Under Maven Project, the Language is set to Java, and the Spring Boot version is 2.1.7. Project Metadata includes Group (com.example) and Artifact (demo). The Dependencies section shows two selected dependencies: Spring Web Starter and Contract Verifier. At the bottom, there are buttons for 'Generate the project' and 'Explore the project'.

Now you can add files with `REST/` messaging contracts expressed in either Groovy DSL or YAML to the `contracts` directory, which is set by the `contractsDsldir` property. By default, it is `$rootDir/src/test/resources/contracts`. Note that the file name does not matter. You can organize your contracts within this directory with whatever naming scheme you like.

For the HTTP stubs, a contract defines what kind of response should be returned for a given request (taking into account the HTTP methods, URLs, headers, status codes, and so on). The following example shows an HTTP stub contract in both Groovy and YAML:



groovy

```
package contracts

org.springframework.cloud.contract.spec.Contract.make {
    request {
        method 'PUT'
        url '/fraudcheck'
        body([
            "client.id": $(regex('[0-9]{10}')),
            loanAmount: 99999
        ])
        headers {
            contentType('application/json')
        }
    }
    response {
        status OK()
        body([
            fraudCheckStatus: "FRAUD",
            "rejection.reason": "Amount too high"
        ])
        headers {
            contentType('application/json')
        }
    }
}
```

yaml

```
request:
  method: PUT
  url: /fraudcheck
  body:
    "client.id": 1234567890
    loanAmount: 99999
  headers:
    Content-Type: application/json
  matchers:
    body:
      - path: $.['client.id']
        type: by_regex
        value: "[0-9]{10}"
response:
  status: 200
  body:
    fraudCheckStatus: "FRAUD"
    "rejection.reason": "Amount too high"
  headers:
    Content-Type: application/json;charset=UTF-8
```

If you need to use messaging, you can define:

- The input and output messages (taking into account from and where it was sent, the message body, and the header).
- The methods that should be called after the message is received.
- The methods that, when called, should trigger a message.

The following example shows a Camel messaging contract:

groovy

```
def contractDsl = Contract.make {
    name "foo"
    label 'some_label'
    input {
        messageFrom('jms:delete')
        messageBody([
            bookName: 'foo'
        ])
        messageHeaders {
            header('sample', 'header')
        }
        assertThat('bookWasDeleted()')
    }
}
```

yaml

```
label: some_label
input:
  messageFrom: jms:delete
  messageBody:
    bookName: 'foo'
  messageHeaders:
    sample: header
  assertThat: bookWasDeleted()
```

Running `./mvnw clean install` automatically generates tests that verify the application compliance with the added contracts. By default, the generated tests are under `org.springframework.cloud.contract.verifier.tests..`

The generated tests may differ, depending on which framework and test type you have setup in your plugin.

In the next listing, you can find:

- The default test mode for HTTP contracts in `MockMvc`

- A JAX-RS client with the `JAXRS` test mode
- A `WebTestClient`-based test (this is particularly recommended while working with Reactive, `WebFlux`-based applications) set with the `WEBTESTCLIENT` test mode
- A Spock-based test with the `testFramework` property set to `SPOCK`



You need only one of these test frameworks. `MockMvc` is the default. To use one of the other frameworks, add its library to your classpath.

The following listing shows samples for all frameworks:

mockmvc

```
@Test
public void validate_shouldMarkClientAsFraud() throws Exception {
    // given:
    MockMvcRequestSpecification request = given()
        .header("Content-Type", "application/vnd.fraud.v1+json")
        .body("{\"client.id\":\"1234567890\",\"loanAmount\":99999}");

    // when:
    ResponseOptions response = given().spec(request)
        .put("/fraudcheck");

    // then:
    assertThat(response.statusCode()).isEqualTo(200);
    assertThat(response.header("Content-
Type")).matches("application/vnd.fraud.v1.json.*");
    // and:
    DocumentContext parsedJson =
    JsonPath.parse(response.getBody().asString());
    assertThatJson(parsedJson).field("'fraudCheckStatus']").matches("[A-
Z]{5}");
    assertThatJson(parsedJson).field("'rejection.reason']").isEqualTo("Amount
too high");
}
```

```
@SuppressWarnings("rawtypes")
public class FooTest {
    WebTarget webTarget;

    @Test
    public void validate_() throws Exception {

        // when:
        Response response = webTarget
            .path("/users")
            .queryParam("limit", "10")
            .queryParam("offset", "20")
            .queryParam("filter", "email")
            .queryParam("sort", "name")
            .queryParam("search", "55")
            .queryParam("age", "99")
            .queryParam("name", "Denis.Stepanov")
            .queryParam("email", "bob@email.com")
            .request()
            .build("GET")
            .invoke();
        String responseAsString = response.readEntity(String.class);

        // then:
        assertThat(response.getStatus()).isEqualTo(200);

        // and:
        DocumentContext parsedJson = JsonPath.parse(responseAsString);
        assertThatJson(parsedJson).field("[ 'property1'']").isEqualTo("a");
    }
}
```

webtestclient

```
@Test
public void validate_shouldRejectABeerIfTooYoung() throws Exception {
    // given:
    WebTestClientRequestSpecification request = given()
        .header("Content-Type", "application/json")
        .body("{\"age\":10}");

    // when:
    WebTestClientResponse response = given().spec(request)
        .post("/check");

    // then:
    assertThat(response.statusCode()).isEqualTo(200);
    assertThat(response.header("Content-
Type")).matches("application/json.*");
    // and:
    DocumentContext parsedJson =
    JsonPath.parse(response.getBody().asString());
    assertThatJson(parsedJson).field("[ 'status' ]").isEqualTo("NOT_OK");
}
```

spock

```
given:
    ContractVerifierMessage inputMessage = contractVerifierMessaging.create(
        '\\\\'{"bookName":"foo"}\\\\\\',
        ['sample': 'header']
    )

when:
    contractVerifierMessaging.send(inputMessage, 'jms:delete')

then:
    noExceptionThrown()
    bookWasDeleted()
```

As the implementation of the functionalities described by the contracts is not yet present, the tests fail.

To make them pass, you must add the correct implementation of handling either HTTP requests or messages. Also, you must add a base test class for auto-generated tests to the project. This class is extended by all the auto-generated tests and should contain all the setup necessary information needed to run them (for example, [RestAssuredMockMvc](#) controller setup or messaging test setup).

The following example, from [pom.xml](#), shows how to specify the base test class:

```

<build>
  <plugins>
    <plugin>
      <groupId>org.springframework.cloud</groupId>
      <artifactId>spring-cloud-contract-maven-plugin</artifactId>
      <version>2.1.2.RELEASE</version>
      <extensions>>true</extensions>
      <configuration>

<baseClassForTests>com.example.contractTest.BaseTestClass</baseClassForTests> ①
      </configuration>
    </plugin>
    <plugin>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-maven-plugin</artifactId>
    </plugin>
  </plugins>
</build>

```

① The `baseClassForTests` element lets you specify your base test class. It must be a child of a `configuration` element within `spring-cloud-contract-maven-plugin`.

The following example shows a minimal (but functional) base test class:

```

package com.example.contractTest;

import org.junit.Before;

import io.restassured.module.mockmvc.RestAssuredMockMvc;

public class BaseTestClass {

    @Before
    public void setup() {
        RestAssuredMockMvc.standaloneSetup(new FraudController());
    }
}

```

This minimal class really is all you need to get your tests to work. It serves as a starting place to which the automatically generated tests attach.

Now we can move on to the implementation. For that, we first need a data class, which we then use in our controller. The following listing shows the data class:

```
package com.example.Test;

import com.fasterxml.jackson.annotation.JsonProperty;

public class LoanRequest {

    @JsonProperty("client.id")
    private String clientId;

    private Long loanAmount;

    public String getClientId() {
        return clientId;
    }

    public void setClientId(String clientId) {
        this.clientId = clientId;
    }

    public Long getLoanAmount() {
        return loanAmount;
    }

    public void setLoanRequestAmount(Long loanAmount) {
        this.loanAmount = loanAmount;
    }
}
```

The preceding class provides an object in which we can store the parameters. Because the client ID in the contract is called `client.id`, we need to use the `@JsonProperty("client.id")` parameter to map it to the `clientId` field.

Now we can move along to the controller, which the following listing shows:

```

package com.example.docTest;

import org.springframework.web.bind.annotation.PutMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class FraudController {

    @PutMapping(value = "/fraudcheck", consumes="application/json",
produces="application/json")
    public String check(@RequestBody LoanRequest loanRequest) { ①

        if (loanRequest.getLoanAmount() > 10000) { ②
            return "{fraudCheckStatus: FRAUD, rejection.reason: Amount too high}";
        } ③
        else {
            return "{fraudCheckStatus: OK, acceptance.reason: Amount OK}"; ④
        }
    }
}

```

- ① We map the incoming parameters to a `LoanRequest` object.
- ② We check the requested loan amount to see if it is too much.
- ③ If it is too much, we return the JSON (created with a simple string here) that the test expects.
- ④ If we had a test to catch when the amount is allowable, we could match it to this output.

The `FraudController` is about as simple as things get. You can do much more, including logging, validating the client ID, and so on.

Once the implementation and the test base class are in place, the tests pass, and both the application and the stub artifacts are built and installed in the local Maven repository. Information about installing the stubs jar to the local repository appears in the logs, as the following example shows:

```
[INFO] --- spring-cloud-contract-maven-plugin:1.0.0.BUILD-SNAPSHOT:generateStubs
(default-generateStubs) @ http-server ---
[INFO] Building jar: /some/path/http-server/target/http-server-0.0.1-SNAPSHOT-
stubs.jar
[INFO]
[INFO] --- maven-jar-plugin:2.6:jar (default-jar) @ http-server ---
[INFO] Building jar: /some/path/http-server/target/http-server-0.0.1-SNAPSHOT.jar
[INFO]
[INFO] --- spring-boot-maven-plugin:1.5.5.BUILD-SNAPSHOT:repackage (default) @
http-server ---
[INFO]
[INFO] --- maven-install-plugin:2.5.2:install (default-install) @ http-server ---
[INFO] Installing /some/path/http-server/target/http-server-0.0.1-SNAPSHOT.jar to
/path/to/your/.m2/repository/com/example/http-server/0.0.1-SNAPSHOT/http-server-
0.0.1-SNAPSHOT.jar
[INFO] Installing /some/path/http-server/pom.xml to
/path/to/your/.m2/repository/com/example/http-server/0.0.1-SNAPSHOT/http-server-
0.0.1-SNAPSHOT.pom
[INFO] Installing /some/path/http-server/target/http-server-0.0.1-SNAPSHOT-
stubs.jar to /path/to/your/.m2/repository/com/example/http-server/0.0.1-
SNAPSHOT/http-server-0.0.1-SNAPSHOT-stubs.jar
```

You can now merge the changes and publish both the application and the stub artifacts in an online repository.

On the Consumer Side

You can use Spring Cloud Contract Stub Runner in the integration tests to get a running WireMock instance or messaging route that simulates the actual service.

To get started, add the dependency to [Spring Cloud Contract Stub Runner](#), as follows:

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-contract-stub-runner</artifactId>
  <scope>test</scope>
</dependency>
```

You can get the Producer-side stubs installed in your Maven repository in either of two ways:

- By checking out the Producer side repository and adding contracts and generating the stubs by running the following commands:

```
$ cd local-http-server-repo
$ ./mvnw clean install -DskipTests
```



The tests are skipped because the Producer-side contract implementation is not yet in place, so the automatically-generated contract tests fail.

- Getting already existing producer service stubs from a remote repository. To do so, pass the stub artifact IDs and artifact repository URL as **Spring Cloud Contract Stub Runner** properties, as the following example shows:

```
stubrunner:
  ids: 'com.example:http-server-dsl:+:stubs:8080'
  repositoryRoot: https://repo.spring.io/libs-snapshot
```

Now you can annotate your test class with **@AutoConfigureStubRunner**. In the annotation, provide the **group-id** and **artifact-id** for **Spring Cloud Contract Stub Runner** to run the collaborators' stubs for you, as the following example shows:

```
@RunWith(SpringRunner.class)
@SpringBootTest(webEnvironment=WebEnvironment.NONE)
@AutoConfigureStubRunner(ids = {"com.example:http-server-dsl:+:stubs:6565"},
    stubsMode = StubRunnerProperties.StubsMode.LOCAL)
public class LoanApplicationServiceTests {
```



Use the **REMOTE stubsMode** when downloading stubs from an online repository and **LOCAL** for offline work.

In your integration test, you can receive stubbed versions of HTTP responses or messages that are expected to be emitted by the collaborator service. You can see entries similar to the following in the build logs:


```

2016-07-19 14:22:25.403 INFO 41050 --- [          main]
o.s.c.c.stubrunner.AetherStubDownloader : Desired version is + - will try to
resolve the latest version
2016-07-19 14:22:25.438 INFO 41050 --- [          main]
o.s.c.c.stubrunner.AetherStubDownloader : Resolved version is 0.0.1-SNAPSHOT
2016-07-19 14:22:25.439 INFO 41050 --- [          main]
o.s.c.c.stubrunner.AetherStubDownloader : Resolving artifact com.example:http-
server:jar:stubs:0.0.1-SNAPSHOT using remote repositories []
2016-07-19 14:22:25.451 INFO 41050 --- [          main]
o.s.c.c.stubrunner.AetherStubDownloader : Resolved artifact com.example:http-
server:jar:stubs:0.0.1-SNAPSHOT to /path/to/your/.m2/repository/com/example/http-
server/0.0.1-SNAPSHOT/http-server-0.0.1-SNAPSHOT-stubs.jar
2016-07-19 14:22:25.465 INFO 41050 --- [          main]
o.s.c.c.stubrunner.AetherStubDownloader : Unpacking stub from JAR [URI:
file:/path/to/your/.m2/repository/com/example/http-server/0.0.1-SNAPSHOT/http-
server-0.0.1-SNAPSHOT-stubs.jar]
2016-07-19 14:22:25.475 INFO 41050 --- [          main]
o.s.c.c.stubrunner.AetherStubDownloader : Unpacked file to
[/var/folders/0p/xwq47sq106x1_g3dtv6qfm940000gq/T/contracts100276532569594265]
2016-07-19 14:22:27.737 INFO 41050 --- [          main]
o.s.c.c.stubrunner.StubRunnerExecutor   : All stubs are now running RunningStubs
[NamesAndPorts={com.example:http-server:0.0.1-SNAPSHOT:stubs=8080}]

```

14.1.4. Step-by-step Guide to Consumer Driven Contracts (CDC) with Contracts on the Producer Side

Consider an example of fraud detection and the loan issuance process. The business scenario is such that we want to issue loans to people but do not want them to steal from us. The current implementation of our system grants loans to everybody.

Assume that **Loan Issuance** is a client to the **Fraud Detection** server. In the current sprint, we must develop a new feature: if a client wants to borrow too much money, we mark the client as a fraud.

Technical remarks

- Fraud Detection has an **artifact-id** of **http-server**
- Loan Issuance has an artifact-id of **http-client**
- Both have a **group-id** of **com.example**
- For the sake of this example the **Stub Storage** is Nexus/Artifactory

Social remarks

- Both the client and the server development teams need to communicate directly and discuss changes while going through the process
- CDC is all about communication

The [server-side code is available here](#) and [the client code is available here](#).



In this case, the producer owns the contracts. Physically, all of the contracts are in the producer's repository.

Technical Note

If you use the SNAPSHOT, Milestone, or Release Candidate versions you need to add the following section to your build:

Maven

```
<repositories>
  <repository>
    <id>spring-snapshots</id>
    <name>Spring Snapshots</name>
    <url>https://repo.spring.io/snapshot</url>
    <snapshots>
      <enabled>true</enabled>
    </snapshots>
  </repository>
  <repository>
    <id>spring-milestones</id>
    <name>Spring Milestones</name>
    <url>https://repo.spring.io/milestone</url>
    <snapshots>
      <enabled>>false</enabled>
    </snapshots>
  </repository>
  <repository>
    <id>spring-releases</id>
    <name>Spring Releases</name>
    <url>https://repo.spring.io/release</url>
    <snapshots>
      <enabled>>false</enabled>
    </snapshots>
  </repository>
</repositories>
<pluginRepositories>
  <pluginRepository>
    <id>spring-snapshots</id>
    <name>Spring Snapshots</name>
    <url>https://repo.spring.io/snapshot</url>
    <snapshots>
      <enabled>true</enabled>
    </snapshots>
  </pluginRepository>
  <pluginRepository>
    <id>spring-milestones</id>
    <name>Spring Milestones</name>
```

```
<url>https://repo.spring.io/milestone</url>
<snapshots>
  <enabled>>false</enabled>
</snapshots>
</pluginRepository>
<pluginRepository>
  <id>spring-releases</id>
  <name>Spring Releases</name>
  <url>https://repo.spring.io/release</url>
  <snapshots>
    <enabled>>false</enabled>
  </snapshots>
</pluginRepository>
</pluginRepositories>
```

Gradle

```
repositories {
  mavenCentral()
  mavenLocal()
  maven { url "https://repo.spring.io/snapshot" }
  maven { url "https://repo.spring.io/milestone" }
  maven { url "https://repo.spring.io/release" }
}
```

For simplicity, we use the following acronyms:

- Loan Issuance (LI): The HTTP client
- Fraud Detection (FD): The HTTP server
- Spring Cloud Contract (SCC)

The Consumer Side (Loan Issuance)

As a developer of the Loan Issuance service (a consumer of the Fraud Detection server), you might do the following steps:

1. Start doing TDD by writing a test for your feature.
2. Write the missing implementation.
3. Clone the Fraud Detection service repository locally.
4. Define the contract locally in the repo of the fraud detection service.
5. Add the Spring Cloud Contract (SCC) plugin.
6. Run the integration tests.
7. File a pull request.
8. Create an initial implementation.
9. Take over the pull request.

10. Write the missing implementation.
11. Deploy your app.
12. Work online.

We start with the loan issuance flow, which the following UML diagram shows:

[getting started cdc client] | *getting-started-cdc-client.png*

Start Doing TDD by Writing a Test for Your Feature

The following listing shows a test that we might use to check whether a loan amount is too large:

```
@Test
public void shouldBeRejectedDueToAbnormalLoanAmount() {
    // given:
    LoanApplication application = new LoanApplication(new Client("1234567890"),
        99999);
    // when:
    LoanApplicationResult loanApplication = service.loanApplication(application);
    // then:
    assertThat(loanApplication.getLoanApplicationStatus())
        .isEqualTo(LoanApplicationStatus.LOAN_APPLICATION_REJECTED);
    assertThat(loanApplication.getRejectionReason()).isEqualTo("Amount too high");
}
```

Assume that you have written a test of your new feature. If a loan application for a big amount is received, the system should reject that loan application with some description.

Write the Missing Implementation

At some point in time, you need to send a request to the Fraud Detection service. Assume that you need to send the request containing the ID of the client and the amount the client wants to borrow. You want to send it to the `/fraudcheck` URL by using the `PUT` method. To do so, you might use code similar to the following:

```
ResponseEntity<FraudServiceResponse> response = restTemplate.exchange(
    "http://localhost:" + port + "/fraudcheck", HttpMethod.PUT,
    new HttpEntity<>(request, httpHeaders), FraudServiceResponse.class);
```

For simplicity, the port of the Fraud Detection service is set to `8080`, and the application runs on `8090`.



If you start the test at this point, it breaks, because no service currently runs on port `8080`.

Clone the Fraud Detection service repository locally

You can start by playing around with the server side contract. To do so, you must first clone it, by running the following command:

```
$ git clone https://your-git-server.com/server-side.git local-http-server-repo
```

Define the Contract Locally in the Repository of the Fraud Detection Service

As a consumer, you need to define what exactly you want to achieve. You need to formulate your expectations. To do so, write the following contract:



Place the contract in the `src/test/resources/contracts/fraud` folder. The `fraud` folder is important because the producer's test base class name references that folder.

The following example shows our contract, in both Groovy and YAML:

groovy

```
/*
 * Copyright 2013-2020 the original author or authors.
 *
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 *     https://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */

package contracts

org.springframework.cloud.contract.spec.Contract.make {
    request { // (1)
        method 'PUT' // (2)
        url '/fraudcheck' // (3)
        body([ // (4)
            "client.id": $(regex('[0-9]{10}')),
            loanAmount : 99999
        ])
        headers { // (5)
```

```

        contentType('application/json')
    }
}
response { // (6)
    status OK() // (7)
    body([ // (8)
        fraudCheckStatus : "FRAUD",
        "rejection.reason": "Amount too high"
    ])
    headers { // (9)
        contentType('application/json')
    }
}
}
}

```

/*

From the Consumer perspective, when shooting a request in the integration test:

- (1) - If the consumer sends a request
- (2) - With the "PUT" method
- (3) - to the URL "/fraudcheck"
- (4) - with the JSON body that
 - * has a field `client.id` that matches a regular expression `[0-9]{10}`
 - * has a field `loanAmount` that is equal to `99999`
- (5) - with header `Content-Type` equal to `application/json`
- (6) - then the response will be sent with
- (7) - status equal `200`
- (8) - and JSON body equal to


```
{ "fraudCheckStatus": "FRAUD", "rejectionReason": "Amount too high" }
```
- (9) - with header `Content-Type` equal to `application/json`

From the Producer perspective, in the autogenerated producer-side test:

- (1) - A request will be sent to the producer
 - (2) - With the "PUT" method
 - (3) - to the URL "/fraudcheck"
 - (4) - with the JSON body that
 - * has a field `client.id` that will have a generated value that matches a regular expression `[0-9]{10}`
 - * has a field `loanAmount` that is equal to `99999`
 - (5) - with header `Content-Type` equal to `application/json`
 - (6) - then the test will assert if the response has been sent with
 - (7) - status equal `200`
 - (8) - and JSON body equal to


```
{ "fraudCheckStatus": "FRAUD", "rejectionReason": "Amount too high" }
```
 - (9) - with header `Content-Type` matching `application/json.*`
- */

yaml

```
request: # (1)
```

```
method: PUT # (2)
url: /yamlfraudcheck # (3)
body: # (4)
  "client.id": 1234567890
  loanAmount: 99999
headers: # (5)
  Content-Type: application/json
matchers:
  body:
    - path: $.['client.id'] # (6)
      type: by_regex
      value: "[0-9]{10}"
response: # (7)
  status: 200 # (8)
  body: # (9)
    fraudCheckStatus: "FRAUD"
    "rejection.reason": "Amount too high"
  headers: # (10)
    Content-Type: application/json
```

```
#From the Consumer perspective, when shooting a request in the integration test:
#
```

```
#(1) - If the consumer sends a request
#(2) - With the "PUT" method
#(3) - to the URL "/yamlfraudcheck"
#(4) - with the JSON body that
# * has a field `client.id`
# * has a field `loanAmount` that is equal to `99999`
#(5) - with header `Content-Type` equal to `application/json`
#(6) - and a `client.id` json entry matches the regular expression `[0-9]{10}`
#(7) - then the response will be sent with
#(8) - status equal `200`
#(9) - and JSON body equal to
# { "fraudCheckStatus": "FRAUD", "rejectionReason": "Amount too high" }
#(10) - with header `Content-Type` equal to `application/json`
#
```

```
#From the Producer perspective, in the autogenerated producer-side test:
#
```

```
#(1) - A request will be sent to the producer
#(2) - With the "PUT" method
#(3) - to the URL "/yamlfraudcheck"
#(4) - with the JSON body that
# * has a field `client.id` `1234567890`
# * has a field `loanAmount` that is equal to `99999`
#(5) - with header `Content-Type` equal to `application/json`
#(7) - then the test will assert if the response has been sent with
#(8) - status equal `200`
#(9) - and JSON body equal to
# { "fraudCheckStatus": "FRAUD", "rejectionReason": "Amount too high" }
#(10) - with header `Content-Type` equal to `application/json`
```

The YML contract is quite straightforward. However, when you take a look at the Contract written with a statically typed Groovy DSL, you might wonder what the `value(client(...), server(...))` parts are. By using this notation, Spring Cloud Contract lets you define parts of a JSON block, a URL, or other structure that is dynamic. In case of an identifier or a timestamp, you need not hardcode a value. You want to allow some different ranges of values. To enable ranges of values, you can set regular expressions that match those values for the consumer side. You can provide the body by means of either a map notation or String with interpolations. We highly recommend using the map notation.



You must understand the map notation in order to set up contracts. See the [Groovy docs regarding JSON](#).

The previously shown contract is an agreement between two sides that:

- If an HTTP request is sent with all of
 - A **PUT** method on the `/fraudcheck` endpoint
 - A JSON body with a `client.id` that matches the regular expression `[0-9]{10}` and `loanAmount` equal to `99999`,
 - A **Content-Type** header with a value of `application/vnd.fraud.v1+json`
- Then an HTTP response is sent to the consumer that
 - Has status `200`
 - Contains a JSON body with the `fraudCheckStatus` field containing a value of `FRAUD` and the `rejectionReason` field having a value of `Amount too high`
 - Has a **Content-Type** header with a value of `application/vnd.fraud.v1+json`

Once you are ready to check the API in practice in the integration tests, you need to install the stubs locally.

Add the Spring Cloud Contract Verifier Plugin

We can add either a Maven or a Gradle plugin. In this example, we show how to add Maven. First, we add the **Spring Cloud Contract** BOM, as the following example shows:

```
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>org.springframework.cloud</groupId>
      <artifactId>spring-cloud-dependencies</artifactId>
      <version>${spring-cloud-release.version}</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
  </dependencies>
</dependencyManagement>
```


Next, add the **Spring Cloud Contract Verifier** Maven plugin, as the following example shows:

```
<plugin>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-contract-maven-plugin</artifactId>
  <version>${spring-cloud-contract.version}</version>
  <extensions>true</extensions>
  <configuration>

<packageWithBaseClasses>com.example.fraud</packageWithBaseClasses>
<!--           <convertToYaml>true</convertToYaml>-->
  </configuration>
  <!-- if additional dependencies are needed e.g. for Pact -->
  <dependencies>
    <dependency>
      <groupId>org.springframework.cloud</groupId>
      <artifactId>spring-cloud-contract-pact</artifactId>
      <version>${spring-cloud-contract.version}</version>
    </dependency>
  </dependencies>
</plugin>
```

Since the plugin was added, you get the **Spring Cloud Contract Verifier** features, which, from the provided contracts:

- Generate and run tests
- Produce and install stubs

You do not want to generate tests, since you, as the consumer, want only to play with the stubs. You need to skip the test generation and execution. To do so, run the following commands:

```
$ cd local-http-server-repo
$ ./mvnw clean install -DskipTests
```

Once you run those commands, you should see something like the following content in the logs:

```
[INFO] --- spring-cloud-contract-maven-plugin:1.0.0.BUILD-SNAPSHOT:generateStubs
(default-generateStubs) @ http-server ---
[INFO] Building jar: /some/path/http-server/target/http-server-0.0.1-SNAPSHOT-
stubs.jar
[INFO]
[INFO] --- maven-jar-plugin:2.6:jar (default-jar) @ http-server ---
[INFO] Building jar: /some/path/http-server/target/http-server-0.0.1-SNAPSHOT.jar
[INFO]
[INFO] --- spring-boot-maven-plugin:1.5.5.BUILD-SNAPSHOT:repackage (default) @
http-server ---
[INFO]
[INFO] --- maven-install-plugin:2.5.2:install (default-install) @ http-server ---
[INFO] Installing /some/path/http-server/target/http-server-0.0.1-SNAPSHOT.jar to
/path/to/your/.m2/repository/com/example/http-server/0.0.1-SNAPSHOT/http-server-
0.0.1-SNAPSHOT.jar
[INFO] Installing /some/path/http-server/pom.xml to
/path/to/your/.m2/repository/com/example/http-server/0.0.1-SNAPSHOT/http-server-
0.0.1-SNAPSHOT.pom
[INFO] Installing /some/path/http-server/target/http-server-0.0.1-SNAPSHOT-
stubs.jar to /path/to/your/.m2/repository/com/example/http-server/0.0.1-
SNAPSHOT/http-server-0.0.1-SNAPSHOT-stubs.jar
```

The following line is extremely important:

```
[INFO] Installing /some/path/http-server/target/http-server-0.0.1-SNAPSHOT-
stubs.jar to /path/to/your/.m2/repository/com/example/http-server/0.0.1-
SNAPSHOT/http-server-0.0.1-SNAPSHOT-stubs.jar
```

It confirms that the stubs of the `http-server` have been installed in the local repository.

Running the Integration Tests

In order to profit from the Spring Cloud Contract Stub Runner functionality of automatic stub downloading, you must do the following in your consumer side project (`Loan Application service`):

1. Add the `Spring Cloud Contract` BOM, as follows:

```
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>org.springframework.cloud</groupId>
      <artifactId>spring-cloud-dependencies</artifactId>
      <version>${spring-cloud-release-train.version}</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
  </dependencies>
</dependencyManagement>
```

2. Add the dependency to `Spring Cloud Contract Stub Runner`, as follows:

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-contract-stub-runner</artifactId>
  <scope>test</scope>
</dependency>
```

3. Annotate your test class with `@AutoConfigureStubRunner`. In the annotation, provide the `group-id` and `artifact-id` for the Stub Runner to download the stubs of your collaborators. (Optional step) Because you are playing with the collaborators offline, you can also provide the offline work switch (`StubRunnerProperties.StubsMode.LOCAL`).

```
@SpringBootTest(webEnvironment = WebEnvironment.NONE)
@AutoConfigureStubRunner(ids = {
    "com.example:http-server-dsl:0.0.1:stubs" }, stubsMode =
StubRunnerProperties.StubsMode.LOCAL)
public class LoanApplicationServiceTests {
```

Now, when you run your tests, you see something like the following output in the logs:

```

2016-07-19 14:22:25.403 INFO 41050 --- [           main]
o.s.c.c.stubrunner.AetherStubDownloader : Desired version is + - will try to
resolve the latest version
2016-07-19 14:22:25.438 INFO 41050 --- [           main]
o.s.c.c.stubrunner.AetherStubDownloader : Resolved version is 0.0.1-SNAPSHOT
2016-07-19 14:22:25.439 INFO 41050 --- [           main]
o.s.c.c.stubrunner.AetherStubDownloader : Resolving artifact com.example:http-
server:jar:stubs:0.0.1-SNAPSHOT using remote repositories []
2016-07-19 14:22:25.451 INFO 41050 --- [           main]
o.s.c.c.stubrunner.AetherStubDownloader : Resolved artifact com.example:http-
server:jar:stubs:0.0.1-SNAPSHOT to /path/to/your/.m2/repository/com/example/http-
server/0.0.1-SNAPSHOT/http-server-0.0.1-SNAPSHOT-stubs.jar
2016-07-19 14:22:25.465 INFO 41050 --- [           main]
o.s.c.c.stubrunner.AetherStubDownloader : Unpacking stub from JAR [URI:
file:/path/to/your/.m2/repository/com/example/http-server/0.0.1-SNAPSHOT/http-
server-0.0.1-SNAPSHOT-stubs.jar]
2016-07-19 14:22:25.475 INFO 41050 --- [           main]
o.s.c.c.stubrunner.AetherStubDownloader : Unpacked file to
[/var/folders/0p/xwq47sq106x1_g3dtv6qfm940000gq/T/contracts100276532569594265]
2016-07-19 14:22:27.737 INFO 41050 --- [           main]
o.s.c.c.stubrunner.StubRunnerExecutor   : All stubs are now running RunningStubs
[NamesAndPorts={com.example:http-server:0.0.1-SNAPSHOT:stubs=8080}]

```

This output means that Stub Runner has found your stubs and started a server for your application with a group ID of `com.example` and an artifact ID of `http-server` with version `0.0.1-SNAPSHOT` of the stubs and with the `stubs` classifier on port `8080`.

Filing a Pull Request

What you have done until now is an iterative process. You can play around with the contract, install it locally, and work on the consumer side until the contract works as you wish.

Once you are satisfied with the results and the test passes, you can publish a pull request to the server side. Currently, the consumer side work is done.

The Producer Side (Fraud Detection server)

As a developer of the Fraud Detection server (a server to the Loan Issuance service), you might want to do the following

- Take over the pull request
- Write the missing implementation
- Deploy the application

The following UML diagram shows the fraud detection flow:

[getting started cdc server] | [getting-started-cdc-server.png](#)

Taking over the Pull Request

As a reminder, the following listing shows the initial implementation:

```
@RequestMapping(value = "/fraudcheck", method = PUT)
public FraudCheckResult fraudCheck(@RequestBody FraudCheck fraudCheck) {
    return new FraudCheckResult(FraudCheckStatus.OK, NO_REASON);
}
```

Then you can run the following commands:

```
$ git checkout -b contract-change-pr master
$ git pull https://your-git-server.com/server-side-fork.git contract-change-pr
```

You must add the dependencies needed by the autogenerated tests, as follows:

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-contract-verifier</artifactId>
  <scope>test</scope>
</dependency>
```

In the configuration of the Maven plugin, you must pass the `packageWithBaseClasses` property, as follows:

```

<plugin>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-contract-maven-plugin</artifactId>
  <version>${spring-cloud-contract.version}</version>
  <extensions>true</extensions>
  <configuration>

<packageWithBaseClasses>com.example.fraud</packageWithBaseClasses>
<!--
      <convertToYaml>true</convertToYaml>-->
  </configuration>
  <!-- if additional dependencies are needed e.g. for Pact -->
  <dependencies>
    <dependency>
      <groupId>org.springframework.cloud</groupId>
      <artifactId>spring-cloud-contract-pact</artifactId>
      <version>${spring-cloud-contract.version}</version>
    </dependency>
  </dependencies>
</plugin>

```



This example uses “convention-based” naming by setting the `packageWithBaseClasses` property. Doing so means that the two last packages combine to make the name of the base test class. In our case, the contracts were placed under `src/test/resources/contracts/fraud`. Since you do not have two packages starting from the `contracts` folder, pick only one, which should be `fraud`. Add the `Base` suffix and capitalize `fraud`. That gives you the `FraudBase` test class name.

All the generated tests extend that class. Over there, you can set up your Spring Context or whatever is necessary. In this case, you should use [Rest Assured MVC](#) to start the server side `FraudDetectionController`. The following listing shows the `FraudBase` class:

```

/*
 * Copyright 2013-2020 the original author or authors.
 *
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 *     https://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */

package com.example.fraud;

import io.restassured.module.mockmvc.RestAssuredMockMvc;
import org.junit.jupiter.api.BeforeEach;

public class FraudBase {

    @BeforeEach
    public void setup() {
        RestAssuredMockMvc.standaloneSetup(new FraudDetectionController(),
            new FraudStatsController(stubbedStatsProvider()));
    }

    private StatsProvider stubbedStatsProvider() {
        return fraudType -> {
            switch (fraudType) {
                case DRUNKS:
                    return 100;
                case ALL:
                    return 200;
            }
            return 0;
        };
    }

    public void assertThatRejectionReasonIsNull(Object rejectionReason) {
        assert rejectionReason == null;
    }

}

```

Now, if you run the `./mvnw clean install`, you get something like the following output:

Results :

Tests in error:

ContractVerifierTest.validate_shouldMarkClientAsFraud:32 >> IllegalState
Parsed...

This error occurs because you have a new contract from which a test was generated and it failed since you have not implemented the feature. The auto-generated test would look like the following test method:

```
@Test
public void validate_shouldMarkClientAsFraud() throws Exception {
    // given:
    MockMvcRequestSpecification request = given()
        .header("Content-Type", "application/vnd.fraud.v1+json")
        .body("{\"client.id\":\"1234567890\",\"loanAmount\":99999}");

    // when:
    ResponseOptions response = given().spec(request)
        .put("/fraudcheck");

    // then:
    assertThat(response.statusCode()).isEqualTo(200);
    assertThat(response.header("Content-
Type")).matches("application/vnd.fraud.v1.json.*");
    // and:
    DocumentContext parsedJson =
    JsonPath.parse(response.getBody().asString());
    assertThatJson(parsedJson).field("[ 'fraudCheckStatus' ]").matches("[A-
Z]{5}");
    assertThatJson(parsedJson).field("[ 'rejection.reason' ]").isEqualTo("Amount
too high");
}
```

If you used the Groovy DSL, you can see that all of the `producer()` parts of the Contract that were present in the `value(consumer(...), producer(...))` blocks got injected into the test. In case of using YAML, the same applied for the `matchers` sections of the `response`.

Note that, on the producer side, you are also doing TDD. The expectations are expressed in the form of a test. This test sends a request to our own application with the URL, headers, and body defined in the contract. It is also expecting precisely defined values in the response. In other words, you have the `red` part of `red, green, and refactor`. It is time to convert the `red` into the `green`.

Write the Missing Implementation

Because you know the expected input and expected output, you can write the missing implementation as follows:

```
@RequestMapping(value = "/fraudcheck", method = PUT)
public FraudCheckResult fraudCheck(@RequestBody FraudCheck fraudCheck) {
    if (amountGreaterThanThreshold(fraudCheck)) {
        return new FraudCheckResult(FraudCheckStatus.FRAUD, AMOUNT_TOO_HIGH);
    }
    return new FraudCheckResult(FraudCheckStatus.OK, NO_REASON);
}
```

When you run `./mvnw clean install` again, the tests pass. Since the `Spring Cloud Contract Verifier` plugin adds the tests to the `generated-test-sources`, you can actually run those tests from your IDE.

Deploying Your Application

Once you finish your work, you can deploy your changes. To do so, you must first merge the branch by running the following commands:

```
$ git checkout master
$ git merge --no-ff contract-change-pr
$ git push origin master
```

Your CI might run something a command such as `./mvnw clean deploy`, which would publish both the application and the stub artifacts.

Consumer Side (Loan Issuance), Final Step

As a developer of the loan issuance service (a consumer of the Fraud Detection server), I want to:

- Merge our feature branch to `master`
- Switch to online mode of working

The following UML diagram shows the final state of the process:

[getting started cdc client final] | [getting-started-cdc-client-final.png](#)

Merging a Branch to Master

The following commands show one way to merge a branch into master with Git:

```
$ git checkout master
$ git merge --no-ff contract-change-pr
```

Working Online

Now you can disable the offline work for Spring Cloud Contract Stub Runner and indicate where the repository with your stubs is located. At this moment, the stubs of the server side are automatically downloaded from Nexus/Artifactory. You can set the value of `stubsMode` to `REMOTE`. The following code shows an example of achieving the same thing by changing the properties:

```
stubrunner:
  ids: 'com.example:http-server-dsl+:stubs:8080'
  repositoryRoot: https://repo.spring.io/libs-snapshot
```

That's it. You have finished the tutorial.

14.1.5. Next Steps

Hopefully, this section provided some of the Spring Cloud Contract basics and got you on your way to writing your own applications. If you are a task-oriented type of developer, you might want to jump over to spring.io and check out some of the [getting started](#) guides that solve specific “How do I do that with Spring?” problems. We also have Spring Cloud Contract-specific “[how-to](#)” reference documentation.

Otherwise, the next logical step is to read [Using Spring Cloud Contract](#). If you are really impatient, you could also jump ahead and read about [Spring Cloud Contract features](#).

In addition to that you can check out the following videos:

- "Consumer Driven Contracts and Your Microservice Architecture" by Olga Maciaszek-Sharma and Marcin Grzejszczak



Demo

Who is who?



CONSUMER

BLACK
TERMINAL
BLACK IDE



PRODUCER

WHITE
TERMINAL
WHITE IDE

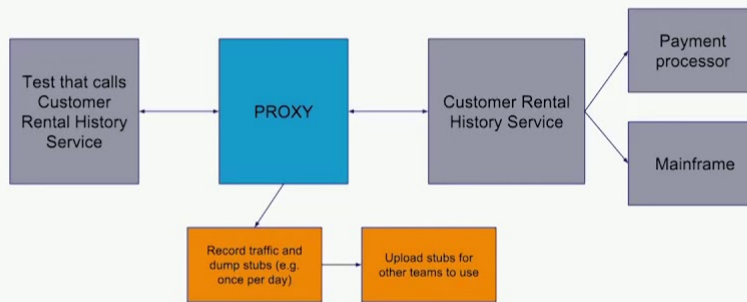
28



CONSUMER DRIVEN CONTRACTS LIKE TDD TO THE API

- "Contract Tests in the Enterprise" by Marcin Grzejszczak

Generating Stubs From Proxy



SpringOne Platform by Pivotal

Unless otherwise indicated, Pivotal and the Pivotal logo are © 2013-2017 Pivotal Software, Inc. and licensed under a Creative Commons Attribution-NonCommercial-ShareAlike license (<http://creativecommons.org/licenses/by-nc-sa/4.0/>)

27



DEVOXX™
POLAND

- "Why Contract Tests Matter?" by Marcin Grzejszczak

IT talk ^{LUB} + LJUG



You can find the default project samples at [samples](#).

You can find the Spring Cloud Contract workshops [here](#).

14.2. Using Spring Cloud Contract

This section goes into more detail about how you should use Spring Cloud Contract. It covers topics such as flows of how to work with Spring Cloud Contract. We also cover some Spring Cloud Contract best practices.

If you are starting out with Spring Cloud Contract, you should probably read the [Getting Started](#) guide before diving into this section.

14.2.1. Provider Contract Testing with Stubs in Nexus or Artifactory

You can check the [Developing Your First Spring Cloud Contract based application](#) link to see the provider contract testing with stubs in the Nexus or Artifactory flow.

You can also check the [workshop page](#) for a step-by-step instruction on how to do this flow.

14.2.2. Provider Contract Testing with Stubs in Git

In this flow, we perform the provider contract testing (the producer has no knowledge of how consumers use their API). The stubs are uploaded to a separate repository (they are not uploaded to Artifactory or Nexus).

Prerequisites

Before testing provider contracts with stubs in git, you must provide a git repository that contains all the stubs for each producer. For an example of such a project, see [this samples](#) or [this sample](#). As a result of pushing stubs there, the repository has the following structure:

```
$ tree .
├── META-INF
│   └── folder.with.group.id.as.its.name
│       ├── folder-with-artifact-id
│       │   └── folder-with-version
│       │       ├── contractA.groovy
│       │       ├── contractB.yml
│       │       └── contractC.groovy
```

You must also provide consumer code that has Spring Cloud Contract Stub Runner set up. For an example of such a project, see [this sample](#) and search for a `BeerControllerGitTest` test. You must also provide producer code that has Spring Cloud Contract set up, together with a plugin. For an example of such a project, see [this sample](#).

The Flow

The flow looks exactly as the one presented in [Developing Your First Spring Cloud Contract based application](#), but the `Stub Storage` implementation is a git repository.

You can read more about setting up a git repository and setting consumer and producer side in the [How To](#) page of the documentation.

Consumer setup

In order to fetch the stubs from a git repository instead of Nexus or Artifactory, you need to use the `git` protocol in the URL of the `repositoryRoot` property in Stub Runner. The following example shows how to set it up:

Annotation

```
@AutoConfigureStubRunner(  
    stubsMode = StubRunnerProperties.StubsMode.REMOTE,  
    repositoryRoot = "git://git@github.com:spring-cloud-samples/spring-cloud-  
contract-nodejs-contracts-git.git",  
    ids = "com.example:artifact-id:0.0.1")
```

JUnit 4 Rule

```
@Rule  
public StubRunnerRule rule = new StubRunnerRule()  
    .downloadStub("com.example","artifact-id", "0.0.1")  
    .repoRoot("git://git@github.com:spring-cloud-samples/spring-cloud-  
contract-nodejs-contracts-git.git")  
    .stubsMode(StubRunnerProperties.StubsMode.REMOTE);
```

JUnit 5 Extension

```
@RegisterExtension  
public StubRunnerExtension stubRunnerExtension = new StubRunnerExtension()  
    .downloadStub("com.example","artifact-id", "0.0.1")  
    .repoRoot("git://git@github.com:spring-cloud-samples/spring-cloud-  
contract-nodejs-contracts-git.git")  
    .stubsMode(StubRunnerProperties.StubsMode.REMOTE);
```

Setting up the Producer

In order to push the stubs to a git repository instead of Nexus or Artifactory, you need to use the **git** protocol in the URL of the plugin setup. Also you need to explicitly tell the plugin to push the stubs at the end of the build process. The following example shows how to do so:

```
<plugin>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-contract-maven-plugin</artifactId>
  <version>${spring-cloud-contract.version}</version>
  <extensions>>true</extensions>
  <configuration>
    <!-- Base class mappings etc. -->

    <!-- We want to pick contracts from a Git repository -->
    <contractsRepositoryUrl>git://git://git@github.com:spring-cloud-
samples/spring-cloud-contract-nodejs-contracts-git.git</contractsRepositoryUrl>

    <!-- We reuse the contract dependency section to set up the path
to the folder that contains the contract definitions. In our case the
path will be /groupId/artifactId/version/contracts -->
    <contractDependency>
      <groupId>${project.groupId}</groupId>
      <artifactId>${project.artifactId}</artifactId>
      <version>${project.version}</version>
    </contractDependency>

    <!-- The contracts mode can't be classpath -->
    <contractsMode>REMOTE</contractsMode>
  </configuration>
  <executions>
    <execution>
      <phase>package</phase>
      <goals>
        <!-- By default we will not push the stubs back to SCM,
you have to explicitly add it as a goal -->
        <goal>pushStubsToScm</goal>
      </goals>
    </execution>
  </executions>
</plugin>
```

gradle

```
contracts {
    // We want to pick contracts from a Git repository
    contractDependency {
        stringNotation = "${project.group}:${project.name}:${project.version}"
    }
    /*
    We reuse the contract dependency section to set up the path
    to the folder that contains the contract definitions. In our case the
    path will be /groupId/artifactId/version/contracts
    */
    contractRepository {
        repositoryUrl = "git://git://git@github.com:spring-cloud-samples/spring-
cloud-contract-nodejs-contracts-git.git"
    }
    // The mode can't be classpath
    contractsMode = "REMOTE"
    // Base class mappings etc.
}

/*
In this scenario we want to publish stubs to SCM whenever
the `publish` task is executed
*/
publish.dependsOn("publishStubsToScm")
```

You can read more about setting up a git repository in the [How To page](#) of the documentation.

14.2.3. Consumer Driven Contracts with Contracts on the Producer Side

See [Step-by-step Guide to Consumer Driven Contracts \(CDC\) with Contracts on the Producer Side](#) to see the Consumer Driven Contracts with contracts on the producer side flow.

14.2.4. Consumer Driven Contracts with Contracts in an External Repository

In this flow, we perform Consumer Driven Contract testing. The contract definitions are stored in a separate repository.

See the [workshop page](#) for step-by-step instructions on how to do this flow.

Prerequisites

To use consumer-driven contracts with the contracts held in an external repository, you need to set up a git repository that:

- Contains all the contract definitions for each producer.
- Can package the contract definitions in a JAR.

- For each contract producer, contains a way (for example, `pom.xml`) to install stubs locally through the Spring Cloud Contract Plugin (SCC Plugin)

For more information, see the [How To section](#), where we describe how to set up such a repository. For an example of such a project, see [this sample](#).

You also need consumer code that has Spring Cloud Contract Stub Runner set up. For an example of such a project, see [this sample](#). You also need producer code that has Spring Cloud Contract set up, together with a plugin. For an example of such a project, see [this sample](#). The stub storage is Nexus or Artifactory

At a high level, the flow looks as follows:

1. The consumer works with the contract definitions from the separate repository
2. Once the consumer's work is done, a branch with working code is done on the consumer side and a pull request is made to the separate repository that holds the contract definitions.
3. The producer takes over the pull request to the separate repository with contract definitions and installs the JAR with all contracts locally.
4. The producer generates tests from the locally stored JAR and writes the missing implementation to make the tests pass.
5. Once the producer's work is done, the pull request to the repository that holds the contract definitions is merged.
6. After the CI tool builds the repository with the contract definitions and the JAR with contract definitions gets uploaded to Nexus or Artifactory, the producer can merge its branch.
7. Finally, the consumer can switch to working online to fetch stubs of the producer from a remote location, and the branch can be merged to master.

Consumer Flow

The consumer:

1. Writes a test that would send a request to the producer.

The test fails due to no server being present.

2. Clones the repository that holds the contract definitions.
3. Set up the requirements as contracts under the folder with the consumer name as a subfolder of the producer.

For example, for a producer named `producer` and a consumer named `consumer`, the contracts would be stored under `src/main/resources/contracts/producer/consumer/`

4. Once the contracts are defined, installs the producer stubs to local storage, as the following example shows:

```
$ cd src/main/resource/contracts/producer
$ ./mvnw clean install
```

5. Sets up Spring Cloud Contract (SCC) Stub Runner in the consumer tests, to:

- Fetch the producer stubs from local storage.
- Work in the stubs-per-consumer mode (this enables consumer driven contracts mode).

The SCC Stub Runner:

- Fetches the producer stubs.
- Runs an in-memory HTTP server stub with the producer stubs.
- Now your test communicates with the HTTP server stub and your tests pass
- Create a pull request to the repository with contract definitions, with the new contracts for the producer
- Branch your consumer code, until the producer team has merged their code

The following UML diagram shows the consumer flow:

[flow overview consumer cdc external consumer] | *flow-overview-consumer-cdc-external-*

consumer.png

Producer Flow

The producer:

1. Takes over the pull request to the repository with contract definitions. You can do it from the command line, as follows

```
$ git checkout -b the_branch_with_pull_request master
git pull https://github.com/user_id/project_name.git
the_branch_with_pull_request
```

2. Installs the contract definitions, as follows

```
$ ./mvnw clean install
```

3. Sets up the plugin to fetch the contract definitions from a JAR instead of from `src/test/resources/contracts`, as follows:

Maven

```
<plugin>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-contract-maven-plugin</artifactId>
  <version>${spring-cloud-contract.version}</version>
  <extensions>true</extensions>
  <configuration>
    <!-- We want to use the JAR with contracts with the following
coordinates -->
    <contractDependency>
      <groupId>com.example</groupId>
      <artifactId>beer-contracts</artifactId>
    </contractDependency>
    <!-- The JAR with contracts should be taken from Maven local -->
    <contractsMode>LOCAL</contractsMode>
    <!-- ... additional configuration -->
  </configuration>
</plugin>
```

Gradle

```
contracts {
  // We want to use the JAR with contracts with the following coordinates
  // group id `com.example`, artifact id `beer-contracts`, LATEST version and
  NO classifier
  contractDependency {
    stringNotation = 'com.example:beer-contracts:::'
  }
  // The JAR with contracts should be taken from Maven local
  contractsMode = "LOCAL"
  // Additional configuration
}
```

4. Runs the build to generate tests and stubs, as follows:

Maven

```
./mvnw clean install
```

Gradle

```
./gradlew clean build
```

5. Writes the missing implementation, to make the tests pass.

6. Merges the pull request to the repository with contract definitions, as follows:

```
$ git commit -am "Finished the implementation to make the contract tests pass"  
$ git checkout master  
$ git merge --no-ff the_branch_with_pull_request  
$ git push origin master
```

7. The CI system builds the project with the contract definitions and uploads the JAR with the contract definitions to Nexus or Artifactory.

8. Switches to working remotely.

9. Sets up the plugin so that the contract definitions are no longer taken from the local storage but from a remote location, as follows:

Maven

```
<plugin>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-contract-maven-plugin</artifactId>
  <version>${spring-cloud-contract.version}</version>
  <extensions>true</extensions>
  <configuration>
    <!-- We want to use the JAR with contracts with the following
coordinates -->
    <contractDependency>
      <groupId>com.example</groupId>
      <artifactId>beer-contracts</artifactId>
    </contractDependency>
    <!-- The JAR with contracts should be taken from a remote location -->
    <contractsMode>REMOTE</contractsMode>
    <!-- ... additional configuration -->
  </configuration>
</plugin>
```

Gradle

```
contracts {
  // We want to use the JAR with contracts with the following coordinates
  // group id `com.example`, artifact id `beer-contracts`, LATEST version and
  NO classifier
  contractDependency {
    stringNotation = 'com.example:beer-contracts:::'
  }
  // The JAR with contracts should be taken from a remote location
  contractsMode = "REMOTE"
  // Additional configuration
}
```

10. Merges the producer code with the new implementation.

11. The CI system:

- Builds the project
- Generates tests, stubs, and the stub JAR
- Uploads the artifact with the application and the stubs to Nexus or Artifactory.

The following UML diagram shows the producer process:

[flow overview consumer cdc external producer] | *flow-overview-consumer-cdc-external-*

producer.png

14.2.5. Consumer Driven Contracts with Contracts on the Producer Side, Pushed to Git

You can check [Step-by-step Guide to Consumer Driven Contracts \(CDC\) with contracts laying on the producer side](#) to see the consumer driven contracts with contracts on the producer side flow.

The stub storage implementation is a git repository. We describe its setup in the [Provider Contract Testing with Stubs in Git](#) section.

You can read more about setting up a git repository for the consumer and producer sides in the [How To](#) page of the documentation.

14.2.6. Provider Contract Testing with Stubs in Artifactory for a non-Spring Application

The Flow

You can check [Developing Your First Spring Cloud Contract based application](#) to see the flow for provider contract testing with stubs in Nexus or Artifactory.

Setting up the Consumer

For the consumer side, you can use a JUnit rule. That way, you need not start a Spring context. The following listing shows such a rule (in JUnit4 and JUnit 5);

JUnit 4 Rule

```
@Rule
public StubRunnerRule rule = new StubRunnerRule()
    .downloadStub("com.example","artifact-id", "0.0.1")
    .repoRoot("git://git@github.com:spring-cloud-samples/spring-cloud-
contract-nodejs-contracts-git.git")
    .stubsMode(StubRunnerProperties.StubsMode.REMOTE);
```

JUnit 5 Extension

```
@Rule
public StubRunnerExtension stubRunnerExtension = new StubRunnerExtension()
    .downloadStub("com.example","artifact-id", "0.0.1")
    .repoRoot("git://git@github.com:spring-cloud-samples/spring-cloud-
contract-nodejs-contracts-git.git")
    .stubsMode(StubRunnerProperties.StubsMode.REMOTE);
```

Setting up the Producer

By default, the Spring Cloud Contract Plugin uses Rest Assured's `MockMvc` setup for the generated tests. Since non-Spring applications do not use `MockMvc`, you can change the `testMode` to `EXPLICIT` to send a real request to an application bound at a specific port.

In this example, we use a framework called `Javalin` to start a non-Spring HTTP server.

Assume that we have the following application:

```
package com.example.demo;

import io.javalin.Javalin;

public class DemoApplication {

    public static void main(String[] args) {
        new DemoApplication().run(7000);
    }

    public Javalin start(int port) {
        return Javalin.create().start(port);
    }

    public Javalin registerGet(Javalin app) {
        return app.get("/", ctx -> ctx.result("Hello World"));
    }

    public Javalin run(int port) {
        return registerGet(start(port));
    }

}
```

Given that application, we can set up the plugin to use the `EXPLICIT` mode (that is, to send out requests to a real port), as follows:

maven

```
<plugin>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-contract-maven-plugin</artifactId>
  <version>${spring-cloud-contract.version}</version>
  <extensions>>true</extensions>
  <configuration>
    <baseClassForTests>com.example.demo.BaseClass</baseClassForTests>
    <!-- This will setup the EXPLICIT mode for the tests -->
    <testMode>EXPLICIT</testMode>
  </configuration>
</plugin>
```

gradle

```
contracts {
  // This will setup the EXPLICIT mode for the tests
  testMode = "EXPLICIT"
  baseClassForTests = "com.example.demo.BaseClass"
}
```

The base class might resemble the following:

```

import io.javalin.Javalin;
import io.restassured.RestAssured;
import org.junit.After;
import org.junit.Before;
import org.springframework.util.SocketUtils;

public class BaseClass {

    Javalin app;

    @Before
    public void setup() {
        // pick a random port
        int port = SocketUtils.findAvailableTcpPort();
        // start the application at a random port
        this.app = start(port);
        // tell Rest Assured where the started application is
        RestAssured.baseURI = "http://localhost:" + port;
    }

    @After
    public void close() {
        // stop the server after each test
        this.app.stop();
    }

    private Javalin start(int port) {
        // reuse the production logic to start a server
        return new DemoApplication().run(port);
    }
}

```

With such a setup:

- We have setup the Spring Cloud Contract plugin to use the **EXPLICIT** mode to send real requests instead of mocked ones.
- We have defined a base class that:
 - Starts the HTTP server on a random port for each test.
 - Sets Rest Assured to send requests to that port.
 - Closes the HTTP server after each test.

14.2.7. Provider Contract Testing with Stubs in Artifactory in a non-JVM World

In this flow, we assume that:

- The API Producer and API Consumer are non-JVM applications.
- The contract definitions are written in YAML.
- The Stub Storage is Artifactory or Nexus.
- Spring Cloud Contract Docker (SCC Docker) and Spring Cloud Contract Stub Runner Docker (SCC Stub Runner Docker) images are used.

You can read more about how to use Spring Cloud Contract with Docker [in this page](#).

[Here](#), you can read a blog post about how to use Spring Cloud Contract in a polyglot world.

[Here](#), you can find a sample of a NodeJS application that uses Spring Cloud Contract both as a producer and a consumer.

Producer Flow

At a high level, the producer:

1. Writes contract definitions (for example, in YAML).
2. Sets up the build tool to:
 - a. Start the application with mocked services on a given port.

If mocking is not possible, you can setup the infrastructure and define tests in a stateful way.

- b. Run the Spring Cloud Contract Docker image and pass the port of a running application as an environment variable.

The SCC Docker image: * Generates the tests from the attached volume. * Runs the tests against the running application.

Upon test completion, stubs get uploaded to a stub storage site (such as Artifactory or Git).

The following UML diagram shows the producer flow:

[flows provider non jvm producer] | *flows-provider-non-jvm-producer.png*

Consumer Flow

At a high level, the consumer:

1. Sets up the build tool to:
 - Start the Spring Cloud Contract Stub Runner Docker image and start the stubs.

The environment variables configure:

- The stubs to fetch.
- The location of the repositories.

Note that:

- To use the local storage, you can also attach it as a volume.
 - The ports at which the stubs are running need to be exposed.
2. Run the application tests against the running stubs.

The following UML diagram shows the consumer flow:

[flows provider non jvm consumer] | *flows-provider-non-jvm-consumer.png*

14.2.8. Provider Contract Testing with REST Docs and Stubs in Nexus or Artifactory

In this flow, we do not use a Spring Cloud Contract plugin to generate tests and stubs. We write [Spring RESTDocs](#) and, from them, we automatically generate stubs. Finally, we set up our builds to package the stubs and upload them to the stub storage site — in our case, Nexus or Artifactory.

See the [workshop page](#) for a step-by-step instruction on how to use this flow.

Producer Flow

As a producer, we:

1. We write RESTDocs tests of our API.
2. We add Spring Cloud Contract Stub Runner starter to our build ([spring-cloud-starter-contract-stub-runner](#)), as follows

maven

```
<dependencies>
  <dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-contract-stub-runner</artifactId>
    <scope>test</scope>
  </dependency>
</dependencies>

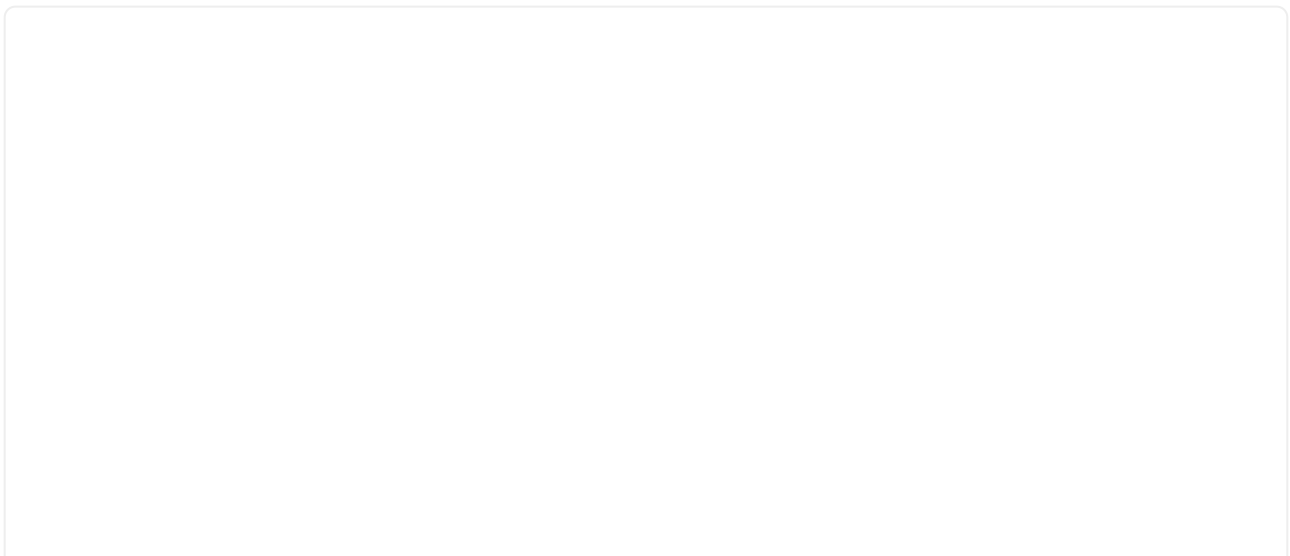
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>org.springframework.cloud</groupId>
      <artifactId>spring-cloud-dependencies</artifactId>
      <version>${spring-cloud.version}</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
  </dependencies>
</dependencyManagement>
```

gradle

```
dependencies {
    testImplementation 'org.springframework.cloud:spring-cloud-starter-
contract-stub-runner'
}

dependencyManagement {
    imports {
        mavenBom "org.springframework.cloud:spring-cloud-
dependencies:${springCloudVersion}"
    }
}
```

3. We set up the build tool to package our stubs, as follows:



```

<!-- pom.xml -->
<plugins>
  <plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-assembly-plugin</artifactId>
    <executions>
      <execution>
        <id>stub</id>
        <phase>prepare-package</phase>
        <goals>
          <goal>single</goal>
        </goals>
        <inherited>>false</inherited>
        <configuration>
          <attach>>true</attach>
          <descriptors>
            ${basedir}/src/assembly/stub.xml
          </descriptors>
        </configuration>
      </execution>
    </executions>
  </plugin>
</plugins>

<!-- src/assembly/stub.xml -->
<assembly
  xmlns="http://maven.apache.org/plugins/maven-assembly-
plugin/assembly/1.1.3"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/plugins/maven-assembly-
plugin/assembly/1.1.3 http://maven.apache.org/xsd/assembly-1.1.3.xsd">
  <id>stubs</id>
  <formats>
    <format>jar</format>
  </formats>
  <includeBaseDirectory>>false</includeBaseDirectory>
  <fileSets>
    <fileSet>
      <directory>${project.build.directory}/generated-
snippets/stubs</directory>
      <outputDirectory>META-
INF/${project.groupId}/${project.artifactId}/${project.version}/mappings</output
Directory>
      <includes>
        <include>**/*</include>
      </includes>
    </fileSet>
  </fileSets>
</assembly>

```

gradle

```
task stubsJar(type: Jar) {
    classifier = "stubs"
    into("META-INF/${project.group}/${project.name}/${project.version}/mappings") {
        include('**/*.*)
        from("${project.buildDir}/generated-snippets/stubs")
    }
}
// we need the tests to pass to build the stub jar
stubsJar.dependsOn(test)
bootJar.dependsOn(stubsJar)
```

Now, when we run the tests, stubs are automatically published and packaged.

The following UML diagram shows the producer flow:

[flows provider rest docs producer] | *flows-provider-rest-docs-producer.png*

Consumer Flow

Since the consumer flow is not affected by the tool used to generate the stubs, you can check [Developing Your First Spring Cloud Contract based application](#) to see the flow for consumer side of the provider contract testing with stubs in Nexus or Artifactory.

14.2.9. What to Read Next

You should now understand how you can use Spring Cloud Contract and some best practices that you should follow. You can now go on to learn about specific [Spring Cloud Contract features](#), or you could skip ahead and read about the [advanced features of Spring Cloud Contract](#).

14.3. Spring Cloud Contract Features

This section dives into the details of Spring Cloud Contract. Here you can learn about the key features that you may want to use and customize. If you have not already done so, you might want to read the "[getting-started.pdf](#)" and "[using.pdf](#)" sections, so that you have a good grounding of the basics.

14.3.1. Contract DSL

Spring Cloud Contract supports the DSLs written in the following languages:

- Groovy
- YAML
- Java
- Kotlin



Spring Cloud Contract supports defining multiple contracts in a single file.

The following example shows a contract definition:




```
org.springframework.cloud.contract.spec.Contract.make {
    request {
        method 'PUT'
        url '/api/12'
        headers {
            header 'Content-Type':
'application/vnd.org.springframework.cloud.contract.verifier.twitter-places-
analyzer.v1+json'
        }
        body '''\
    [{
        "created_at": "Sat Jul 26 09:38:57 +0000 2014",
        "id": 492967299297845248,
        "id_str": "492967299297845248",
        "text": "Gonna see you at Warsaw",
        "place":
        {
            "attributes": {},
            "bounding_box":
            {
                "coordinates":
                [[
                    [-77.119759,38.791645],
                    [-76.909393,38.791645],
                    [-76.909393,38.995548],
                    [-77.119759,38.995548]
                ]],
                "type": "Polygon"
            },
            "country": "United States",
            "country_code": "US",
            "full_name": "Washington, DC",
            "id": "01fbe706f872cb32",
            "name": "Washington",
            "place_type": "city",
            "url": "https://api.twitter.com/1/geo/id/01fbe706f872cb32.json"
        }
    }
    ...
}
response {
    status OK()
}
}
```

yaml

```
description: Some description
name: some name
priority: 8
ignored: true
request:
  url: /foo
  queryParameters:
    a: b
    b: c
  method: PUT
  headers:
    foo: bar
    fooReq: baz
  body:
    foo: bar
  matchers:
    body:
      - path: $.foo
        type: by_regex
        value: bar
    headers:
      - key: foo
        regex: bar
response:
  status: 200
  headers:
    foo2: bar
    foo3: foo33
    fooRes: baz
  body:
    foo2: bar
    foo3: baz
    nullValue: null
  matchers:
    body:
      - path: $.foo2
        type: by_regex
        value: bar
      - path: $.foo3
        type: by_command
        value: executeMe($it)
      - path: $.nullValue
        type: by_null
        value: null
    headers:
      - key: foo2
        regex: bar
      - key: foo3
        command: andMeToo($it)
```

java

```
import java.util.Collection;
import java.util.Collections;
import java.util.function.Supplier;

import org.springframework.cloud.contract.spec.Contract;
import org.springframework.cloud.contract.verifier.util.ContractVerifierUtil;

class contract_rest implements Supplier<Collection<Contract>> {

    @Override
    public Collection<Contract> get() {
        return Collections.singletonList(Contract.make(c -> {
            c.description("Some description");
            c.name("some name");
            c.priority(8);
            c.ignored();
            c.request(r -> {
                r.url("/foo", u -> {
                    u.queryParameters(q -> {
                        q.parameter("a", "b");
                        q.parameter("b", "c");
                    });
                });
                r.method(r.PUT());
                r.headers(h -> {
                    h.header("foo", r.value(r.client(r.regex("bar")),
r.server("bar")));
                    h.header("fooReq", "baz");
                });
                r.body(ContractVerifierUtil.map().entry("foo", "bar"));
                r.bodyMatchers(m -> {
                    m.jsonPath("$.foo", m.byRegex("bar"));
                });
            });
            c.response(r -> {
                r.fixedDelayMilliseconds(1000);
                r.status(r.OK());
                r.headers(h -> {
                    h.header("foo2", r.value(r.server(r.regex("bar")),
r.client("bar")));
                    h.header("foo3", r.value(r.server(r.execute("andMeToo($it)")),
r.client("foo33")));
                    h.header("fooRes", "baz");
                });
                r.body(ContractVerifierUtil.map().entry("foo2", "bar")
                    .entry("foo3", "baz").entry("nullValue", null));
                r.bodyMatchers(m -> {
                    m.jsonPath("$.foo2", m.byRegex("bar"));
                    m.jsonPath("$.foo3", m.byCommand("executeMe($it)"));
                });
            });
        });
    }
}
```

```
        m.jsonPath("$.nullValue", m.byNull());
    });
});
}
}
```

kotlin

```
import org.springframework.cloud.contract.spec.ContractDsl.Companion.contract
import org.springframework.cloud.contract.spec.withQueryParameters

contract {
    name = "some name"
    description = "Some description"
    priority = 8
    ignored = true
    request {
        url = url("/foo") withQueryParameters {
            parameter("a", "b")
            parameter("b", "c")
        }
        method = PUT
        headers {
            header("foo", value(client(regex("bar")), server("bar")))
            header("fooReq", "baz")
        }
        body = body(mapOf("foo" to "bar"))
        bodyMatchers {
            jsonPath("$.foo", byRegex("bar"))
        }
    }
    response {
        delay = fixedMilliseconds(1000)
        status = OK
        headers {
            header("foo2", value(server(regex("bar")), client("bar")))
            header("foo3", value(server(execute("andMeToo(\${it}")),
client("foo33"))))
            header("fooRes", "baz")
        }
        body = body(mapOf(
            "foo" to "bar",
            "foo3" to "baz",
            "nullValue" to null
        ))
        bodyMatchers {
            jsonPath("$.foo2", byRegex("bar"))
            jsonPath("$.foo3", byCommand("executeMe(\${it}"))
            jsonPath("$.nullValue", byNull)
        }
    }
}
```



You can compile contracts to stubs mapping by using the following standalone Maven command:

```
mvn org.springframework.cloud:spring-cloud-contract-maven-plugin:convert
```

Contract DSL in Groovy

If you are not familiar with Groovy, do not worry - you can use Java syntax in the Groovy DSL files as well.

If you decide to write the contract in Groovy, do not be alarmed if you have not used Groovy before. Knowledge of the language is not really needed, as the Contract DSL uses only a tiny subset of it (only literals, method calls, and closures). Also, the DSL is statically typed, to make it programmer-readable without any knowledge of the DSL itself.



Remember that, inside the Groovy contract file, you have to provide the fully qualified name to the `Contract` class and `make` static imports, such as `org.springframework.cloud.spec.Contract.make { ... }`. You can also provide an import to the `Contract` class (`import org.springframework.cloud.spec.Contract`) and then call `Contract.make { ... }`.

Contract DSL in Java

To write a contract definition in Java, you need to create a class, that implements either the `Supplier<Contract>` interface for a single contract or `Supplier<Collection<Contract>>` for multiple contracts.

You can also write the contract definitions under `src/test/java` (e.g. `src/test/java/contracts`) so that you don't have to modify the classpath of your project. In this case you'll have to provide a new location of contract definitions to your Spring Cloud Contract plugin.

Maven

```
<plugin>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-contract-maven-plugin</artifactId>
  <version>${spring-cloud-contract.version}</version>
  <extensions>>true</extensions>
  <configuration>
    <contractsDirectory>src/test/java/contracts</contractsDirectory>
  </configuration>
</plugin>
```

Gradle

```
contracts {
    contractsDslDir = new File(project.rootDir, "src/test/java/contracts")
}
```

Contract DSL in Kotlin

To get started with writing contracts in Kotlin you would need to start with a (newly created) Kotlin Script file (.kts). Just like the with the Java DSL you can put your contracts in any directory of your choice. The Maven and Gradle plugins will look at the `src/test/resources/contracts` directory by default.

You need to explicitly pass the the `spring-cloud-contract-spec-kotlin` dependency to your project plugin setup.

Maven

```
<plugin>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-contract-maven-plugin</artifactId>
  <version>${spring-cloud-contract.version}</version>
  <extensions>true</extensions>
  <configuration>
    <!-- some config -->
  </configuration>
  <dependencies>
    <dependency>
      <groupId>org.springframework.cloud</groupId>
      <artifactId>spring-cloud-contract-spec-kotlin</artifactId>
      <version>${spring-cloud-contract.version}</version>
    </dependency>
  </dependencies>
</plugin>

<dependencies>
  <!-- Remember to add this for the DSL support in the IDE and on the
consumer side -->
  <dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-contract-spec-kotlin</artifactId>
    <scope>test</scope>
  </dependency>
</dependencies>
```



```

buildscript {
    repositories {
        // ...
    }
    dependencies {
        classpath "org.springframework.cloud:spring-cloud-contract-gradle-
plugin:${scContractVersion}"
        // remember to add this:
        classpath "org.springframework.cloud:spring-cloud-contract-spec-
kotlin:${scContractVersion}"
    }
}

dependencies {
    // ...

    // Remember to add this for the DSL support in the IDE and on the consumer
side
    testImplementation "org.springframework.cloud:spring-cloud-contract-spec-
kotlin"
}

```



Remember that, inside the Kotlin Script file, you have to provide the fully qualified name to the `ContractDSL` class. Generally you would use its contract function like this: `org.springframework.cloud.contract.spec.ContractDsl.contract { ... }`. You can also provide an import to the `contract` function (`import org.springframework.cloud.contract.spec.ContractDsl.Companion.contract`) and then call `contract { ... }`.

Contract DSL in YAML

In order to see a schema of a YAML contract, you can check out the [YML Schema](#) page.

Limitations



The support for verifying the size of JSON arrays is experimental. If you want to turn it on, set the value of the following system property to `true`: `spring.cloud.contract.verifier.assert.size`. By default, this feature is set to `false`. You can also set the `assertJsonSize` property in the plugin configuration.



Because JSON structure can have any form, it can be impossible to parse it properly when using the Groovy DSL and the `value(consumer(...), producer(...))` notation in `GString`. That is why you should use the Groovy Map notation.

Common Top-Level Elements

The following sections describe the most common top-level elements:

- [Description](#)
- [Name](#)
- [Ignoring Contracts](#)
- [Contracts in Progress](#)
- [Passing Values from Files](#)

Description

You can add a **description** to your contract. The description is arbitrary text. The following code shows an example:

groovy

```
        org.springframework.cloud.contract.spec.Contract.make {
            description('')
given:
    An input
when:
    Sth happens
then:
    Output
''')
    }
```

yaml

```
description: Some description
name: some name
priority: 8
ignored: true
request:
  url: /foo
  queryParameters:
    a: b
    b: c
  method: PUT
  headers:
    foo: bar
    fooReq: baz
  body:
    foo: bar
  matchers:
    body:
      - path: $.foo
        type: by_regex
        value: bar
    headers:
      - key: foo
        regex: bar
response:
  status: 200
  headers:
    foo2: bar
    foo3: foo33
    fooRes: baz
  body:
    foo2: bar
    foo3: baz
    nullValue: null
  matchers:
    body:
      - path: $.foo2
        type: by_regex
        value: bar
      - path: $.foo3
        type: by_command
        value: executeMe($it)
      - path: $.nullValue
        type: by_null
        value: null
    headers:
      - key: foo2
        regex: bar
      - key: foo3
        command: andMeToo($it)
```

java

```
Contract.make(c -> {  
    c.description("Some description");  
}));
```

kotlin

```
contract {  
    description = ""  
given:  
    An input  
when:  
    Sth happens  
then:  
    Output  
""  
}
```

Name

You can provide a name for your contract. Assume that you provided the following name: **should register a user**. If you do so, the name of the autogenerated test is **validate_should_register_a_user**. Also, the name of the stub in a WireMock stub is **should_register_a_user.json**.



You must ensure that the name does not contain any characters that make the generated test not compile. Also, remember that, if you provide the same name for multiple contracts, your autogenerated tests fail to compile and your generated stubs override each other.

The following example shows how to add a name to a contract:

groovy

```
org.springframework.cloud.contract.spec.Contract.make {  
    name("some_special_name")  
}
```

yml

```
name: some name
```

java

```
Contract.make(c -> {  
    c.name("some name");  
}));
```

kotlin

```
contract {  
    name = "some_special_name"  
}
```

Ignoring Contracts

If you want to ignore a contract, you can either set a value for ignored contracts in the plugin configuration or set the **ignored** property on the contract itself. The following example shows how to do so:

groovy

```
org.springframework.cloud.contract.spec.Contract.make {  
    ignored()  
}
```

yml

```
ignored: true
```

java

```
Contract.make(c -> {  
    c.ignored();  
}));
```

kotlin

```
contract {  
    ignored = true  
}
```

Contracts in Progress

A contract in progress will not generate tests on the producer side, but will allow generation of stubs.



Use this feature with caution as it may lead to false positives. You generate stubs for your consumers to use without actually having the implementation in place!

If you want to set a contract in progress the following example shows how to do so:

groovy

```
org.springframework.cloud.contract.spec.Contract.make {  
    inProgress()  
}
```

yml

```
inProgress: true
```

java

```
Contract.make(c -> {  
    c.inProgress();  
}));
```

kotlin

```
contract {  
    inProgress = true  
}
```

You can set the value of the `failOnInProgress` Spring Cloud Contract plugin property to ensure that your build will break when at least one contract in progress remains in your sources.

Passing Values from Files

Starting with version `1.2.0`, you can pass values from files. Assume that you have the following resources in your project:

```
├── src  
│   ├── test  
│   │   ├── resources  
│   │   │   ├── contracts  
│   │   │   │   ├── readFromFile.groovy  
│   │   │   │   ├── request.json  
│   │   │   │   └── response.json
```

Further assume that your contract is as follows:

groovy

```
/*
 * Copyright 2013-2020 the original author or authors.
 *
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 *     https://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */

import org.springframework.cloud.contract.spec.Contract

Contract.make {
    request {
        method('PUT')
        headers {
            contentType(applicationJson())
        }
        body(file("request.json"))
        url("/1")
    }
    response {
        status OK()
        body(file("response.json"))
        headers {
            contentType(applicationJson())
        }
    }
}
}
```

yml

```
request:
  method: GET
  url: /foo
  bodyFromFile: request.json
response:
  status: 200
  bodyFromFile: response.json
```


java

```
import java.util.Collection;
import java.util.Collections;
import java.util.function.Supplier;

import org.springframework.cloud.contract.spec.Contract;

class contract_rest_from_file implements Supplier<Collection<Contract>> {

    @Override
    public Collection<Contract> get() {
        return Collections.singletonList(Contract.make(c -> {
            c.request(r -> {
                r.url("/foo");
                r.method(r.GET());
                r.body(r.file("request.json"));
            });
            c.response(r -> {
                r.status(r.OK());
                r.body(r.file("response.json"));
            });
        }));
    }
}
```

kotlin

```
import org.springframework.cloud.contract.spec.ContractDsl.Companion.contract

contract {
    request {
        url = url("/1")
        method = PUT
        headers {
            contentType = APPLICATION_JSON
        }
        body = bodyFromFile("request.json")
    }
    response {
        status = OK
        body = bodyFromFile("response.json")
        headers {
            contentType = APPLICATION_JSON
        }
    }
}
```

Further assume that the JSON files is as follows:

request.json

```
{
  "status": "REQUEST"
}
```

response.json

```
{
  "status": "RESPONSE"
}
```

When test or stub generation takes place, the contents of the `request.json` and `response.json` files are passed to the body of a request or a response. The name of the file needs to be a file with location relative to the folder in which the contract lays.

If you need to pass the contents of a file in binary form, you can use the `fileAsBytes` method in the coded DSL or a `bodyFromFileAsBytes` field in YAML.

The following example shows how to pass the contents of binary files:

groovy

```
import org.springframework.cloud.contract.spec.Contract

Contract.make {
  request {
    url("/1")
    method(PUT())
    headers {
      contentType(applicationOctetStream())
    }
    body(fileAsBytes("request.pdf"))
  }
  response {
    status 200
    body(fileAsBytes("response.pdf"))
    headers {
      contentType(applicationOctetStream())
    }
  }
}
```

yml

```
request:
  url: /1
  method: PUT
  headers:
    Content-Type: application/octet-stream
  bodyFromFileAsBytes: request.pdf
response:
  status: 200
  bodyFromFileAsBytes: response.pdf
  headers:
    Content-Type: application/octet-stream
```

java

```
import java.util.Collection;
import java.util.Collections;
import java.util.function.Supplier;

import org.springframework.cloud.contract.spec.Contract;

class contract_rest_from_pdf implements Supplier<Collection<Contract>> {

    @Override
    public Collection<Contract> get() {
        return Collections.singletonList(Contract.make(c -> {
            c.request(r -> {
                r.url("/1");
                r.method(r.PUT());
                r.body(r.fileAsBytes("request.pdf"));
                r.headers(h -> {
                    h.contentType(h.applicationOctetStream());
                });
            });
            c.response(r -> {
                r.status(r.OK());
                r.body(r.fileAsBytes("response.pdf"));
                r.headers(h -> {
                    h.contentType(h.applicationOctetStream());
                });
            });
        }));
    }
}
```

kotlin

```
import org.springframework.cloud.contract.spec.ContractDsl.Companion.contract

contract {
    request {
        url = url("/1")
        method = PUT
        headers {
            contentType = APPLICATION_OCTET_STREAM
        }
        body = bodyFromFileAsBytes("contracts/request.pdf")
    }
    response {
        status = OK
        body = bodyFromFileAsBytes("contracts/response.pdf")
        headers {
            contentType = APPLICATION_OCTET_STREAM
        }
    }
}
```



You should use this approach whenever you want to work with binary payloads, both for HTTP and messaging.

14.3.2. Contracts for HTTP

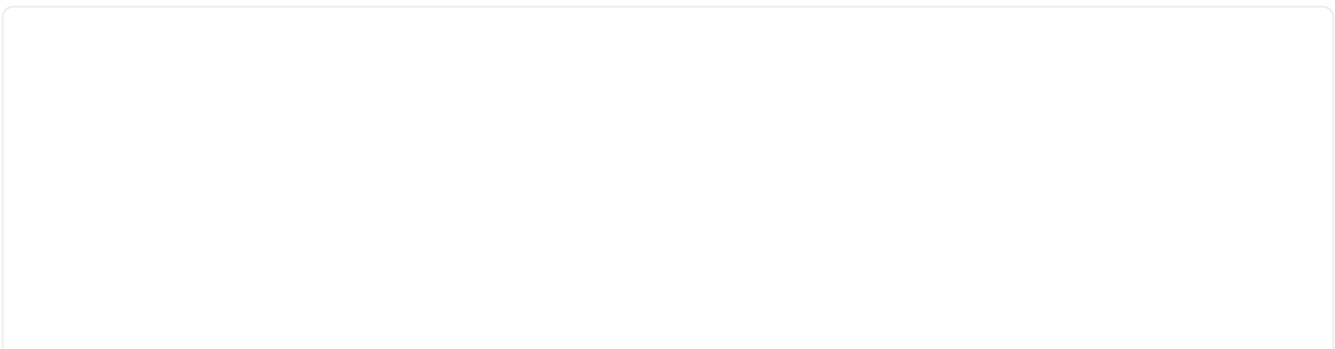
Spring Cloud Contract lets you verify applications that use REST or HTTP as a means of communication. Spring Cloud Contract verifies that, for a request that matches the criteria from the **request** part of the contract, the server provides a response that is in keeping with the **response** part of the contract. Subsequently, the contracts are used to generate WireMock stubs that, for any request matching the provided criteria, provide a suitable response.

HTTP Top-Level Elements

You can call the following methods in the top-level closure of a contract definition:

- **request**: Mandatory
- **response** : Mandatory
- **priority**: Optional

The following example shows how to define an HTTP request contract:



groovy

```
org.springframework.cloud.contract.spec.Contract.make {
    // Definition of HTTP request part of the contract
    // (this can be a valid request or invalid depending
    // on type of contract being specified).
    request {
        method GET()
        url "/foo"
        //...
    }

    // Definition of HTTP response part of the contract
    // (a service implementing this contract should respond
    // with following response after receiving request
    // specified in "request" part above).
    response {
        status 200
        //...
    }

    // Contract priority, which can be used for overriding
    // contracts (1 is highest). Priority is optional.
    priority 1
}
```

yml

```
priority: 8
request:
...
response:
...
```

java

```
org.springframework.cloud.contract.spec.Contract.make(c -> {
    // Definition of HTTP request part of the contract
    // (this can be a valid request or invalid depending
    // on type of contract being specified).
    c.request(r -> {
        r.method(r.GET());
        r.url("/foo");
        // ...
    });

    // Definition of HTTP response part of the contract
    // (a service implementing this contract should respond
    // with following response after receiving request
    // specified in "request" part above).
    c.response(r -> {
        r.status(200);
        // ...
    });

    // Contract priority, which can be used for overriding
    // contracts (1 is highest). Priority is optional.
    c.priority(1);
});
```

kotlin

```
contract {
    // Definition of HTTP request part of the contract
    // (this can be a valid request or invalid depending
    // on type of contract being specified).
    request {
        method = GET
        url = url("/foo")
        // ...
    }

    // Definition of HTTP response part of the contract
    // (a service implementing this contract should respond
    // with following response after receiving request
    // specified in "request" part above).
    response {
        status = OK
        // ...
    }

    // Contract priority, which can be used for overriding
    // contracts (1 is highest). Priority is optional.
    priority = 1
}
```

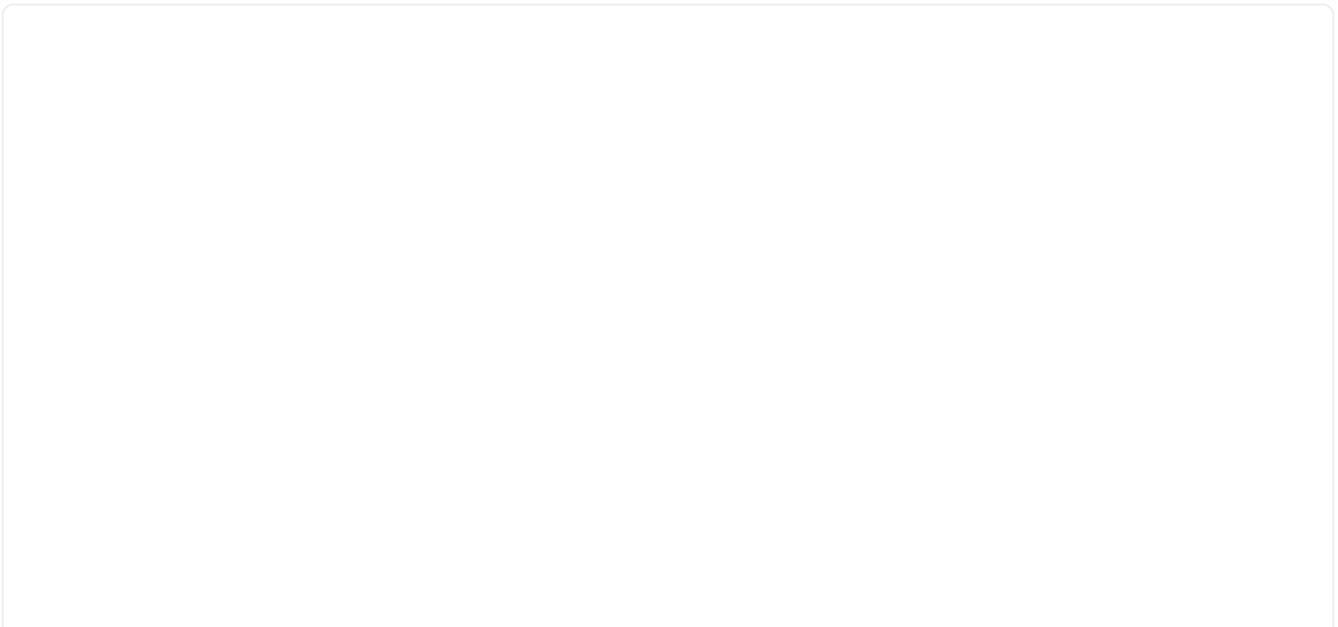


If you want to make your contract have a higher priority, you need to pass a lower number to the `priority` tag or method. For example, a `priority` with a value of `5` has higher priority than a `priority` with a value of `10`.

HTTP Request

The HTTP protocol requires only the method and the URL to be specified in a request. The same information is mandatory in request definition of the contract.

The following example shows a contract for a request:



groovy

```
org.springframework.cloud.contract.spec.Contract.make {
    request {
        // HTTP request method (GET/POST/PUT/DELETE).
        method 'GET'

        // Path component of request URL is specified as follows.
        urlPath('/users')
    }

    response {
        //...
        status 200
    }
}
```

yml

```
method: PUT
url: /foo
```

java

```
org.springframework.cloud.contract.spec.Contract.make(c -> {
    c.request(r -> {
        // HTTP request method (GET/POST/PUT/DELETE).
        r.method("GET");

        // Path component of request URL is specified as follows.
        r.urlPath("/users");
    });

    c.response(r -> {
        // ...
        r.status(200);
    });
});
```


kotlin

```
contract {
    request {
        // HTTP request method (GET/POST/PUT/DELETE).
        method = method("GET")

        // Path component of request URL is specified as follows.
        urlPath = path("/users")
    }
    response {
        // ...
        status = code(200)
    }
}
```

You can specify an absolute rather than a relative `url`, but using `urlPath` is the recommended way, as doing so makes the tests be host-independent.

The following example uses `url`:

groovy

```
org.springframework.cloud.contract.spec.Contract.make {
    request {
        method 'GET'

        // Specifying `url` and `urlPath` in one contract is illegal.
        url('http://localhost:8888/users')
    }

    response {
        //...
        status 200
    }
}
```

yml

```
request:
  method: PUT
  urlPath: /foo
```

java

```
org.springframework.cloud.contract.spec.Contract.make(c -> {
    c.request(r -> {
        r.method("GET");

        // Specifying `url` and `urlPath` in one contract is illegal.
        r.url("http://localhost:8888/users");
    });

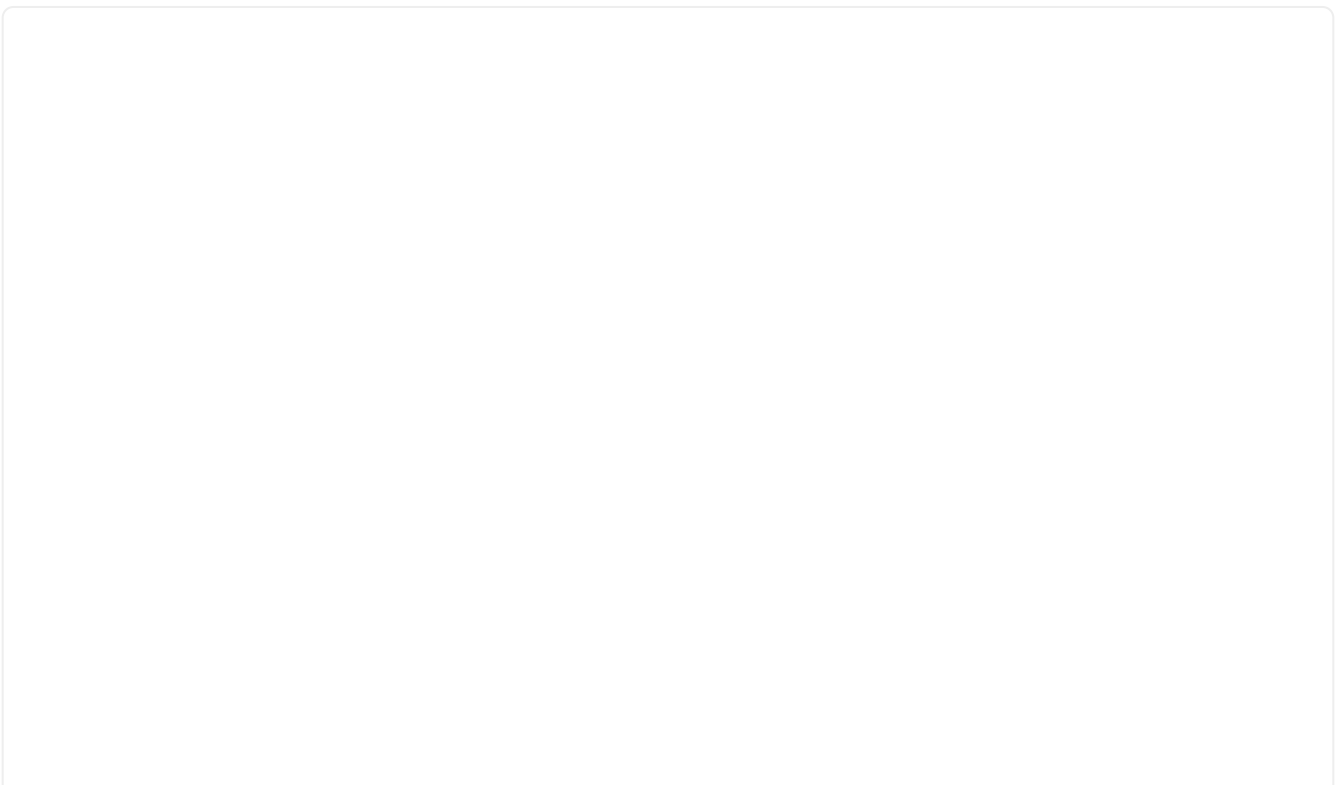
    c.response(r -> {
        // ...
        r.status(200);
    });
});
```

kotlin

```
contract {
    request {
        method = GET

        // Specifying `url` and `urlPath` in one contract is illegal.
        url("http://localhost:8888/users")
    }
    response {
        // ...
        status = OK
    }
}
```

request may contain query parameters, as the following example (which uses `urlPath`) shows:



```
org.springframework.cloud.contract.spec.Contract.make {
    request {
        //...
        method GET()

        urlPath('/users') {

            // Each parameter is specified in form
            // `paramName` : paramValue` where parameter value
            // may be a simple literal or one of matcher functions,
            // all of which are used in this example.
            queryParameters {

                // If a simple literal is used as value
                // default matcher function is used (equalTo)
                parameter 'limit': 100

                // `equalTo` function simply compares passed value
                // using identity operator (==).
                parameter 'filter': equalTo("email")

                // `containing` function matches strings
                // that contains passed substring.
                parameter 'gender': value(consumer(containing("[mf]")),
producer('mf'))

                // `matching` function tests parameter
                // against passed regular expression.
                parameter 'offset': value(consumer(matching("[0-9]+")),
producer(123))

                // `notMatching` functions tests if parameter
                // does not match passed regular expression.
                parameter 'loginStartsWith':
value(consumer(notMatching(".{0,2}")), producer(3))
            }
        }

        //...
    }

    response {
        //...
        status 200
    }
}
```

yaml

```
request:  
...  
queryParameters:  
  a: b  
  b: c
```

java

```
org.springframework.cloud.contract.spec.Contract.make(c -> {
    c.request(r -> {
        // ...
        r.method(r.GET());

        r.urlPath("/users", u -> {

            // Each parameter is specified in form
            // `paramName` : paramValue` where parameter value
            // may be a simple literal or one of matcher functions,
            // all of which are used in this example.
            u.queryParameters(q -> {

                // If a simple literal is used as value
                // default matcher function is used (equalTo)
                q.parameter("limit", 100);

                // `equalTo` function simply compares passed value
                // using identity operator (==).
                q.parameter("filter", r.equalTo("email"));

                // `containing` function matches strings
                // that contains passed substring.
                q.parameter("gender",
                    r.value(r.consumer(r.containing("[mf]")),
                        r.producer("mf")));

                // `matching` function tests parameter
                // against passed regular expression.
                q.parameter("offset",
                    r.value(r.consumer(r.matching("[0-9]+")),
                        r.producer(123)));

                // `notMatching` functions tests if parameter
                // does not match passed regular expression.
                q.parameter("loginStartsWith",
                    r.value(r.consumer(r.notMatching(".{0,2}")),
                        r.producer(3)));

            });
        });

        // ...
    });

    c.response(r -> {
        // ...
        r.status(200);
    });
});
```

```

contract {
    request {
        // ...
        method = GET

        // Each parameter is specified in form
        // `paramName` : paramValue` where parameter value
        // may be a simple literal or one of matcher functions,
        // all of which are used in this example.
        urlPath = path("/users") withQueryParameters {
            // If a simple literal is used as value
            // default matcher function is used (equalTo)
            parameter("limit", 100)

            // `equalTo` function simply compares passed value
            // using identity operator (==).
            parameter("filter", equalTo("email"))

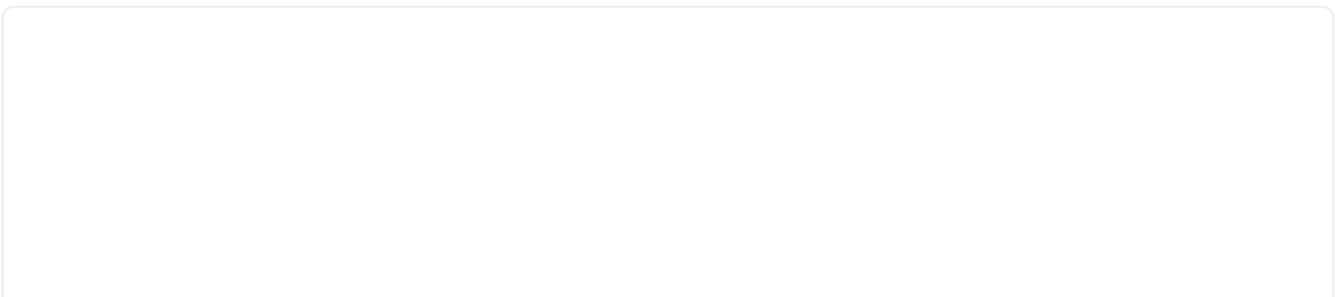
            // `containing` function matches strings
            // that contains passed substring.
            parameter("gender", value(consumer(containing("[mf]")),
producer("mf")))

            // `matching` function tests parameter
            // against passed regular expression.
            parameter("offset", value(consumer(matching("[0-9]+")),
producer(123)))

            // `notMatching` functions tests if parameter
            // does not match passed regular expression.
            parameter("loginStartsWith", value(consumer(notMatching(".{0,2}")),
producer(3)))
        }
    }
    // ...
}
response {
    // ...
    status = code(200)
}
}

```

request can contain additional request headers, as the following example shows:



groovy

```
org.springframework.cloud.contract.spec.Contract.make {
    request {
        //...
        method GET()
        url "/foo"

        // Each header is added in form `Header-Name' : 'Header-Value'`.
        // there are also some helper methods
        headers {
            header 'key': 'value'
            contentType(applicationJson())
        }

        //...
    }

    response {
        //...
        status 200
    }
}
```

yml

```
request:
  ...
headers:
  foo: bar
  fooReq: baz
```

java

```
org.springframework.cloud.contract.spec.Contract.make(c -> {
    c.request(r -> {
        // ...
        r.method(r.GET());
        r.url("/foo");

        // Each header is added in form `Header-Name' : 'Header-Value'`.
        // there are also some helper methods
        r.headers(h -> {
            h.header("key", "value");
            h.contentType(h.applicationJson());
        });

        // ...
    });

    c.response(r -> {
        // ...
        r.status(200);
    });
});
```

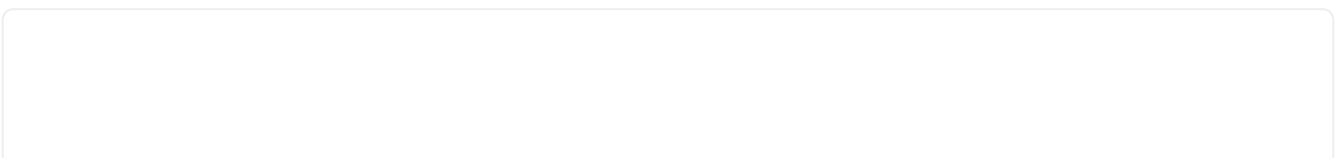
kotlin

```
contract {
    request {
        // ...
        method = GET
        url = url("/foo")

        // Each header is added in form `Header-Name' : 'Header-Value'`.
        // there are also some helper variables
        headers {
            header("key", "value")
            contentType = APPLICATION_JSON
        }

        // ...
    }
    response {
        // ...
        status = OK
    }
}
```

request may contain additional request cookies, as the following example shows:



groovy

```
org.springframework.cloud.contract.spec.Contract.make {
  request {
    //...
    method GET()
    url "/foo"

    // Each Cookies is added in form `Cookie-Key` : `Cookie-Value`.
    // there are also some helper methods
    cookies {
      cookie 'key': 'value'
      cookie('another_key', 'another_value')
    }

    //...
  }

  response {
    //...
    status 200
  }
}
```

yml

```
request:
  ...
cookies:
  foo: bar
  fooReq: baz
```

java

```
org.springframework.cloud.contract.spec.Contract.make(c -> {
    c.request(r -> {
        // ...
        r.method(r.GET());
        r.url("/foo");

        // Each Cookies is added in form ``Cookie-Key' : 'Cookie-Value'`.
        // there are also some helper methods
        r.cookies(ck -> {
            ck.cookie("key", "value");
            ck.cookie("another_key", "another_value");
        });

        // ...
    });

    c.response(r -> {
        // ...
        r.status(200);
    });
});
```

kotlin

```
contract {
    request {
        // ...
        method = GET
        url = url("/foo")

        // Each Cookies is added in form ``Cookie-Key' : 'Cookie-Value'`.
        // there are also some helper methods
        cookies {
            cookie("key", "value")
            cookie("another_key", "another_value")
        }

        // ...
    }

    response {
        // ...
        status = code(200)
    }
}
```

request may contain a request body, as the following example shows:

groovy

```
org.springframework.cloud.contract.spec.Contract.make {
    request {
        //...
        method GET()
        url "/foo"

        // Currently only JSON format of request body is supported.
        // Format will be determined from a header or body's content.
        body '''{ "login" : "john", "name": "John The Contract" }'''
    }

    response {
        //...
        status 200
    }
}
```

yml

```
request:
  ...
body:
  foo: bar
```

java

```
org.springframework.cloud.contract.spec.Contract.make(c -> {
    c.request(r -> {
        // ...
        r.method(r.GET());
        r.url("/foo");

        // Currently only JSON format of request body is supported.
        // Format will be determined from a header or body's content.
        r.body("{ \"login\" : \"john\", \"name\": \"John The Contract\" }");
    });

    c.response(r -> {
        // ...
        r.status(200);
    });
});
```

kotlin

```
contract {
    request {
        // ...
        method = GET
        url = url("/foo")

        // Currently only JSON format of request body is supported.
        // Format will be determined from a header or body's content.
        body = body("{ \"login\" : \"john\", \"name\": \"John The Contract\" }")
    }
    response {
        // ...
        status = OK
    }
}
```

request can contain multipart elements. To include multipart elements, use the **multipart** method/section, as the following examples show:

groovy

```
org.springframework.cloud.contract.spec.Contract contractDsl =
org.springframework.cloud.contract.spec.Contract.make {
    request {
        method 'PUT'
        url '/multipart'
        headers {
            contentType('multipart/form-data;boundary=AaB03x')
        }
        multipart(
            // key (parameter name), value (parameter value) pair
            formParameter: $(c(regex('.+')), p('formParameterValue')),
            someBooleanParameter: $(c(regex(anyBoolean())), p('true')),
            // a named parameter (e.g. with `file` name) that represents file
with
            // `name` and `content`. You can also call `named("fileName",
"fileContent")`
            file: named(
                // name of the file
                name: $(c(regex(nonEmpty())), p('filename.csv')),
                // content of the file
                content: $(c(regex(nonEmpty())), p('file content')),
                // content type for the part
                contentType: $(c(regex(nonEmpty()))),
p('application/json'))
        )
    }
}
```

```

    response {
        status OK()
    }
}
org.springframework.cloud.contract.spec.Contract contractDsl =
org.springframework.cloud.contract.spec.Contract.make {
    request {
        method "PUT"
        url "/multipart"
        headers {
            contentType('multipart/form-data;boundary=AaB03x')
        }
        multipart(
            file: named(
                name: value(stub(regex('.+')), test('file')),
                content: value(stub(regex('.+')), test([100, 117, 100, 97]
as byte[]))
            )
        )
    }
    response {
        status 200
    }
}
}

```

yaml

```
request:
  method: PUT
  url: /multipart
  headers:
    Content-Type: multipart/form-data;boundary=AaB03x
  multipart:
    params:
      # key (parameter name), value (parameter value) pair
      formParameter: "formParameterValue"
      someBooleanParameter: true
    named:
      - paramName: file
        fileName: filename.csv
        fileContent: file content
  matchers:
    multipart:
      params:
        - key: formParameter
          regex: ".+"
        - key: someBooleanParameter
          predefined: any_boolean
      named:
        - paramName: file
          fileName:
            predefined: non_empty
          fileContent:
            predefined: non_empty
  response:
    status: 200
```

java

```
import java.util.Collection;
import java.util.Collections;
import java.util.HashMap;
import java.util.Map;
import java.util.function.Supplier;

import org.springframework.cloud.contract.spec.Contract;
import org.springframework.cloud.contract.spec.internal.DslProperty;
import org.springframework.cloud.contract.spec.internal.Request;
import org.springframework.cloud.contract.verifier.util.ContractVerifierUtil;

class contract_multipart implements Supplier<Collection<Contract>> {

    private static Map<String, DslProperty> namedProps(Request r) {
        Map<String, DslProperty> map = new HashMap<>();
        // name of the file
```

```

        map.put("name", r.$(r.c(r.regex(r.nonEmpty()))), r.p("filename.csv"));
        // content of the file
        map.put("content", r.$(r.c(r.regex(r.nonEmpty()))), r.p("file content"));
        // content type for the part
        map.put("contentType", r.$(r.c(r.regex(r.nonEmpty()))),
r.p("application/json"));
        return map;
    }

    @Override
    public Collection<Contract> get() {
        return Collections.singletonList(Contract.make(c -> {
            c.request(r -> {
                r.method("PUT");
                r.url("/multipart");
                r.headers(h -> {
                    h.contentType("multipart/form-data;boundary=AaB03x");
                });
                r.multipart(ContractVerifierUtil.map()
                    // key (parameter name), value (parameter value) pair
                    .entry("formParameter",
                        r.$(r.c(r.regex("\.+\"")),
                            r.p("\"formParameterValue\"")))
                    .entry("someBooleanParameter",
                        r.$(r.c(r.regex(r.anyBoolean()))), r.p("true")))
                    // a named parameter (e.g. with `file` name) that
represents file
                    // with
                    // `name` and `content`. You can also call
                    `named("fileName",
                        // "fileContent")`
                    .entry("file", r.named(namedProps(r))));
            });
            c.response(r -> {
                r.status(r.OK());
            });
        }));
    }
}

```

kotlin

```
import org.springframework.cloud.contract.spec.ContractDsl.Companion.contract

contract {
    request {
        method = PUT
        url = url("/multipart")
        multipart {
            field("formParameter", value(consumer(regex("\\.+\\")),
producer("\\formParameterValue\\")))
            field("someBooleanParameter", value(consumer(anyBoolean),
producer("true")))
            field("file",
                named(
                    // name of the file
                    value(consumer(regex(nonEmpty)), producer("filename.csv")),
                    // content of the file
                    value(consumer(regex(nonEmpty)), producer("file content")),
                    // content type for the part
                    value(consumer(regex(nonEmpty)), producer("application/json"))
                )
            )
        }
        headers {
            contentType = "multipart/form-data;boundary=AaB03x"
        }
    }
    response {
        status = OK
    }
}
```

In the preceding example, we define parameters in either of two ways:

Coded DSL

- Directly, by using the map notation, where the value can be a dynamic property (such as `formParameter: $(consumer(...), producer(...))`).
- By using the `named(...)` method that lets you set a named parameter. A named parameter can set a `name` and `content`. You can call it either by using a method with two arguments, such as `named("fileName", "fileContent")`, or by using a map notation, such as `named(name: "fileName", content: "fileContent")`.

YAML

- The multipart parameters are set in the `multipart.params` section.
- The named parameters (the `fileName` and `fileContent` for a given parameter name) can be set in the `multipart.named` section. That section contains the `paramName` (the name of the parameter), `fileName` (the name of the file), `fileContent` (the content of the file) fields.

- The dynamic bits can be set via the `matchers.multipart` section.
 - For parameters, use the `params` section, which can accept `regex` or a `predefined` regular expression.
 - for named params, use the `named` section where first you define the parameter name with `paramName`. Then you can pass the parametrization of either `fileName` or `fileContent` in a `regex` or in a `predefined` regular expression.

From the contract in the preceding example, the generated test and stubs look as follows:

Test

```
// given:
MockMvcRequestSpecification request = given()
    .header("Content-Type", "multipart/form-data;boundary=AaB03x")
    .param("formParameter", "\"formParameterValue\"")
    .param("someBooleanParameter", "true")
    .multiPart("file", "filename.csv", "file content".getBytes());

// when:
ResponseOptions response = given().spec(request)
    .put("/multipart");

// then:
assertThat(response.statusCode()).isEqualTo(200);
```

```

    ...
  {
    "request" : {
      "url" : "/multipart",
      "method" : "PUT",
      "headers" : {
        "Content-Type" : {
          "matches" : "multipart/form-data;boundary=AaB03x.*"
        }
      },
      "bodyPatterns" : [ {
        "matches" : ".*--(.*)\\r\\nContent-Disposition: form-data;
name=\\\"formParameter\\\"\\r\\n(Content-Type: .*\\r\\n)?(Content-Transfer-Encoding:
.*\\r\\n)?(Content-Length: \\d+\\r\\n)?\\r\\n\\\".+\\\"\\r\\n--\\d\\d1.*"
      }, {
        "matches" : ".*--(.*)\\r\\nContent-Disposition: form-data;
name=\\\"someBooleanParameter\\\"\\r\\n(Content-Type: .*\\r\\n)?(Content-Transfer-
Encoding: .*\\r\\n)?(Content-Length: \\d+\\r\\n)?\\r\\n(true|false)\\r\\n--
\\d\\d1.*"
      }, {
        "matches" : ".*--(.*)\\r\\nContent-Disposition: form-data; name=\\\"file\\\";
filename=\\\"[\\\\S\\\\s]+\\\"\\r\\n(Content-Type: .*\\r\\n)?(Content-Transfer-
Encoding: .*\\r\\n)?(Content-Length: \\d+\\r\\n)?\\r\\n[\\\\S\\\\s]+\\r\\n--
\\d\\d1.*"
      } ]
    },
    "response" : {
      "status" : 200,
      "transformers" : [ "response-template", "foo-transformer" ]
    }
  }
  ...

```

HTTP Response

The response must contain an HTTP status code and may contain other information. The following code shows an example:

groovy

```
org.springframework.cloud.contract.spec.Contract.make {
    request {
        //...
        method GET()
        url "/foo"
    }
    response {
        // Status code sent by the server
        // in response to request specified above.
        status OK()
    }
}
```

yml

```
response:
  ...
  status: 200
```

java

```
org.springframework.cloud.contract.spec.Contract.make(c -> {
    c.request(r -> {
        // ...
        r.method(r.GET());
        r.url("/foo");
    });
    c.response(r -> {
        // Status code sent by the server
        // in response to request specified above.
        r.status(r.OK());
    });
});
```

kotlin

```
contract {
  request {
    // ...
    method = GET
    url =url("/foo")
  }
  response {
    // Status code sent by the server
    // in response to request specified above.
    status = OK
  }
}
```

Besides status, the response may contain headers, cookies, and a body, which are specified the same way as in the request (see [HTTP Request](#)).



In the Groovy DSL, you can reference the `org.springframework.cloud.contract.spec.internal.HttpStatus` methods to provide a meaningful status instead of a digit. For example, you can call `OK()` for a status `200` or `BAD_REQUEST()` for `400`.

Dynamic properties

The contract can contain some dynamic properties: timestamps, IDs, and so on. You do not want to force the consumers to stub their clocks to always return the same value of time so that it gets matched by the stub.

For the Groovy DSL, you can provide the dynamic parts in your contracts in two ways: pass them directly in the body or set them in a separate section called `bodyMatchers`.



Before 2.0.0, these were set by using `testMatchers` and `stubMatchers`. See the [migration guide](#) for more information.

For YAML, you can use only the `matchers` section.



Entries inside the `matchers` must reference existing elements of the payload. For more information check this [issue](#).

Dynamic Properties inside the Body



This section is valid only for the Coded DSL (Groovy, Java etc.). Check out the [Dynamic Properties in the Matchers Sections](#) section for YAML examples of a similar feature.

You can set the properties inside the body either with the `value` method or, if you use the Groovy

map notation, with `$()`. The following example shows how to set dynamic properties with the `value` method:

value

```
value(consumer(...), producer(...))
value(c(...), p(...))
value(stub(...), test(...))
value(client(...), server(...))
```

\$

```
$(consumer(...), producer(...))
$(c(...), p(...))
$(stub(...), test(...))
$(client(...), server(...))
```

Both approaches work equally well. The `stub` and `client` methods are aliases over the `consumer` method. Subsequent sections take a closer look at what you can do with those values.

Regular Expressions



This section is valid only for Groovy DSL. Check out the [Dynamic Properties in the Matchers Sections](#) section for YAML examples of a similar feature.

You can use regular expressions to write your requests in the contract DSL. Doing so is particularly useful when you want to indicate that a given response should be provided for requests that follow a given pattern. Also, you can use regular expressions when you need to use patterns and not exact values both for your tests and your server-side tests.

Make sure that `regex` matches a whole region of a sequence, as, internally, a call to `Pattern.matches()` is called. For instance, `abc` does not match `aabc`, but `.abc` does. There are several additional [known limitations](#) as well.

The following example shows how to use regular expressions to write a request:

groovy

```
org.springframework.cloud.contract.spec.Contract.make {
    request {
        method('GET')
        url $(consumer(~/\[/[0-9]{2}/), producer('/12'))
    }
    response {
        status OK()
        body(
            id: $(anyNumber()),
            surname: $(
                consumer('Kowalsky'),
                producer(regex('[a-zA-Z]+'))
            ),
            name: 'Jan',
            created: $(consumer('2014-02-02 12:23:43')),
            producer(execute('currentDate(it)')),
            correlationId: value(consumer('5d1f9fef-e0dc-4f3d-a7e4-
72d2220dd827'),
                producer(regex('[a-fA-F0-9]{8}-[a-fA-F0-9]{4}-[a-fA-F0-
9]{4}-[a-fA-F0-9]{4}-[a-fA-F0-9]{12}'))
            )
        )
        headers {
            header 'Content-Type': 'text/plain'
        }
    }
}
```

java

```
org.springframework.cloud.contract.spec.Contract.make(c -> {
    c.request(r -> {
        r.method("GET");
        r.url(r.$(r.consumer(r.regex("\\[/[0-9]{2}")), r.producer("/12"))));
    });
    c.response(r -> {
        r.status(r.OK());
        r.body(ContractVerifierUtil.map().entry("id", r.$(r.anyNumber()))
            .entry("surname", r.$(r.consumer("Kowalsky"),
                r.producer(r.regex("[a-zA-Z]+"))));
        r.headers(h -> {
            h.header("Content-Type", "text/plain");
        });
    });
});
```

kotlin

```
contract {
    request {
        method = method("GET")
        url = url(v(consumer(regex("\\/[0-9]{2}")), producer("/12")))
    }
    response {
        status = OK
        body(mapOf(
            "id" to v(anyNumber),
            "surname" to v(consumer("Kowalsky"), producer(regex("[a-zA-Z]+")))
        ))
        headers {
            header("Content-Type", "text/plain")
        }
    }
}
```

You can also provide only one side of the communication with a regular expression. If you do so, then the contract engine automatically provides the generated string that matches the provided regular expression. The following code shows an example for Groovy:

```
org.springframework.cloud.contract.spec.Contract.make {
    request {
        method 'PUT'
        url value(consumer(regex('/foo/[0-9]{5}')))
        body([
            requestElement: $(consumer(regex('[0-9]{5}')))
        ])
        headers {
            header('header',
$(consumer(regex('application/vnd\\.fraud\\.v1\\.json;.*'))))
        }
    }
    response {
        status OK()
        body([
            responseElement: $(producer(regex('[0-9]{7}')))
        ])
        headers {
            contentType("application/vnd.fraud.v1+json")
        }
    }
}
```

In the preceding example, the opposite side of the communication has the respective data generated for request and response.

Spring Cloud Contract comes with a series of predefined regular expressions that you can use in your contracts, as the following example shows:

```
public static RegexProperty onlyAlphaUnicode() {
    return new RegexProperty(ONLY_ALPHA_UNICODE).asString();
}

public static RegexProperty alphaNumeric() {
    return new RegexProperty(ALPHA_NUMERIC).asString();
}

public static RegexProperty number() {
    return new RegexProperty(NUMBER).asDouble();
}

public static RegexProperty positiveInt() {
    return new RegexProperty(POSITIVE_INT).asInteger();
}

public static RegexProperty anyBoolean() {
    return new RegexProperty(TRUE_OR_FALSE).asBooleanType();
}

public static RegexProperty anInteger() {
    return new RegexProperty(INTEGER).asInteger();
}

public static RegexProperty aDouble() {
    return new RegexProperty(DOUBLE).asDouble();
}

public static RegexProperty ipAddress() {
    return new RegexProperty(IP_ADDRESS).asString();
}

public static RegexProperty hostname() {
    return new RegexProperty(HOSTNAME_PATTERN).asString();
}

public static RegexProperty email() {
    return new RegexProperty(EMAIL).asString();
}

public static RegexProperty url() {
    return new RegexProperty(URL).asString();
}

public static RegexProperty httpsUrl() {
    return new RegexProperty(HTTPS_URL).asString();
}
```



```
public static RegexProperty uuid() {
    return new RegexProperty(UUID).asString();
}

public static RegexProperty isoDate() {
    return new RegexProperty(ANY_DATE).asString();
}

public static RegexProperty isoDateTime() {
    return new RegexProperty(ANY_DATE_TIME).asString();
}

public static RegexProperty isoTime() {
    return new RegexProperty(ANY_TIME).asString();
}

public static RegexProperty iso8601WithOffset() {
    return new RegexProperty(ISO8601_WITH_OFFSET).asString();
}

public static RegexProperty nonEmpty() {
    return new RegexProperty(NON_EMPTY).asString();
}

public static RegexProperty nonBlank() {
    return new RegexProperty(NON_BLANK).asString();
}
```

In your contract, you can use it as follows (example for the Groovy DSL):

```

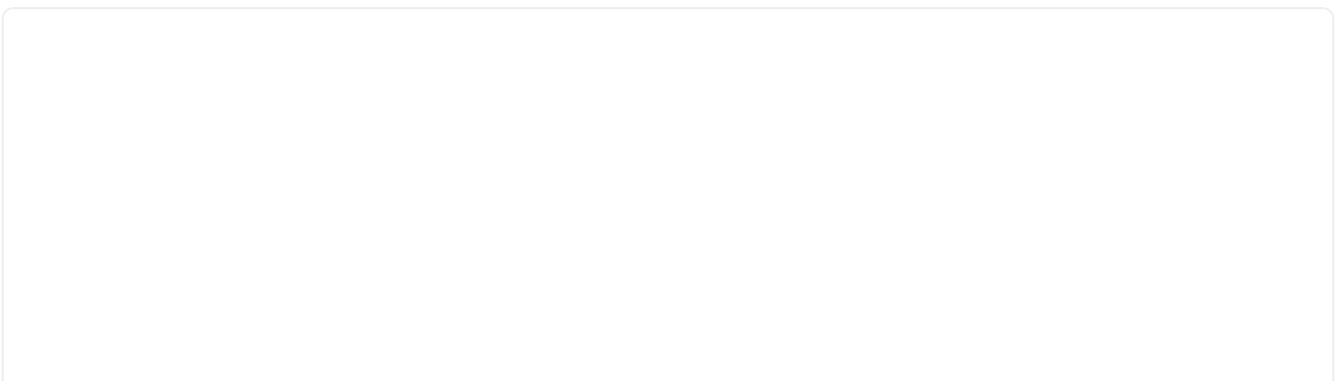
Contract dslWithOptionalsInString = Contract.make {
  priority 1
  request {
    method POST()
    url '/users/password'
    headers {
      contentType(applicationJson())
    }
    body(
      email: $(consumer(optional(regex(email()))), producer('abc@abc.com')),
      callback_url: $(consumer(regex(hostname()))),
      producer('http://partners.com')
    )
  }
  response {
    status 404
    headers {
      contentType(applicationJson())
    }
    body(
      code: value(consumer("123123"), producer(optional("123123"))),
      message: "User not found by email = [${value(producer(regex(email()))),
      consumer('not.existing@user.com'))}]"
    )
  }
}

```

To make matters even simpler, you can use a set of predefined objects that automatically assume that you want a regular expression to be passed. All of those methods start with the **any** prefix, as follows:

```
T anyAlphaUnicode();
T anyAlphaNumeric();
T anyNumber();
T anyInteger();
T anyPositiveInt();
T anyDouble();
T anyHex();
T aBoolean();
T anyIpAddress();
T anyHostname();
T anyEmail();
T anyUrl();
T anyHttpsUrl();
T anyUuid();
T anyDate();
T anyDateTime();
T anyTime();
T anyIso8601WithOffset();
T anyNonBlankString();
T anyNonEmptyString();
T anyOf(String... values);
```

The following example shows how you can reference those methods:



groovy

```
Contract contractDsl = Contract.make {
  name "foo"
  label 'trigger_event'
  input {
    triggeredBy('toString()')
  }
  outputMessage {
    sentTo 'topic.rateablequote'
    body([
      alpha          : $(anyAlphaUnicode()),
      number         : $(anyNumber()),
      anInteger      : $(anyInteger()),
      positiveInt    : $(anyPositiveInt()),
      aDouble        : $(anyDouble()),
      aBoolean       : $(aBoolean()),
      ip             : $(anyIpAddress()),
      hostname       : $(anyHostname()),
      email          : $(anyEmail()),
      url            : $(anyUrl()),
      httpsUrl       : $(anyHttpsUrl()),
      uuid           : $(anyUuid()),
      date           : $(anyDate()),
      dateTime       : $(anyDateTime()),
      time           : $(anyTime()),
      iso8601WithOffset: $(anyIso8601WithOffset()),
      nonBlankString : $(anyNonBlankString()),
      nonEmptyString : $(anyNonEmptyString()),
      anyOf          : $(anyOf('foo', 'bar'))
    ])
  }
}
```

```

contract {
    name = "foo"
    label = "trigger_event"
    input {
        triggeredBy = "toString()"
    }
    outputMessage {
        sentTo = sentTo("topic.rateablequote")
        body(mapOf(
            "alpha" to v(anyAlphaUnicode),
            "number" to v(anyNumber),
            "anInteger" to v(anyInteger),
            "positiveInt" to v(anyPositiveInt),
            "aDouble" to v(anyDouble),
            "aBoolean" to v(aBoolean),
            "ip" to v(anyIpAddress),
            "hostname" to v(anyAlphaUnicode),
            "email" to v(anyEmail),
            "url" to v(anyUrl),
            "httpsUrl" to v(anyHttpsUrl),
            "uuid" to v(anyUuid),
            "date" to v(anyDate),
            "dateTime" to v(anyDateTime),
            "time" to v(anyTime),
            "iso8601WithOffset" to v(anyIso8601WithOffset),
            "nonBlankString" to v(anyNonBlankString),
            "nonEmptyString" to v(anyNonEmptyString),
            "anyOf" to v(anyOf('foo', 'bar'))
        ))
        headers {
            header("Content-Type", "text/plain")
        }
    }
}

```

Limitations



Due to certain limitations of the [Xeger](#) library that generates a string out of a regex, do not use the `$` and `^` signs in your regex if you rely on automatic generation. See [Issue 899](#).



Do not use a `LocalDate` instance as a value for `$` (for example, `$(consumer(LocalDate.now()))`). It causes a `java.lang.StackOverflowError`. Use `$(consumer(LocalDate.now().toString()))` instead. See [Issue 900](#).

Passing Optional Parameters



This section is valid only for Groovy DSL. Check out the [Dynamic Properties in the Matchers Sections](#) section for YAML examples of a similar feature.

You can provide optional parameters in your contract. However, you can provide optional parameters only for the following:

- The STUB side of the Request
- The TEST side of the Response

The following example shows how to provide optional parameters:

groovy

```
org.springframework.cloud.contract.spec.Contract.make {
    priority 1
    name "optionals"
    request {
        method 'POST'
        url '/users/password'
        headers {
            contentType(applicationJson())
        }
        body(
            email: $(consumer(optional(regex(email())))),
            producer('abc@abc.com'),
            callback_url: $(consumer(regex(hostname()))),
            producer('https://partners.com'))
        )
    }
    response {
        status 404
        headers {
            header 'Content-Type': 'application/json'
        }
        body(
            code: value(consumer("123123"), producer(optional("123123")))
        )
    }
}
```

java

```
org.springframework.cloud.contract.spec.Contract.make(c -> {
    c.priority(1);
    c.name("optionals");
    c.request(r -> {
        r.method("POST");
        r.url("/users/password");
        r.headers(h -> {
            h.contentType(h.applicationJson());
        });
        r.body(ContractVerifierUtil.map()
            .entry("email",
                r.$(r.consumer(r.optional(r.regex(r.email()))),
                    r.producer("abc@abc.com")))
            .entry("callback_url", r.$(r.consumer(r.regex(r.hostname())),
                r.producer("https://partners.com"))));
    });
    c.response(r -> {
        r.status(404);
        r.headers(h -> {
            h.header("Content-Type", "application/json");
        });
        r.body(ContractVerifierUtil.map().entry("code", r.value(
            r.consumer("123123"), r.producer(r.optional("123123"))));
    });
});
```

kotlin

```
contract { c ->
    priority = 1
    name = "optionals"
    request {
        method = POST
        url = url("/users/password")
        headers {
            contentType = APPLICATION_JSON
        }
        body = body(mapOf(
            "email" to v(consumer(optional(regex(email))),
producer("abc@abc.com")),
            "callback_url" to v(consumer(regex(hostname)),
producer("https://partners.com"))
        ))
    }
    response {
        status = NOT_FOUND
        headers {
            header("Content-Type", "application/json")
        }
        body(mapOf(
            "code" to value(consumer("123123"), producer(optional("123123")))
        ))
    }
}
```

By wrapping a part of the body with the `optional()` method, you create a regular expression that must be present 0 or more times.

If you use Spock, the following test would be generated from the previous example:

groovy

```
        """\npackage com.example\n\nimport com.jayway.jsonpath.DocumentContext\nimport com.jayway.jsonpath.JsonPath\nimport spock.lang.Specification\nimport io.restassured.module.mockmvc.specification.MockMvcRequestSpecification\nimport io.restassured.response.ResponseOptions\n\nimport static\norg.springframework.cloud.contract.verifier.assertion.SpringCloudContractAssertion\ns.assertThat\nimport static\norg.springframework.cloud.contract.verifier.util.ContractVerifierUtil.*\nimport static com.toomuchcoding.jsonassert.JsonAssertion.assertThatJson\nimport static io.restassured.module.mockmvc.RestAssuredMockMvc.*\n\n@SuppressWarnings("rawtypes")\nclass FooSpec extends Specification {\n\n    \tdef validate_optionals() throws Exception {\n        \t\tgiven:\n        \t\t\tMockMvcRequestSpecification request = given()\n        \t\t\t\t.header("Content-Type", "application/json")\n        \t\t\t\t.body(''{ "email": "abc@abc.com", "callback_url": "https://partners.com"}''\n        \t\t\t\t)\n\n        \t\twhen:\n        \t\t\tResponseOptions response = given().spec(request)\n        \t\t\t\t.post("/users/password")\n\n        \t\tthen:\n        \t\t\tresponse.statusCode() == 404\n        \t\t\tresponse.header("Content-Type") == 'application/json'\n\n        \t\tand:\n        \t\t\tDocumentContext parsedJson = JsonPath.parse(response.body.asString())\n        \t\t\tassertThatJson(parsedJson).field("[ 'code' ]").matches("(123123)?")\n        \t\t\t\n    }\n\n}\n    """
```

The following stub would also be generated:

```

        ...
    {
        "request" : {
            "url" : "/users/password",
            "method" : "POST",
            "bodyPatterns" : [ {
                "matchesJsonPath" : "$[?(@['email'] =~ /([a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-
]+\\\\.\\\\[a-zA-Z]{2,6})?/)]"
            }, {
                "matchesJsonPath" : "$[?(@['callback_url'] =~
/((http[s]?|ftp):\\\\/\\\\/)?(?:[^\:\\\\/\\\\s]+)(:[0-9]{1,5})?/)]"
            } ],
            "headers" : {
                "Content-Type" : {
                    "equalTo" : "application/json"
                }
            }
        },
        "response" : {
            "status" : 404,
            "body" : "{\\\"code\\\":\\\"123123\\\",\\\"message\\\":\\\"User not found by email ==
[not.existing@user.com]\\\"}",
            "headers" : {
                "Content-Type" : "application/json"
            }
        },
        "priority" : 1
    }
    ...

```

Executing Custom Methods on the Server Side



This section is valid only for Groovy DSL. Check out the [Dynamic Properties in the Matchers Sections](#) section for YAML examples of a similar feature.

You can define a method call that runs on the server side during the test. Such a method can be added to the class defined as `baseClassForTests` in the configuration. The following code shows an example of the contract portion of the test case:

groovy

```
method GET()
```

java

```
r.method(r.GET());
```

kotlin

```
method = GET
```

The following code shows the base class portion of the test case:

```
abstract class BaseMockMvcSpec extends Specification {  
  
    def setup() {  
        RestAssuredMockMvc.standaloneSetup(new PairIdController())  
    }  
  
    void isProperCorrelationId(Integer correlationId) {  
        assert correlationId == 123456  
    }  
  
    void isEmpty(String value) {  
        assert value == null  
    }  
  
}
```



You cannot use both a `String` and `execute` to perform concatenation. For example, calling `header('Authorization', 'Bearer ' + execute('authToken()'))` leads to improper results. Instead, call `header('Authorization', execute('authToken()'))` and ensure that the `authToken()` method returns everything you need.

The type of the object read from the JSON can be one of the following, depending on the JSON path:

- **String**: If you point to a `String` value in the JSON.
- **JSONArray**: If you point to a `List` in the JSON.
- **Map**: If you point to a `Map` in the JSON.
- **Number**: If you point to `Integer`, `Double`, and other numeric type in the JSON.
- **Boolean**: If you point to a `Boolean` in the JSON.

In the request part of the contract, you can specify that the `body` should be taken from a method.



You must provide both the consumer and the producer side. The `execute` part is applied for the whole body, not for parts of it.

The following example shows how to read an object from JSON:

```
Contract contractDsl = Contract.make {
  request {
    method 'GET'
    url '/something'
    body(
      $(c('foo'), p(execute('hashCode()')))
    )
  }
  response {
    status OK()
  }
}
```

The preceding example results in calling the `hashCode()` method in the request body. It should resemble the following code:

```
// given:
MockMvcRequestSpecification request = given()
  .body(hashCode());

// when:
ResponseOptions response = given().spec(request)
  .get("/something");

// then:
assertThat(response.statusCode()).isEqualTo(200);
```

Referencing the Request from the Response

The best situation is to provide fixed values, but sometimes you need to reference a request in your response.

If you write contracts in the Groovy DSL, you can use the `fromRequest()` method, which lets you reference a bunch of elements from the HTTP request. You can use the following options:

- `fromRequest().url()`: Returns the request URL and query parameters.
- `fromRequest().query(String key)`: Returns the first query parameter with a given name.
- `fromRequest().query(String key, int index)`: Returns the nth query parameter with a given name.
- `fromRequest().path()`: Returns the full path.
- `fromRequest().path(int index)`: Returns the nth path element.

- `fromRequest().header(String key)`: Returns the first header with a given name.
- `fromRequest().header(String key, int index)`: Returns the nth header with a given name.
- `fromRequest().body()`: Returns the full request body.
- `fromRequest().body(String jsonPath)`: Returns the element from the request that matches the JSON Path.

If you use the YAML contract definition or the Java one, you have to use the [Handlebars {{{ }}}](#) notation with custom Spring Cloud Contract functions to achieve this. In that case, you can use the following options:

- `{{{ request.url }}}`: Returns the request URL and query parameters.
- `{{{ request.query.key.[index] }}}`: Returns the nth query parameter with a given name. For example, for a key of `thing`, the first entry is `{{{ request.query.thing.[0] }}}`
- `{{{ request.path }}}`: Returns the full path.
- `{{{ request.path.[index] }}}`: Returns the nth path element. For example, the first entry is `'{{{ request.path.[0] }}}'`
- `{{{ request.headers.key }}}`: Returns the first header with a given name.
- `{{{ request.headers.key.[index] }}}`: Returns the nth header with a given name.
- `{{{ request.body }}}`: Returns the full request body.
- `{{{ jsonpath this 'your.json.path' }}}`: Returns the element from the request that matches the JSON Path. For example, for a JSON path of `$.here`, use `{{{ jsonpath this '$.here' }}}`

Consider the following contract:

groovy

```
Contract contractDsl = Contract.make {
    request {
        method 'GET'
        url('/api/v1/xxxx') {
            queryParameters {
                parameter('foo', 'bar')
                parameter('foo', 'bar2')
            }
        }
        headers {
            header(authorization(), 'secret')
            header(authorization(), 'secret2')
        }
        body(foo: 'bar', baz: 5)
    }
    response {
        status OK()
        headers {
            header(authorization(), "foo ${fromRequest().header(authorization())}")
        }
    }
}
```

```

bar")
    }
    body(
        url: fromRequest().url(),
        path: fromRequest().path(),
        pathIndex: fromRequest().path(1),
        param: fromRequest().query('foo'),
        paramIndex: fromRequest().query('foo', 1),
        authorization: fromRequest().header('Authorization'),
        authorization2: fromRequest().header('Authorization', 1),
        fullBody: fromRequest().body(),
        responseFoo: fromRequest().body('$.foo'),
        responseBaz: fromRequest().body('$.baz'),
        responseBaz2: "Bla bla ${fromRequest().body('$.foo')} bla bla",
        rawUrl: fromRequest().rawUrl(),
        rawPath: fromRequest().rawPath(),
        rawPathIndex: fromRequest().rawPath(1),
        rawParam: fromRequest().rawQuery('foo'),
        rawParamIndex: fromRequest().rawQuery('foo', 1),
        rawAuthorization: fromRequest().rawHeader('Authorization'),
        rawAuthorization2: fromRequest().rawHeader('Authorization', 1),
        rawResponseFoo: fromRequest().rawBody('$.foo'),
        rawResponseBaz: fromRequest().rawBody('$.baz'),
        rawResponseBaz2: "Bla bla ${fromRequest().rawBody('$.foo')} bla
bla"
    )
}
}
Contract contractDsl = Contract.make {
    request {
        method 'GET'
        url('/api/v1/xxxx') {
            queryParameters {
                parameter('foo', 'bar')
                parameter('foo', 'bar2')
            }
        }
        headers {
            header(authorization(), 'secret')
            header(authorization(), 'secret2')
        }
        body(foo: "bar", baz: 5)
    }
    response {
        status OK()
        headers {
            contentType(applicationJson())
        }
        body(''
            {
                "responseFoo": "{${fromRequest().body('$.foo')}}",

```

```

        "responseBaz": {{{ jsonPath request.body '$.baz' }}}},
        "responseBaz2": "Bla bla {{{ jsonPath request.body '$.foo' }}}
bla bla"
    }
    '''.toString()
}
}

```

yaml

```

request:
  method: GET
  url: /api/v1/xxxx
  queryParameters:
    foo:
      - bar
      - bar2
  headers:
    Authorization:
      - secret
      - secret2
  body:
    foo: bar
    baz: 5
response:
  status: 200
  headers:
    Authorization: "foo {{{ request.headers.Authorization.0 }}} bar"
  body:
    url: "{{{ request.url }}"
    path: "{{{ request.path }}"
    pathIndex: "{{{ request.path.1 }}"
    param: "{{{ request.query.foo }}"
    paramIndex: "{{{ request.query.foo.1 }}"
    authorization: "{{{ request.headers.Authorization.0 }}"
    authorization2: "{{{ request.headers.Authorization.1 }}"
    fullBody: "{{{ request.body }}"
    responseFoo: "{{{ jsonpath this '$.foo' }}"
    responseBaz: "{{{ jsonpath this '$.baz' }}"
    responseBaz2: "Bla bla {{{ jsonpath this '$.foo' }}} bla bla"

```

java

```
package contracts.beer.rest;

import java.util.function.Supplier;

import org.springframework.cloud.contract.spec.Contract;

import static
org.springframework.cloud.contract.verifier.util.ContractVerifierUtil.map;

class shouldReturnStatsForAUser implements Supplier<Contract> {

    @Override
    public Contract get() {
        return Contract.make(c -> {
            c.request(r -> {
                r.method("POST");
                r.url("/stats");
                r.body(map().entry("name", r.anyAlphaUnicode()));
                r.headers(h -> {
                    h.contentType(h.applicationJson());
                });
            });
            c.response(r -> {
                r.status(r.OK());
                r.body(map()
                    .entry("text",
                        "Dear {{{jsonPath request.body '$.name'}}} thanks
for your interested in drinking beer")
                    .entry("quantity", r.$(r.c(5), r.p(r.anyNumber()))));
                r.headers(h -> {
                    h.contentType(h.applicationJson());
                });
            });
        });
    }
}
```


kotlin

```
package contracts.beer.rest

import org.springframework.cloud.contract.spec.ContractDsl.Companion.contract

contract {
    request {
        method = method("POST")
        url = url("/stats")
        body(mapOf(
            "name" to anyAlphaUnicode
        ))
        headers {
            contentType = APPLICATION_JSON
        }
    }
    response {
        status = OK
        body(mapOf(
            "text" to "Don't worry ${fromRequest().body("$.name")} thanks for your
interested in drinking beer",
            "quantity" to v(c(5), p(anyNumber))
        ))
        headers {
            contentType = fromRequest().header(CONTENT_TYPE)
        }
    }
}
```

Running a JUnit test generation leads to a test that resembles the following example:

```

// given:
MockMvcRequestSpecification request = given()
    .header("Authorization", "secret")
    .header("Authorization", "secret2")
    .body("{\"foo\":\"bar\",\"baz\":5}");

// when:
ResponseOptions response = given().spec(request)
    .queryParam("foo", "bar")
    .queryParam("foo", "bar2")
    .get("/api/v1/xxxx");

// then:
assertThat(response.statusCode()).isEqualTo(200);
assertThat(response.header("Authorization")).isEqualTo("foo secret bar");
// and:
DocumentContext parsedJson = JsonPath.parse(response.getBody().asString());
assertThatJson(parsedJson).field("[ 'fullBody' ]").isEqualTo("{\"foo\":\"bar\",\"baz\":5}");
assertThatJson(parsedJson).field("[ 'authorization' ]").isEqualTo("secret");
assertThatJson(parsedJson).field("[ 'authorization2' ]").isEqualTo("secret2");
assertThatJson(parsedJson).field("[ 'path' ]").isEqualTo("/api/v1/xxxx");
assertThatJson(parsedJson).field("[ 'param' ]").isEqualTo("bar");
assertThatJson(parsedJson).field("[ 'paramIndex' ]").isEqualTo("bar2");
assertThatJson(parsedJson).field("[ 'pathIndex' ]").isEqualTo("v1");
assertThatJson(parsedJson).field("[ 'responseBaz' ]").isEqualTo(5);
assertThatJson(parsedJson).field("[ 'responseFoo' ]").isEqualTo("bar");
assertThatJson(parsedJson).field("[ 'url' ]").isEqualTo("/api/v1/xxxx?foo=bar&foo=bar2"
);
assertThatJson(parsedJson).field("[ 'responseBaz2' ]").isEqualTo("Bla bla bar bla
bla");

```

As you can see, elements from the request have been properly referenced in the response.

The generated WireMock stub should resemble the following example:

```

{
  "request" : {
    "urlPath" : "/api/v1/xxxx",
    "method" : "POST",
    "headers" : {
      "Authorization" : {
        "equalTo" : "secret2"
      }
    },
    "queryParameters" : {
      "foo" : {
        "equalTo" : "bar2"
      }
    },
    "bodyPatterns" : [ {
      "matchesJsonPath" : "$[?(@.['baz'] == 5)]"
    }, {
      "matchesJsonPath" : "$[?(@.['foo'] == 'bar')]"
    } ]
  },
  "response" : {
    "status" : 200,
    "body" :
    "{\\"authorization\\":\\"{{{request.headers.Authorization.[0]}}}\",\\"path\\":\\"{{{request.
    path}}}\",\\"responseBaz\\":\\"{{{jsonpath this '$.baz'}}}
    ,\\"param\\":\\"{{{request.query.foo.[0]}}}\",\\"pathIndex\\":\\"{{{request.path.[1]}}}\",\\"
    responseBaz2\\":\\"Bla bla {{{jsonpath this '$.foo'}}} bla
    bla\\",\\"responseFoo\\":\\"{{{jsonpath this
    '$.foo'}}}\",\\"authorization2\\":\\"{{{request.headers.Authorization.[1]}}}\",\\"fullBody
    \\":\\"{{{escapejsonbody}}}\",\\"url\\":\\"{{{request.url}}}\",\\"paramIndex\\":\\"{{{request.
    query.foo.[1]}}}\\"},
    "headers" : {
      "Authorization" : "{{{{request.headers.Authorization.[0]}}};foo"
    },
    "transformers" : [ "response-template" ]
  }
}

```

Sending a request such as the one presented in the **request** part of the contract results in sending the following response body:

```

{
  "url" : "/api/v1/xxxx?foo=bar&foo=bar2",
  "path" : "/api/v1/xxxx",
  "pathIndex" : "v1",
  "param" : "bar",
  "paramIndex" : "bar2",
  "authorization" : "secret",
  "authorization2" : "secret2",
  "fullBody" : "{\"foo\": \"bar\", \"baz\": 5}",
  "responseFoo" : "bar",
  "responseBaz" : 5,
  "responseBaz2" : "Bla bla bar bla bla"
}

```



This feature works only with WireMock versions greater than or equal to 2.5.1. The Spring Cloud Contract Verifier uses WireMock's `response-template` response transformer. It uses Handlebars to convert the Mustache `{{{ }}}` templates into proper values. Additionally, it registers two helper functions:

- `escapejsonbody`: Escapes the request body in a format that can be embedded in a JSON.
- `jsonpath`: For a given parameter, find an object in the request body.

Dynamic Properties in the Matchers Sections

If you work with [Pact](#), the following discussion may seem familiar. Quite a few users are used to having a separation between the body and setting the dynamic parts of a contract.

You can use the `bodyMatchers` section for two reasons:

- Define the dynamic values that should end up in a stub. You can set it in the `request` or `inputMessage` part of your contract.
- Verify the result of your test. This section is present in the `response` or `outputMessage` side of the contract.

Currently, Spring Cloud Contract Verifier supports only JSON path-based matchers with the following matching possibilities:

Coded DSL

- For the stubs (in tests on the consumer's side):
 - `byEquality()`: The value taken from the consumer's request in the provided JSON path must be equal to the value provided in the contract.
 - `byRegex(...)`: The value taken from the consumer's request in the provided JSON path must match the regex. You can also pass the type of the expected matched value (for example, `asString()`, `asLong()`, and so on).
 - `byDate()`: The value taken from the consumer's request in the provided JSON path must match the regex for an ISO Date value.

- `byTimestamp()`: The value taken from the consumer's request in the provided JSON path must match the regex for an ISO DateTime value.
- `byTime()`: The value taken from the consumer's request in the provided JSON path must match the regex for an ISO Time value.
- For the verification (in generated tests on the Producer's side):
 - `byEquality()`: The value taken from the producer's response in the provided JSON path must be equal to the provided value in the contract.
 - `byRegex(...)`: The value taken from the producer's response in the provided JSON path must match the regex.
 - `byDate()`: The value taken from the producer's response in the provided JSON path must match the regex for an ISO Date value.
 - `byTimestamp()`: The value taken from the producer's response in the provided JSON path must match the regex for an ISO DateTime value.
 - `byTime()`: The value taken from the producer's response in the provided JSON path must match the regex for an ISO Time value.
 - `byType()`: The value taken from the producer's response in the provided JSON path needs to be of the same type as the type defined in the body of the response in the contract. `byType` can take a closure, in which you can set `minOccurrence` and `maxOccurrence`. For the request side, you should use the closure to assert size of the collection. That way, you can assert the size of the flattened collection. To check the size of an unflattened collection, use a custom method with the `byCommand(...)` `testMatcher`.
 - `byCommand(...)`: The value taken from the producer's response in the provided JSON path is passed as an input to the custom method that you provide. For example, `byCommand('thing($it)')` results in calling a `thing` method to which the value matching the JSON Path gets passed. The type of the object read from the JSON can be one of the following, depending on the JSON path:
 - `String`: If you point to a `String` value.
 - `JSONArray`: If you point to a `List`.
 - `Map`: If you point to a `Map`.
 - `Number`: If you point to `Integer`, `Double`, or another kind of number.
 - `Boolean`: If you point to a `Boolean`.
 - `byNull()`: The value taken from the response in the provided JSON path must be null.

YAML



See the Groovy section for detailed explanation of what the types mean.

For YAML, the structure of a matcher resembles the following example:

```
- path: $.thing1
  type: by_regex
  value: thing2
  regexType: as_string
```

Alternatively, if you want to use one of the predefined regular expressions [`only_alpha_unicode`, `number`, `any_boolean`, `ip_address`, `hostname`, `email`, `url`, `uuid`, `iso_date`, `iso_date_time`, `iso_time`, `iso_8601_with_offset`, `non_empty`, `non_blank`], you can use something similar to the following example:

```
- path: $.thing1
  type: by_regex
  predefined: only_alpha_unicode
```

The following list shows the allowed list of `type` values:

- For `stubMatchers`:
 - `by_equality`
 - `by_regex`
 - `by_date`
 - `by_timestamp`
 - `by_time`
 - `by_type`
 - Two additional fields (`minOccurrence` and `maxOccurrence`) are accepted.
- For `testMatchers`:
 - `by_equality`
 - `by_regex`
 - `by_date`
 - `by_timestamp`
 - `by_time`
 - `by_type`
 - Two additional fields (`minOccurrence` and `maxOccurrence`) are accepted.
 - `by_command`
 - `by_null`

You can also define which type the regular expression corresponds to in the `regexType` field. The following list shows the allowed regular expression types:

- `as_integer`
- `as_double`
- `as_float`
- `as_long`
- `as_short`

- `as_boolean`
- `as_string`

Consider the following example:

groovy

```
Contract contractDsl = Contract.make {
  request {
    method 'GET'
    urlPath '/get'
    body([
      duck           : 123,
      alpha          : 'abc',
      number         : 123,
      aBoolean       : true,
      date           : '2017-01-01',
      dateTime       : '2017-01-01T01:23:45',
      time           : '01:02:34',
      valueWithoutAMatcher: 'foo',
      valueWithTypeMatch : 'string',
      key            : [
        'complex.key': 'foo'
      ]
    ])
  }
  bodyMatchers {
    jsonPath('$.duck', byRegex("[0-9]{3}").asInteger())
    jsonPath('$.duck', byEquality())
    jsonPath('$.alpha', byRegex(onlyAlphaUnicode()).asString())
    jsonPath('$.alpha', byEquality())
    jsonPath('$.number', byRegex(number()).asInteger())
    jsonPath('$.aBoolean', byRegex(anyBoolean()).asBooleanType())
    jsonPath('$.date', byDate())
    jsonPath('$.dateTime', byTimestamp())
    jsonPath('$.time', byTime())
    jsonPath("\$.['key'].['complex.key']", byEquality())
  }
  headers {
    contentType(applicationJson())
  }
}
response {
  status OK()
  body([
    duck           : 123,
    alpha          : 'abc',
    number         : 123,
    positiveInteger : 1234567890,
    negativeInteger : -1234567890,
    positiveDecimalNumber: 123.4567890,
```

```

        negativeDecimalNumber: -123.4567890,
        aBoolean                : true,
        date                    : '2017-01-01',
        dateTime                 : '2017-01-01T01:23:45',
        time                     : "01:02:34",
        valueWithoutAMatcher    : 'foo',
        valueWithTypeMatch      : 'string',
        valueWithMin             : [
            1, 2, 3
        ],
        valueWithMax             : [
            1, 2, 3
        ],
        valueWithMinMax         : [
            1, 2, 3
        ],
        valueWithMinEmpty       : [],
        valueWithMaxEmpty       : [],
        key                      : [
            'complex.key': 'foo'
        ],
        nullValue                : null
    ])
    bodyMatchers {
        // asserts the jsonpath value against manual regex
        jsonPath('$.duck', byRegex("[0-9]{3}").asInteger())
        // asserts the jsonpath value against the provided value
        jsonPath('$.duck', byEquality())
        // asserts the jsonpath value against some default regex
        jsonPath('$.alpha', byRegex(onlyAlphaUnicode()).asString())
        jsonPath('$.alpha', byEquality())
        jsonPath('$.number', byRegex(number()).asInteger())
        jsonPath('$.positiveInteger', byRegex(anInteger()).asInteger())
        jsonPath('$.negativeInteger', byRegex(anInteger()).asInteger())
        jsonPath('$.positiveDecimalNumber', byRegex(aDouble()).asDouble())
        jsonPath('$.negativeDecimalNumber', byRegex(aDouble()).asDouble())
        jsonPath('$.aBoolean', byRegex(anyBoolean()).asBooleanType())
        // asserts vs inbuilt time related regex
        jsonPath('$.date', byDate())
        jsonPath('$.dateTime', byTimestamp())
        jsonPath('$.time', byTime())
        // asserts that the resulting type is the same as in response body
        jsonPath('$.valueWithTypeMatch', byType())
        jsonPath('$.valueWithMin', byType {
            // results in verification of size of array (min 1)
            minOccurrence(1)
        })
        jsonPath('$.valueWithMax', byType {
            // results in verification of size of array (max 3)
            maxOccurrence(3)
        })
    }
}

```



```

    jsonPath('$.valueWithMinMax', byType {
        // results in verification of size of array (min 1 & max 3)
        minOccurrence(1)
        maxOccurrence(3)
    })
    jsonPath('$.valueWithMinEmpty', byType {
        // results in verification of size of array (min 0)
        minOccurrence(0)
    })
    jsonPath('$.valueWithMaxEmpty', byType {
        // results in verification of size of array (max 0)
        maxOccurrence(0)
    })
    // will execute a method `assertThatValueIsANumber`
    jsonPath('$.duck', byCommand('assertThatValueIsANumber($it)'))
    jsonPath("$.['key'].['complex.key']", byEquality())
    jsonPath('$.nullValue', byNull())
}
headers {
    contentType(applicationJson())
    header('Some-Header', $(c('someValue'), p(regex('[a-zA-Z]{9}'))))
}
}
}

```

yml

```

request:
  method: GET
  urlPath: /get/1
  headers:
    Content-Type: application/json
  cookies:
    foo: 2
    bar: 3
  queryParameters:
    limit: 10
    offset: 20
    filter: 'email'
    sort: name
    search: 55
    age: 99
    name: John.Doe
    email: 'bob@email.com'
  body:
    duck: 123
    alpha: "abc"
    number: 123
    aBoolean: true
    date: "2017-01-01"
    dateTime: "2017-01-01T01:23:45"

```

```
time: "01:02:34"
valueWithoutAMatcher: "foo"
valueWithTypeMatch: "string"
key:
  "complex.key": 'foo'
nullValue: null
valueWithMin:
  - 1
  - 2
  - 3
valueWithMax:
  - 1
  - 2
  - 3
valueWithMinMax:
  - 1
  - 2
  - 3
valueWithMinEmpty: []
valueWithMaxEmpty: []
matchers:
  url:
    regex: /get/[0-9]
    # predefined:
    # execute a method
    #command: 'equals($it)'
  queryParameters:
    - key: limit
      type: equal_to
      value: 20
    - key: offset
      type: containing
      value: 20
    - key: sort
      type: equal_to
      value: name
    - key: search
      type: not_matching
      value: '^[0-9]{2}$'
    - key: age
      type: not_matching
      value: '^\\w*$'
    - key: name
      type: matching
      value: 'John.*'
    - key: hello
      type: absent
  cookies:
    - key: foo
      regex: '[0-9]'
    - key: bar
```

```
    command: 'equals($it)'  
headers:  
  - key: Content-Type  
    regex: "application/json.*"  
body:  
  - path: $.duck  
    type: by_regex  
    value: "[0-9]{3}"  
  - path: $.duck  
    type: by_equality  
  - path: $.alpha  
    type: by_regex  
    predefined: only_alpha_unicode  
  - path: $.alpha  
    type: by_equality  
  - path: $.number  
    type: by_regex  
    predefined: number  
  - path: $.aBoolean  
    type: by_regex  
    predefined: any_boolean  
  - path: $.date  
    type: by_date  
  - path: $.dateTime  
    type: by_timestamp  
  - path: $.time  
    type: by_time  
  - path: "$.['key']['complex.key']"  
    type: by_equality  
  - path: $.nullvalue  
    type: by_null  
  - path: $.valueWithMin  
    type: by_type  
    minOccurrence: 1  
  - path: $.valueWithMax  
    type: by_type  
    maxOccurrence: 3  
  - path: $.valueWithMinMax  
    type: by_type  
    minOccurrence: 1  
    maxOccurrence: 3  
response:  
  status: 200  
  cookies:  
    foo: 1  
    bar: 2  
  body:  
    duck: 123  
    alpha: "abc"  
    number: 123  
    aBoolean: true
```

```
date: "2017-01-01"
dateTime: "2017-01-01T01:23:45"
time: "01:02:34"
valueWithoutAMatcher: "foo"
valueWithTypeMatch: "string"
valueWithMin:
  - 1
  - 2
  - 3
valueWithMax:
  - 1
  - 2
  - 3
valueWithMinMax:
  - 1
  - 2
  - 3
valueWithMinEmpty: []
valueWithMaxEmpty: []
key:
  'complex.key': 'foo'
nullValue: null
matchers:
  headers:
    - key: Content-Type
      regex: "application/json.*"
  cookies:
    - key: foo
      regex: '[0-9]'
    - key: bar
      command: 'equals($it)'
body:
  - path: $.duck
    type: by_regex
    value: "[0-9]{3}"
  - path: $.duck
    type: by_equality
  - path: $.alpha
    type: by_regex
    predefined: only_alpha_unicode
  - path: $.alpha
    type: by_equality
  - path: $.number
    type: by_regex
    predefined: number
  - path: $.aBoolean
    type: by_regex
    predefined: any_boolean
  - path: $.date
    type: by_date
  - path: $.dateTime
```

```

    type: by_timestamp
  - path: $.time
    type: by_time
  - path: $.valueWithTypeMatch
    type: by_type
  - path: $.valueWithMin
    type: by_type
    minOccurrence: 1
  - path: $.valueWithMax
    type: by_type
    maxOccurrence: 3
  - path: $.valueWithMinMax
    type: by_type
    minOccurrence: 1
    maxOccurrence: 3
  - path: $.valueWithMinEmpty
    type: by_type
    minOccurrence: 0
  - path: $.valueWithMaxEmpty
    type: by_type
    maxOccurrence: 0
  - path: $.duck
    type: by_command
    value: assertThatValueIsANumber($it)
  - path: $.nullValue
    type: by_null
    value: null
headers:
  Content-Type: application/json

```

In the preceding example, you can see the dynamic portions of the contract in the `matchers` sections. For the request part, you can see that, for all fields but `valueWithoutAMatcher`, the values of the regular expressions that the stub should contain are explicitly set. For the `valueWithoutAMatcher`, the verification takes place in the same way as without the use of matchers. In that case, the test performs an equality check.

For the response side in the `bodyMatchers` section, we define the dynamic parts in a similar manner. The only difference is that the `byType` matchers are also present. The verifier engine checks four fields to verify whether the response from the test has a value for which the JSON path matches the given field, is of the same type as the one defined in the response body, and passes the following check (based on the method being called):

- For `$.valueWithTypeMatch`, the engine checks whether the type is the same.
- For `$.valueWithMin`, the engine checks the type and asserts whether the size is greater than or equal to the minimum occurrence.
- For `$.valueWithMax`, the engine checks the type and asserts whether the size is smaller than or equal to the maximum occurrence.
- For `$.valueWithMinMax`, the engine checks the type and asserts whether the size is between the

minimum and maximum occurrence.

The resulting test resembles the following example (note that an `and` section separates the autogenerated assertions and the assertion from matchers):

```
// given:
MockMvcRequestSpecification request = given()
    .header("Content-Type", "application/json")

    .body("{\"duck\":123,\"alpha\":\"abc\",\"number\":123,\"aBoolean\":true,\"date\":\"2017-01-01\",\"dateTime\":\"2017-01-01T01:23:45\",\"time\":\"01:02:34\",\"valueWithoutAMatcher\":\"foo\",\"valueWithTypeMatch\":\"string\",\"key\":{\"complex.key\":\"foo\"}}");

// when:
ResponseOptions response = given().spec(request)
    .get("/get");

// then:
assertThat(response.statusCode()).isEqualTo(200);
assertThat(response.header("Content-Type")).matches("application/json.*");
// and:
DocumentContext parsedJson = JsonPath.parse(response.getBody().asString());
assertThatJson(parsedJson).field("'valueWithoutAMatcher']").isEqualTo("foo");
// and:
assertThat(parsedJson.read("$.duck", String.class)).matches("[0-9]{3}");
assertThat(parsedJson.read("$.duck", Integer.class)).isEqualTo(123);
assertThat(parsedJson.read("$.alpha", String.class)).matches("[\\p{L}]*");
assertThat(parsedJson.read("$.alpha", String.class)).isEqualTo("abc");
assertThat(parsedJson.read("$.number", String.class)).matches("-?((\\d*\\.\\d+|\\d+))");
assertThat(parsedJson.read("$.aBoolean", String.class)).matches("(true|false)");
assertThat(parsedJson.read("$.date", String.class)).matches("(\\d\\d\\d\\d\\d)-(0[1-9]|1[012])-(0[1-9]|[12][0-9]|3[01])");
assertThat(parsedJson.read("$.dateTime", String.class)).matches("([0-9]{4})-(1[0-2]|0[1-9])-(3[01]|0[1-9]|[12][0-9])T(2[0-3]|[01][0-9]):([0-5][0-9]):([0-5][0-9])");
assertThat(parsedJson.read("$.time", String.class)).matches("(2[0-3]|[01][0-9]):([0-5][0-9]):([0-5][0-9])");
assertThat((Object)
    parsedJson.read("$.valueWithTypeMatch")).isInstanceOf(java.lang.String.class);
assertThat((Object)
    parsedJson.read("$.valueWithMin")).isInstanceOf(java.util.List.class);
assertThat((java.lang.Iterable) parsedJson.read("$.valueWithMin",
    java.util.Collection.class)).as("$.valueWithMin").hasSizeGreaterThanOrEqualTo(1);
assertThat((Object)
    parsedJson.read("$.valueWithMax")).isInstanceOf(java.util.List.class);
assertThat((java.lang.Iterable) parsedJson.read("$.valueWithMax",
    java.util.Collection.class)).as("$.valueWithMax").hasSizeLessThanOrEqualTo(3);
assertThat((Object)
    parsedJson.read("$.valueWithMinMax")).isInstanceOf(java.util.List.class);
```

```

assertThat((java.lang.Iterable) parsedJson.read("$.valueWithMinMax",
java.util.Collection.class)).as("$.valueWithMinMax").hasSizeBetween(1, 3);
assertThat((Object)
parsedJson.read("$.valueWithMinEmpty")).isInstanceOf(java.util.List.class);
assertThat((java.lang.Iterable) parsedJson.read("$.valueWithMinEmpty",
java.util.Collection.class)).as("$.valueWithMinEmpty").hasSizeGreaterThanOrEqualTo(0);
assertThat((Object)
parsedJson.read("$.valueWithMaxEmpty")).isInstanceOf(java.util.List.class);
assertThat((java.lang.Iterable) parsedJson.read("$.valueWithMaxEmpty",
java.util.Collection.class)).as("$.valueWithMaxEmpty").hasSizeLessThanOrEqualTo(0);
assertThatValueIsANumber(parsedJson.read("$.duck"));
assertThat(parsedJson.read("$.['key']['complex.key']",
String.class)).isEqualTo("foo");

```



Notice that, for the `byCommand` method, the example calls the `assertThatValueIsANumber`. This method must be defined in the test base class or be statically imported to your tests. Notice that the `byCommand` call was converted to `assertThatValueIsANumber(parsedJson.read("$.duck"))`; That means that the engine took the method name and passed the proper JSON path as a parameter to it.

The resulting WireMock stub is in the following example:

```

'''
{
  "request" : {
    "urlPath" : "/get",
    "method" : "POST",
    "headers" : {
      "Content-Type" : {
        "matches" : "application/json.*"
      }
    },
    "bodyPatterns" : [ {
      "matchesJsonPath" : "$.['list']['some']['nested'][?(@.['anothervalue'] == 4)]"
    }, {
      "matchesJsonPath" : "$[?(@.['valueWithoutAMatcher'] == 'foo')]"
    }, {
      "matchesJsonPath" : "$[?(@.['valueWithTypeMatch'] == 'string')]"
    }, {
      "matchesJsonPath" : "$.['list']['someother']['nested'][?(@.['json'] == 'with
value')]"
    }, {
      "matchesJsonPath" : "$.['list']['someother']['nested'][?(@.['anothervalue'] ==
4)]"
    }, {
      "matchesJsonPath" : "$[?(@.duck =~ /[0-9]{3})/]"
    }, {
      "matchesJsonPath" : "$[?(@.duck == 123)]"
    }, {

```

```

    "matchesJsonPath" : "$[?(@.alpha =~ /([\p{L}]+)/)]"
  }, {
    "matchesJsonPath" : "$[?(@.alpha == 'abc')]"
  }, {
    "matchesJsonPath" : "$[?(@.number =~ /(-?(\\d*\\.\\d+|\\d+)))/]"
  }, {
    "matchesJsonPath" : "$[?(@.aBoolean =~ /((true|false))/)]"
  }, {
    "matchesJsonPath" : "$[?(@.date =~ /((\\d\\d\\d\\d)-(0[1-9]|1[012])-(0[1-9]|12)[0-9]|3[01]))/)]"
  }, {
    "matchesJsonPath" : "$[?(@.dateTime =~ /(([0-9]{4})-(1[0-2]|0[1-9])-(3[01]|0[1-9]|12)[0-9])T(2[0-3]|[01][0-9]):([0-5][0-9]):([0-5][0-9]))/)]"
  }, {
    "matchesJsonPath" : "$[?(@.time =~ /((2[0-3]|[01][0-9]):([0-5][0-9]):([0-5][0-9]))/)]"
  }, {
    "matchesJsonPath" : "$.list.some.nested[?(@.json =~ /(.*)/)]"
  }, {
    "matchesJsonPath" : "$[?(@.valueWithMin.size() >= 1)]"
  }, {
    "matchesJsonPath" : "$[?(@.valueWithMax.size() <= 3)]"
  }, {
    "matchesJsonPath" : "$[?(@.valueWithMinMax.size() >= 1 && @.valueWithMinMax.size() <= 3)]"
  }, {
    "matchesJsonPath" : "$[?(@.valueWithOccurrence.size() >= 4 && @.valueWithOccurrence.size() <= 4)]"
  } ]
},
"response" : {
  "status" : 200,
  "body" :
"{\\"duck\\":123,\\"alpha\\":\\"abc\\",\\"number\\":123,\\"aBoolean\\":true,\\"date\\":\\"2017-01-01\\",\\"dateTime\\":\\"2017-01-01T01:23:45\\",\\"time\\":\\"01:02:34\\",\\"valueWithoutAMatcher\\":\\"foo\\",\\"valueWithTypeMatch\\":\\"string\\",\\"valueWithMin\\":[1,2,3],\\"valueWithMax\\":[1,2,3],\\"valueWithMinMax\\":[1,2,3],\\"valueWithOccurrence\\":[1,2,3,4]}",
  "headers" : {
    "Content-Type" : "application/json"
  },
  "transformers" : [ "response-template" ]
}
}
'''

```



If you use a **matcher**, the part of the request and response that the **matcher** addresses with the JSON Path gets removed from the assertion. In the case of verifying a collection, you must create matchers for **all** the elements of the collection.

Consider the following example:

```
Contract.make {
  request {
    method 'GET'
    url("/foo")
  }
  response {
    status OK()
    body(events: [[
      operation      : 'EXPORT',
      eventId        : '16f1ed75-0bcc-4f0d-a04d-
3121798faf99',
      status         : 'OK'
    ], [
      operation      : 'INPUT_PROCESSING',
      eventId        : '3bb4ac82-6652-462f-b6d1-
75e424a0024a',
      status         : 'OK'
    ]
  ])
  )
  bodyMatchers {
    jsonPath('$.events[0].operation', byRegex('.+'))
    jsonPath('$.events[0].eventId', byRegex('^([a-fA-F0-9]{8}-[a-fA-F0-9]{4}-
[a-fA-F0-9]{4}-[a-fA-F0-9]{4}-[a-fA-F0-9]{12})$'))
    jsonPath('$.events[0].status', byRegex('.+'))
  }
}
```

The preceding code leads to creating the following test (the code block shows only the assertion section):

and:

```
DocumentContext parsedJson = JsonPath.parse(response.body.asString())
```

```
assertThatJson(parsedJson).array(["'events'"]).contains(["'eventId'"]).isEqualTo("16f1ed75-0bcc-4f0d-a04d-3121798faf99")
```

```
assertThatJson(parsedJson).array(["'events'"]).contains(["'operation'"]).isEqualTo("EXPORT")
```

```
assertThatJson(parsedJson).array(["'events'"]).contains(["'operation'"]).isEqualTo("INPUT_PROCESSING")
```

```
assertThatJson(parsedJson).array(["'events'"]).contains(["'eventId'"]).isEqualTo("3bb4ac82-6652-462f-b6d1-75e424a0024a")
```

```
assertThatJson(parsedJson).array(["'events'"]).contains(["'status'"]).isEqualTo("OK")
```

and:

```
assertThat(parsedJson.read("\$.events[0].operation", String.class)).matches(".+")
```

```
assertThat(parsedJson.read("\$.events[0].eventId", String.class)).matches("^([a-fA-F0-9]{8}-[a-fA-F0-9]{4}-[a-fA-F0-9]{4}-[a-fA-F0-9]{4}-[a-fA-F0-9]{12})\\$")
```

```
assertThat(parsedJson.read("\$.events[0].status", String.class)).matches(".+")
```

As you can see, the assertion is malformed. Only the first element of the array got asserted. In order to fix this, you should apply the assertion to the whole `$.events` collection and assert it with the `byCommand(...)` method.

Asynchronous Support

If you use asynchronous communication on the server side (your controllers are returning `Callable`, `DeferredResult`, and so on), then, inside your contract, you must provide an `async()` method in the `response` section. The following code shows an example:

groovy

```
org.springframework.cloud.contract.spec.Contract.make {
    request {
        method GET()
        url '/get'
    }
    response {
        status OK()
        body 'Passed'
        async()
    }
}
```

yaml

```
response:
  async: true
```

java

```
class contract implements Supplier<Collection<Contract>> {

    @Override
    public Collection<Contract> get() {
        return Collections.singletonList(Contract.make(c -> {
            c.request(r -> {
                // ...
            });
            c.response(r -> {
                r.async();
                // ...
            });
        }));
    }
}
```

kotlin

```
import org.springframework.cloud.contract.spec.ContractDsl.Companion.contract

contract {
    request {
        // ...
    }
    response {
        async = true
        // ...
    }
}
```

You can also use the `fixedDelayMilliseconds` method or property to add delay to your stubs. The following example shows how to do so:

groovy

```
org.springframework.cloud.contract.spec.Contract.make {
    request {
        method GET()
        url '/get'
    }
    response {
        status 200
        body 'Passed'
        fixedDelayMilliseconds 1000
    }
}
```

yml

```
response:
  fixedDelayMilliseconds: 1000
```

java

```
class contract implements Supplier<Collection<Contract>> {

    @Override
    public Collection<Contract> get() {
        return Collections.singletonList(Contract.make(c -> {
            c.request(r -> {
                // ...
            });
            c.response(r -> {
                r.fixedDelayMilliseconds(1000);
                // ...
            });
        }));
    }
}
```

kotlin

```
import org.springframework.cloud.contract.spec.ContractDsl.Companion.contract

contract {
    request {
        // ...
    }
    response {
        delay = fixedMilliseconds(1000)
        // ...
    }
}
```

XML Support for HTTP

For HTTP contracts, we also support using XML in the request and response body. The XML body has to be passed within the `body` element as a `String` or `GString`. Also, body matchers can be provided for both the request and the response. In place of the `jsonPath(...)` method, the `org.springframework.cloud.contract.spec.internal.BodyMatchers.xpath` method should be used, with the desired `xPath` provided as the first argument and the appropriate `MatchingType` as second. All the body matchers apart from `byType()` are supported.

The following example shows a Groovy DSL contract with XML in the response body:



```

Contract.make {
    request {
        method GET()
        urlPath '/get'
        headers {
            contentType(applicationXml())
        }
    }
    response {
        status(OK())
        headers {
            contentType(applicationXml())
        }
        body ""
    }
}

<test>
<duck type='xtype'>123</duck>
<alpha>abc</alpha>
<list>
<elem>abc</elem>
<elem>def</elem>
<elem>ghi</elem>
</list>
<number>123</number>
<aBoolean>true</aBoolean>
<date>2017-01-01</date>
<dateTime>2017-01-01T01:23:45</dateTime>
<time>01:02:34</time>
<valueWithoutAMatcher>foo</valueWithoutAMatcher>
<key><complex>foo</complex></key>
</test>""

    bodyMatchers {
        XPath('/test/duck/text()', byRegexp("[0-9]{3}"))
        XPath('/test/duck/text()',
byCommand('equals($it)'))
        XPath('/test/duck/xxx', byNull())
        XPath('/test/duck/text()', byEquality())
        XPath('/test/alpha/text()',
byRegexp(onlyAlphaUnicode()))
        XPath('/test/alpha/text()', byEquality())
        XPath('/test/number/text()', byRegexp(number()))
        XPath('/test/date/text()', byDate())
        XPath('/test/dateTime/text()', byTimestamp())
        XPath('/test/time/text()', byTime())
        XPath('/test/*/complex/text()', byEquality())
        XPath('/test/duck/@type', byEquality())
    }
}
}

```

yaml

```
include::/home/marcin/repo/spring-cloud-scripts/contract/spring-cloud-contract-verifier/src/test/resources/yml/contract_rest_xml.yml
```

java

```
import java.util.function.Supplier;

import org.springframework.cloud.contract.spec.Contract;

class contract_xml implements Supplier<Contract> {

    @Override
    public Contract get() {
        return Contract.make(c -> {
            c.request(r -> {
                r.method(r.GET());
                r.urlPath("/get");
                r.headers(h -> {
                    h.contentType(h.applicationXml());
                });
            });
            c.response(r -> {
                r.status(r.OK());
                r.headers(h -> {
                    h.contentType(h.applicationXml());
                });
                r.body("<test>\n" + "<duck type='xtype'>123</duck>\n"
                    + "<alpha>abc</alpha>\n" + "<list>\n" +
                    "<elem>abc</elem>\n"
                    + "<elem>def</elem>\n" + "<elem>ghi</elem>\n" +
                    "</list>\n"
                    + "<number>123</number>\n" + "<aBoolean>>true</aBoolean>\n"
                    + "<date>2017-01-01</date>\n"
                    + "<dateTime>2017-01-01T01:23:45</dateTime>\n"
                    + "<time>01:02:34</time>\n"
                    + "<valueWithoutAMatcher>foo</valueWithoutAMatcher>\n"
                    + "<key><complex>foo</complex></key>\n" + "</test>");
                r.bodyMatchers(m -> {
                    m.xpath("/test/duck/text()", m.byRegex("[0-9]{3}"));
                    m.xpath("/test/duck/text()", m.byCommand("equals($it)"));
                    m.xpath("/test/duck/xxx", m.byNull());
                    m.xpath("/test/duck/text()", m.byEquality());
                    m.xpath("/test/alpha/text()",
                    m.byRegex(r.onlyAlphaUnicode()));
                    m.xpath("/test/alpha/text()", m.byEquality());
                    m.xpath("/test/number/text()", m.byRegex(r.number()));
                    m.xpath("/test/date/text()", m.byDate());
                    m.xpath("/test/dateTime/text()", m.byTimestamp());
                });
            });
        });
    }
}
```

```
        m.xpath("/test/time/text()", m.byTime());
        m.xpath("/test/*/complex/text()", m.byEquality());
        m.xpath("/test/duck/@type", m.byEquality());
    });
});
};
}
```


kotlin

```
import org.springframework.cloud.contract.spec.ContractDsl.Companion.contract

contract {
    request {
        method = GET
        urlPath = path("/get")
        headers {
            contentType = APPLICATION_XML
        }
    }
    response {
        status = OK
        headers {
            contentType =APPLICATION_XML
        }
        body = body("<test>\n" + "<duck type='xtype'>123</duck>\n"
            + "<alpha>abc</alpha>\n" + "<list>\n" + "<elem>abc</elem>\n"
            + "<elem>def</elem>\n" + "<elem>ghi</elem>\n" + "</list>\n"
            + "<number>123</number>\n" + "<aBoolean>true</aBoolean>\n"
            + "<date>2017-01-01</date>\n"
            + "<dateTime>2017-01-01T01:23:45</dateTime>\n"
            + "<time>01:02:34</time>\n"
            + "<valueWithoutAMatcher>foo</valueWithoutAMatcher>\n"
            + "<key><complex>foo</complex></key>\n" + "</test>")
        bodyMatchers {
            xPath("/test/duck/text()", byRegex("[0-9]{3}"))
            xPath("/test/duck/text()", byCommand("equals(\$it)"))
            xPath("/test/duck/xxx", byNull)
            xPath("/test/duck/text()", byEquality)
            xPath("/test/alpha/text()", byRegex(onlyAlphaUnicode))
            xPath("/test/alpha/text()", byEquality)
            xPath("/test/number/text()", byRegex(number))
            xPath("/test/date/text()", byDate)
            xPath("/test/dateTime/text()", byTimestamp)
            xPath("/test/time/text()", byTime)
            xPath("/test/*/complex/text()", byEquality)
            xPath("/test/duck/@type", byEquality)
        }
    }
}
```

The following example shows an automatically generated test for XML in the response body:

```

@Test
public void validate_xmlMatches() throws Exception {
    // given:
    MockMvcRequestSpecification request = given()
        .header("Content-Type", "application/xml");

    // when:
    ResponseOptions response = given().spec(request).get("/get");

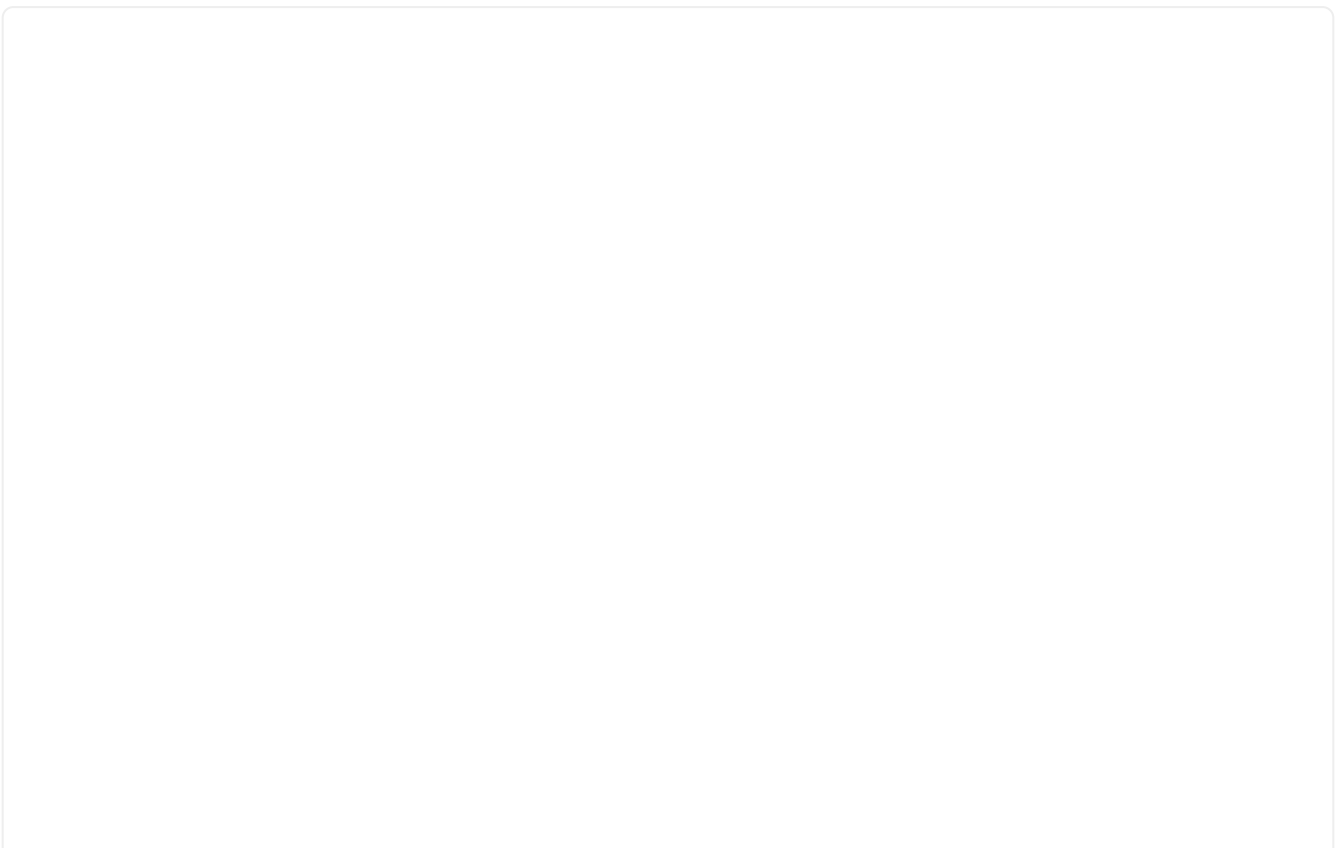
    // then:
    assertThat(response.statusCode()).isEqualTo(200);
    // and:
    DocumentBuilder documentBuilder = DocumentBuilderFactory.newInstance()
        .newDocumentBuilder();
    Document parsedXml = documentBuilder.parse(new InputSource(
        new StringReader(response.getBody().asString())));
    // and:
    assertThat(valueFromXPath(parsedXml, "/test/list/elem/text()").isEqualTo("abc");

    assertThat(valueFromXPath(parsedXml, "/test/list/elem[2]/text()").isEqualTo("def");
    assertThat(valueFromXPath(parsedXml, "/test/duck/text()").matches("[0-9]{3}");
    assertThat(nodeFromXPath(parsedXml, "/test/duck/xxx").isNull();
    assertThat(valueFromXPath(parsedXml, "/test/alpha/text()").matches("[\\p{L}]*");
    assertThat(valueFromXPath(parsedXml, "/test/*/complex/text()").isEqualTo("foo");
    assertThat(valueFromXPath(parsedXml, "/test/duck/@type").isEqualTo("xtype");
}

```

Multiple Contracts in One File

You can define multiple contracts in one file. Such a contract might resemble the following example:



groovy

```
import org.springframework.cloud.contract.spec.Contract

[
  Contract.make {
    name("should post a user")
    request {
      method 'POST'
      url('/users/1')
    }
    response {
      status OK()
    }
  },
  Contract.make {
    request {
      method 'POST'
      url('/users/2')
    }
    response {
      status OK()
    }
  }
]
```

yaml

```
---
name: should post a user
request:
  method: POST
  url: /users/1
response:
  status: 200
---
request:
  method: POST
  url: /users/2
response:
  status: 200
---
request:
  method: POST
  url: /users/3
response:
  status: 200
```

java

```
class contract implements Supplier<Collection<Contract>> {

    @Override
    public Collection<Contract> get() {
        return Arrays.asList(
            Contract.make(c -> {
                c.name("should post a user");
                // ...
            }), Contract.make(c -> {
                // ...
            }), Contract.make(c -> {
                // ...
            })
        );
    }
}
```

kotlin

```
import org.springframework.cloud.contract.spec.ContractDsl.Companion.contract

arrayOf(
    contract {
        name("should post a user")
        // ...
    },
    contract {
        // ...
    },
    contract {
        // ...
    }
}
```

In the preceding example, one contract has the `name` field and the other does not. This leads to generation of two tests that look more or less like the following:

```

package org.springframework.cloud.contract.verifier.tests.com.hello;

import com.example.TestBase;
import com.jayway.jsonpath.DocumentContext;
import com.jayway.jsonpath.JsonPath;
import
com.jayway.restassured.module.mockmvc.specification.MockMvcRequestSpecification;
import com.jayway.restassured.response.ResponseOptions;
import org.junit.Test;

import static com.jayway.restassured.module.mockmvc.RestAssuredMockMvc.*;
import static com.toomuchcoding.jsonassert.JsonAssertion.assertThatJson;
import static org.assertj.core.api.Assertions.assertThat;

public class V1Test extends TestBase {

    @Test
    public void validate_should_post_a_user() throws Exception {
        // given:
        MockMvcRequestSpecification request = given();

        // when:
        ResponseOptions response = given().spec(request)
            .post("/users/1");

        // then:
        assertThat(response.statusCode()).isEqualTo(200);
    }

    @Test
    public void validate_withList_1() throws Exception {
        // given:
        MockMvcRequestSpecification request = given();

        // when:
        ResponseOptions response = given().spec(request)
            .post("/users/2");

        // then:
        assertThat(response.statusCode()).isEqualTo(200);
    }
}

```

Notice that, for the contract that has the `name` field, the generated test method is named `validate_should_post_a_user`. The one that does not have the `name` field is called `validate_withList_1`. It corresponds to the name of the file `WithList.groovy` and the index of the contract in the list.

The generated stubs are shown in the following example:

```
should post a user.json
1_WithList.json
```

The first file got the `name` parameter from the contract. The second got the name of the contract file (`WithList.groovy`) prefixed with the index (in this case, the contract had an index of `1` in the list of contracts in the file).



It is much better to name your contracts, because doing so makes your tests far more meaningful.

Stateful Contracts

Stateful contracts (known also as scenarios) are contract definitions that should be read in order. This might be useful in the following situations:

- You want to execute the contract in a precisely defined order, since you use Spring Cloud Contract to test your stateful application



We really discourage you from doing that, since contract tests should be stateless.

- You want the same endpoint to return different results for the same request.

To create stateful contracts (or scenarios), you need to use the proper naming convention while creating your contracts. The convention requires including an order number followed by an underscore. This works regardless of whether you work with YAML or Groovy. The following listing shows an example:

```
my_contracts_dir\  
  scenario1\  
    1_login.groovy  
    2_showCart.groovy  
    3_logout.groovy
```

Such a tree causes Spring Cloud Contract Verifier to generate WireMock's scenario with a name of `scenario1` and the three following steps:

1. login, marked as `Started` pointing to...
2. showCart, marked as `Step1` pointing to...
3. logout, marked as `Step2` (which closes the scenario).

You can find more details about WireMock scenarios at <https://wiremock.org/docs/stateful-behaviour/>.


```

import io.restassured.module.webtestclient.RestAssuredWebTestClient;
import org.junit.Before;

public abstract class BeerRestBase {

    @Before
    public void setup() {
        RestAssuredWebTestClient.standaloneSetup(
            new ProducerController(personToCheck -> personToCheck.age >= 20));
    }
}
}

```



The `WebTestClient` mode is faster than the `EXPLICIT` mode.

WebFlux with Explicit Mode

You can also use WebFlux with the explicit mode in your generated tests to work with WebFlux. The following example shows how to configure using explicit mode:

Maven

```

<plugin>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-contract-maven-plugin</artifactId>
  <version>${spring-cloud-contract.version}</version>
  <extensions>>true</extensions>
  <configuration>
    <testMode>EXPLICIT</testMode>
  </configuration>
</plugin>

```

Gradle

```

contracts {
    testMode = 'EXPLICIT'
}

```

The following example shows how to set up a base class and RestAssured for Web Flux:

```

@RunWith(SpringRunner.class)
@SpringBootTest(classes = BeerRestBase.Config.class,
    webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT,
    properties = "server.port=0")
public abstract class BeerRestBase {

    // your tests go here

    // in this config class you define all controllers and mocked services
@Configuration
@EnableAutoConfiguration
static class Config {

    @Bean
    PersonCheckingService personCheckingService() {
        return personToCheck -> personToCheck.age >= 20;
    }

    @Bean
    ProducerController producerController() {
        return new ProducerController(personCheckingService());
    }
}
}

```

Working with Context Paths

Spring Cloud Contract supports context paths.

The only change needed to fully support context paths is the switch on the producer side. Also, the autogenerated tests must use explicit mode. The consumer side remains untouched. In order for the generated test to pass, you must use explicit mode. The following example shows how to set the test mode to **EXPLICIT**:



Maven

```
<plugin>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-contract-maven-plugin</artifactId>
  <version>${spring-cloud-contract.version}</version>
  <extensions>>true</extensions>
  <configuration>
    <testMode>EXPLICIT</testMode>
  </configuration>
</plugin>
```

Gradle

```
contracts {
    testMode = 'EXPLICIT'
}
```

That way, you generate a test that does not use MockMvc. It means that you generate real requests and you need to set up your generated test's base class to work on a real socket.

Consider the following contract:

```
org.springframework.cloud.contract.spec.Contract.make {
    request {
        method 'GET'
        url '/my-context-path/url'
    }
    response {
        status OK()
    }
}
```

The following example shows how to set up a base class and RestAssured:

```

import io.restassured.RestAssured;
import org.junit.Before;
import org.springframework.boot.web.server.LocalServerPort;
import org.springframework.boot.test.context.SpringBootTest;

@SpringBootTest(classes = ContextPathTestingBaseClass.class, webEnvironment =
SpringBootTest.WebEnvironment.RANDOM_PORT)
class ContextPathTestingBaseClass {

    @LocalServerPort int port;

    @Before
    public void setup() {
        RestAssured.baseURI = "http://localhost";
        RestAssured.port = this.port;
    }
}

```

If you do it this way:

- All of your requests in the autogenerated tests are sent to the real endpoint with your context path included (for example, `/my-context-path/url`).
- Your contracts reflect that you have a context path. Your generated stubs also have that information (for example, in the stubs, you have to call `/my-context-path/url`).

Working with REST Docs

You can use [Spring REST Docs](#) to generate documentation (for example, in AsciiDoc format) for an HTTP API with Spring MockMvc, `WebTestClient`, or RestAssured. At the same time that you generate documentation for your API, you can also generate WireMock stubs by using Spring Cloud Contract WireMock. To do so, write your normal REST Docs test cases and use `@AutoConfigureRestDocs` to have stubs be automatically generated in the REST Docs output directory.

[rest docs] | *rest-docs.png*

The following example uses `MockMvc`:

```

@RunWith(SpringRunner.class)
@SpringBootTest
@AutoConfigureRestDocs(outputDir = "target/snippets")
@AutoConfigureMockMvc
public class ApplicationTests {

    @Autowired
    private MockMvc mockMvc;

    @Test
    public void contextLoads() throws Exception {
        mockMvc.perform(get("/resource"))
            .andExpect(content().string("Hello World"))
            .andDo(document("resource"));
    }
}

```

This test generates a WireMock stub at `target/snippets/stubs/resource.json`. It matches all `GET` requests to the `/resource` path. The same example with `WebTestClient` (used for testing Spring WebFlux applications) would be as follows:

```

@RunWith(SpringRunner.class)
@SpringBootTest
@AutoConfigureRestDocs(outputDir = "target/snippets")
@AutoConfigureWebTestClient
public class ApplicationTests {

    @Autowired
    private WebTestClient client;

    @Test
    public void contextLoads() throws Exception {
        client.get().uri("/resource").exchange()
            .expectBody(String.class).isEqualTo("Hello World")
            .consumeWith(document("resource"));
    }
}

```

Without any additional configuration, these tests create a stub with a request matcher for the HTTP method and all headers except `host` and `content-length`. To match the request more precisely (for example, to match the body of a POST or PUT), we need to explicitly create a request matcher. Doing so has two effects:

- Creating a stub that matches only in the way you specify.
- Asserting that the request in the test case also matches the same conditions.

The main entry point for this feature is `WireMockRestDocs.verify()`, which can be used as a substitute for the `document()` convenience method, as the following example shows:

```

import static
org.springframework.cloud.contract.wiremock.restdocs.WireMockRestDocs.verify;

@RunWith(SpringRunner.class)
@SpringBootTest
@AutoConfigureRestDocs(outputDir = "target/snippets")
@AutoConfigureMockMvc
public class ApplicationTests {

    @Autowired
    private MockMvc mockMvc;

    @Test
    public void contextLoads() throws Exception {
        mockMvc.perform(post("/resource")
            .content("{\"id\":\"123456\",\"message\":\"Hello World\"}")
            .andExpect(status().isOk())
            .andDo(verify().jsonPath("$.id"))
            .andDo(document("resource")));
    }
}

```

The preceding contract specifies that any valid POST with an `id` field receives the response defined in this test. You can chain together calls to `.jsonPath()` to add additional matchers. If JSON Path is unfamiliar, the [JayWay documentation](#) can help you get up to speed. The `WebTestClient` version of this test has a similar `verify()` static helper that you insert in the same place.

Instead of the `jsonPath` and `contentType` convenience methods, you can also use the WireMock APIs to verify that the request matches the created stub, as the following example shows:

```

@Test
public void contextLoads() throws Exception {
    mockMvc.perform(post("/resource")
        .content("{\"id\":\"123456\",\"message\":\"Hello World\"}")
        .andExpect(status().isOk())
        .andDo(verify()
            .wiremock(WireMock.post(
                urlPathEquals("/resource"))
                .withRequestBody(matchingJsonPath("$.id"))
                .andDo(document("post-resource"))));
    }
}

```

The WireMock API is rich. You can match headers, query parameters, and the request body by regex as well as by JSON path. You can use these features to create stubs with a wider range of parameters. The preceding example generates a stub resembling the following example:

post-resource.json

```
{
  "request" : {
    "url" : "/resource",
    "method" : "POST",
    "bodyPatterns" : [ {
      "matchesJsonPath" : "$.id"
    }
  ]
},
"response" : {
  "status" : 200,
  "body" : "Hello World",
  "headers" : {
    "X-Application-Context" : "application:-1",
    "Content-Type" : "text/plain"
  }
}
}
```



You can use either the `wiremock()` method or the `jsonPath()` and `contentType()` methods to create request matchers, but you cannot use both approaches.

On the consumer side, you can make the `resource.json` generated earlier in this section available on the classpath (by [Publishing Stubs as JARs](#), for example). After that, you can create a stub that uses WireMock in a number of different ways, including by using `@AutoConfigureWireMock(stubs="classpath:resource.json")`, as described earlier in this document.

Generating Contracts with REST Docs

You can also generate Spring Cloud Contract DSL files and documentation with Spring REST Docs. If you do so in combination with Spring Cloud WireMock, you get both the contracts and the stubs.

Why would you want to use this feature? Some people in the community asked questions about a situation in which they would like to move to DSL-based contract definition, but they already have a lot of Spring MVC tests. Using this feature lets you generate the contract files that you can later modify and move to folders (defined in your configuration) so that the plugin finds them.



You might wonder why this functionality is in the WireMock module. The functionality is there because it makes sense to generate both the contracts and the stubs.

Consider the following test:

```
this.mockMvc
    .perform(post("/foo").accept(MediaType.APPLICATION_PDF)
        .accept(MediaType.APPLICATION_JSON)
        .contentType(MediaType.APPLICATION_JSON)
        .content("{\"foo\": 23, \"bar\" : \"baz\" }"))
    .andExpect(status().isOk()).andExpect(content().string("bar"))
    // first WireMock
    .andDo(WireMockRestDocs.verify().jsonPath("$.foo >= 20)")
        .jsonPath("$.bar in ['baz', 'bazz', 'bazzz']")
        .contentType(MediaType.valueOf("application/json")))
    // then Contract DSL documentation
    .andDo(document("index",
SpringCloudContractRestDocs.dslContract()));
```

The preceding test creates the stub presented in the previous section, generating both the contract and a documentation file.

The contract is called `index.groovy` and might resemble the following example:


```

import org.springframework.cloud.contract.spec.Contract

Contract.make {
    request {
        method 'POST'
        url '/foo'
        body(''
            {"foo": 23 }
        '')
        headers {
            header(''Accept'', ''application/json'')
            header(''Content-Type'', ''application/json'')
        }
    }
    response {
        status OK()
        body(''
            bar
        '')
        headers {
            header(''Content-Type'', ''application/json;charset=UTF-8'')
            header(''Content-Length'', ''3'')
        }
        bodyMatchers {
            jsonPath('$[?(@.foo >= 20)]', byType())
        }
    }
}

```

The generated document (formatted in AsciiDoc in this case) contains a formatted contract. The location of this file would be [index/dsl-contract.adoc](#).

14.3.4. Messaging

Spring Cloud Contract lets you verify applications that use messaging as a means of communication. All of the integrations shown in this document work with Spring, but you can also create one of your own and use that.

Messaging DSL Top-Level Elements

The DSL for messaging looks a little bit different than the one that focuses on HTTP. The following sections explain the differences:

- [Output Triggered by a Method](#)
- [Output Triggered by a Message](#)
- [Consumer/Producer](#)

- [Common](#)

Output Triggered by a Method

The output message can be triggered by calling a method (such as a [Scheduler](#) when a contract was started and a message was sent), as shown in the following example:

groovy

```
def dsl = Contract.make {
    // Human readable description
    description 'Some description'
    // Label by means of which the output message can be triggered
    label 'some_label'
    // input to the contract
    input {
        // the contract will be triggered by a method
        triggeredBy('bookReturnedTriggered()')
    }
    // output message of the contract
    outputMessage {
        // destination to which the output message will be sent
        sentTo('output')
        // the body of the output message
        body('{ "bookName" : "foo" }')
        // the headers of the output message
        headers {
            header('BOOK-NAME', 'foo')
        }
    }
}
```

yml

```
# Human readable description
description: Some description
# Label by means of which the output message can be triggered
label: some_label
input:
  # the contract will be triggered by a method
  triggeredBy: bookReturnedTriggered()
# output message of the contract
outputMessage:
  # destination to which the output message will be sent
  sentTo: output
  # the body of the output message
  body:
    bookName: foo
  # the headers of the output message
  headers:
    BOOK-NAME: foo
```

In the previous example case, the output message is sent to `output` if a method called `bookReturnedTriggered` is executed. On the message publisher's side, we generate a test that calls that method to trigger the message. On the consumer side, you can use the `some_label` to trigger the

message.

Output Triggered by a Message

The output message can be triggered by receiving a message, as shown in the following example:

groovy

```
def dsl = Contract.make {
  description 'Some Description'
  label 'some_label'
  // input is a message
  input {
    // the message was received from this destination
    messageFrom('input')
    // has the following body
    messageBody([
      bookName: 'foo'
    ])
    // and the following headers
    messageHeaders {
      header('sample', 'header')
    }
  }
  outputMessage {
    sentTo('output')
    body([
      bookName: 'foo'
    ])
    headers {
      header('BOOK-NAME', 'foo')
    }
  }
}
```

yml

```
# Human readable description
description: Some description
# Label by means of which the output message can be triggered
label: some_label
# input is a message
input:
  messageFrom: input
  # has the following body
  messageBody:
    bookName: 'foo'
  # and the following headers
  messageHeaders:
    sample: 'header'
# output message of the contract
outputMessage:
  # destination to which the output message will be sent
  sentTo: output
  # the body of the output message
  body:
    bookName: foo
  # the headers of the output message
  headers:
    BOOK-NAME: foo
```

In the preceding example, the output message is sent to **output** if a proper message is received on the **input** destination. On the message publisher's side, the engine generates a test that sends the input message to the defined destination. On the consumer side, you can either send a message to the input destination or use a label (**some_label** in the example) to trigger the message.

Consumer/Producer



This section is valid only for Groovy DSL.

In HTTP, you have a notion of **client/stub** and **`server/test** notation. You can also use those paradigms in messaging. In addition, Spring Cloud Contract Verifier also provides the **consumer** and **producer** methods, as presented in the following example (note that you can use either **\$** or **value** methods to provide **consumer** and **producer** parts):

```

        Contract.make {
            name "foo"
                label 'some_label'
                input {
                    messageFrom value(consumer('jms:output'),
producer('jms:input'))
                    messageBody([
                        bookName: 'foo'
                    ])
                    messageHeaders {
                        header('sample', 'header')
                    }
                }
                outputMessage {
                    sentTo $(consumer('jms:input'), producer('jms:output'))
                    body([
                        bookName: 'foo'
                    ])
                }
        }
    }
}

```

Common

In the `input` or `outputMessage` section, you can call `assertThat` with the name of a `method` (for example, `assertThatMessageIsOnTheQueue()`) that you have defined in the base class or in a static import. Spring Cloud Contract runs that method in the generated test.

Integrations

You can use one of the following four integration configurations:

- Apache Camel
- Spring Integration
- Spring Cloud Stream
- Spring AMQP
- Spring JMS (requires embedded broker)
- Spring Kafka (requires embedded broker)

Since we use Spring Boot, if you have added one of these libraries to the classpath, all the messaging configuration is automatically set up.



Remember to put `@AutoConfigureMessageVerifier` on the base class of your generated tests. Otherwise, the messaging part of Spring Cloud Contract does not work.

If you want to use Spring Cloud Stream, remember to add a test dependency on `org.springframework.cloud:spring-cloud-stream`, as follows:

Maven

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-stream</artifactId>
  <type>test-jar</type>
  <scope>test</scope>
  <classifier>test-binder</classifier>
</dependency>
```

Gradle

```
testImplementation(group: 'org.springframework.cloud', name:
'spring-cloud-stream', classifier: 'test-binder')
```

Manual Integration Testing

The main interface used by the tests is `org.springframework.cloud.contract.verifier.messaging.MessageVerifier`. It defines how to send and receive messages. You can create your own implementation to achieve the same goal.

In a test, you can inject a `ContractVerifierMessageExchange` to send and receive messages that follow the contract. Then add `@AutoConfigureMessageVerifier` to your test. The following example shows how to do so:

```
@RunWith(SpringTestRunner.class)
@SpringBootTest
@AutoConfigureMessageVerifier
public static class MessagingContractTests {

    @Autowired
    private MessageVerifier verifier;
    ...
}
```



If your tests require stubs as well, then `@AutoConfigureStubRunner` includes the messaging configuration, so you only need the one annotation.

Producer Side Messaging Test Generation

Having the `input` or `outputMessage` sections in your DSL results in creation of tests on the publisher's

side. By default, JUnit 4 tests are created. However, there is also a possibility to create JUnit 5, TestNG, or Spock tests.

There are three main scenarios that we should take into consideration:

- Scenario 1: There is no input message that produces an output message. The output message is triggered by a component inside the application (for example, a scheduler).
- Scenario 2: The input message triggers an output message.
- Scenario 3: The input message is consumed, and there is no output message.



The destination passed to `messageFrom` or `sentTo` can have different meanings for different messaging implementations. For Stream and Integration, it is first resolved as a `destination` of a channel. Then, if there is no such `destination` it is resolved as a channel name. For Camel, that's a certain component (for example, `jms`).

Scenario 1: No Input Message

Consider the following contract:

groovy

```
def contractDsl = Contract.make {
    name "foo"
    label 'some_label'
    input {
        triggeredBy('bookReturnedTriggered()')
    }
    outputMessage {
        sentTo('activemq:output')
        body(''{ "bookName" : "foo" }''')
        headers {
            header('BOOK-NAME', 'foo')
            messagingContentType(applicationJson())
        }
    }
}
```

yml

```
label: some_label
input:
  triggeredBy: bookReturnedTriggered
outputMessage:
  sentTo: activemq:output
  body:
    bookName: foo
  headers:
    BOOK-NAME: foo
  contentType: application/json
```

For the preceding example, the following test would be created:

JUnit

```
'''\
package com.example;

import com.jayway.jsonpath.DocumentContext;
import com.jayway.jsonpath.JsonPath;
import org.junit.Test;
import org.junit.Rule;
import javax.inject.Inject;
import
org.springframework.cloud.contract.verifier.messaging.internal.ContractVerifierObjectMapper;
import
```

```

org.springframework.cloud.contract.verifier.messaging.internal.ContractVerifierMes
sage;
import
org.springframework.cloud.contract.verifier.messaging.internal.ContractVerifierMes
saging;

import static
org.springframework.cloud.contract.verifier.assertion.SpringCloudContractAssertion
s.assertThat;
import static
org.springframework.cloud.contract.verifier.util.ContractVerifierUtil.*;
import static com.toomuchcoding.jsonassert.JsonAssertion.assertThatJson;
import static
org.springframework.cloud.contract.verifier.messaging.util.ContractVerifierMessagi
ngUtil.headers;
import static
org.springframework.cloud.contract.verifier.util.ContractVerifierUtil.fileToBytes;

@SuppressWarnings("rawtypes")
public class FooTest {
\t@Inject ContractVerifierMessaging contractVerifierMessaging;
\t@Inject ContractVerifierObjectMapper contractVerifierObjectMapper;

\t@Test
\tpublic void validate_foo() throws Exception {
\t\t// when:
\t\t\tbookReturnedTriggered();

\t\t// then:
\t\t\tContractVerifierMessage response =
contractVerifierMessaging.receive("activemq:output");
\t\t\tassertThat(response).isNotNull();

\t\t// and:
\t\t\tassertThat(response.getHeader("BOOK-NAME")).isNotNull();
\t\t\tassertThat(response.getHeader("BOOK-NAME").toString()).isEqualTo("foo");
\t\t\tassertThat(response.getHeader("contentType")).isNotNull();
\t\t\tassertThat(response.getHeader("contentType").toString()).isEqualTo("applicat
ion/json");

\t\t// and:
\t\t\tDocumentContext parsedJson =
JsonPath.parse(contractVerifierObjectMapper.writeValueAsString(response.getPayload
()));
\t\t\tassertThatJson(parsedJson).field("[ 'bookName' ]").isEqualTo("foo");
\t}

}

...

```

```

'''\
package com.example

import com.jayway.jsonpath.DocumentContext
import com.jayway.jsonpath.JsonPath
import spock.lang.Specification
import javax.inject.Inject
import
org.springframework.cloud.contract.verifier.messaging.internal.ContractVerifierObjectMapper
import
org.springframework.cloud.contract.verifier.messaging.internal.ContractVerifierMessage
import
org.springframework.cloud.contract.verifier.messaging.internal.ContractVerifierMessaging

import static
org.springframework.cloud.contract.verifier.assertion.SpringCloudContractAssertions.assertThat
import static
org.springframework.cloud.contract.verifier.util.ContractVerifierUtil.*
import static com.toomuchcoding.jsonassert.JsonAssertion.assertThatJson
import static
org.springframework.cloud.contract.verifier.messaging.util.ContractVerifierMessagingUtil.headers
import static
org.springframework.cloud.contract.verifier.util.ContractVerifierUtil.fileToBytes

@SuppressWarnings("rawtypes")
class FooSpec extends Specification {
    \t@Inject ContractVerifierMessaging contractVerifierMessaging
    \t@Inject ContractVerifierObjectMapper contractVerifierObjectMapper

    \tdef validate_foo() throws Exception {
        \t\twhen:
            \t\t\tbookReturnedTriggered()

            \t\tthen:
                \t\t\tContractVerifierMessage response =
                    contractVerifierMessaging.receive("activemq:output")
                \t\t\tresponse != null

            \t\tand:
                \t\t\tresponse.getHeader("BOOK-NAME") != null
                \t\t\tresponse.getHeader("BOOK-NAME").toString() == 'foo'
                \t\t\tresponse.getHeader("contentType") != null
                \t\t\tresponse.getHeader("contentType").toString() == 'application/json'
    }
}

```

```
\t\tand:  
\t\t\tDocumentContext parsedJson =  
\t\t\tJsonPath.parse(contractVerifierObjectMapper.writeValueAsString(response.getPayload  
\t\t\t\t()))  
\t\t\t\tassertThatJson(parsedJson).field("[ 'bookName' ]").isEqualTo("foo")  
\t\t\t}  
  
\t}  
  
...
```

Scenario 2: Output Triggered by Input

Consider the following contract:

groovy

```
def contractDsl = Contract.make {
  name "foo"
  label 'some_label'
  input {
    messageFrom('jms:input')
    messageBody([
      bookName: 'foo'
    ])
    messageHeaders {
      header('sample', 'header')
    }
  }
  outputMessage {
    sentTo('jms:output')
    body([
      bookName: 'foo'
    ])
    headers {
      header('BOOK-NAME', 'foo')
    }
  }
}
```

yml

```
label: some_label
input:
  messageFrom: jms:input
  messageBody:
    bookName: 'foo'
  messageHeaders:
    sample: header
outputMessage:
  sentTo: jms:output
  body:
    bookName: foo
  headers:
    BOOK-NAME: foo
```

For the preceding contract, the following test would be created:

JUnit

```
package com.example;
```



```

\t\t\tassertThat(response.getHeader("BOOK-NAME")).isNotNull();
\t\t\tassertThat(response.getHeader("BOOK-NAME").toString().isEqualTo("foo");

\t\t// and:
\t\t\tDocumentContext parsedJson =
\t\t\t\tJsonPath.parse(contractVerifierObjectMapper.writeValueAsString(response.getPayload
\t\t\t\t()));
\t\t\tassertThatJson(parsedJson).field("'bookName']").isEqualTo("foo");
\t\t}

}

...

```

Spock

```

        """\
package com.example

import com.jayway.jsonpath.DocumentContext
import com.jayway.jsonpath.JsonPath
import spock.lang.Specification
import javax.inject.Inject
import
org.springframework.cloud.contract.verifier.messaging.internal.ContractVerifierObj
ectMapper
import
org.springframework.cloud.contract.verifier.messaging.internal.ContractVerifierMes
sage
import
org.springframework.cloud.contract.verifier.messaging.internal.ContractVerifierMes
saging

import static
org.springframework.cloud.contract.verifier.assertion.SpringCloudContractAssertion
s.assertThat
import static
org.springframework.cloud.contract.verifier.util.ContractVerifierUtil.*
import static com.toomuchcoding.jsonassert.JsonAssertion.assertThatJson
import static
org.springframework.cloud.contract.verifier.messaging.util.ContractVerifierMessagi
ngUtil.headers
import static
org.springframework.cloud.contract.verifier.util.ContractVerifierUtil.fileToBytes

@SuppressWarnings("rawtypes")
class FooSpec extends Specification {
\t@Inject ContractVerifierMessaging contractVerifierMessaging
\t@Inject ContractVerifierObjectMapper contractVerifierObjectMapper

\tdef validate_foo() throws Exception {

```


groovy

```
def contractDsl = Contract.make {
    name "foo"
    label 'some_label'
    input {
        messageFrom('jms:delete')
        messageBody([
            bookName: 'foo'
        ])
        messageHeaders {
            header('sample', 'header')
        }
        assertThat('bookWasDeleted()')
    }
}
```

yml

```
label: some_label
input:
  messageFrom: jms:delete
  messageBody:
    bookName: 'foo'
  messageHeaders:
    sample: header
  assertThat: bookWasDeleted()
```

For the preceding contract, the following test would be created:

JUnit

```
package com.example;

import com.jayway.jsonpath.DocumentContext;
import com.jayway.jsonpath.JsonPath;
import org.junit.Test;
import org.junit.Rule;
import javax.inject.Inject;
import
org.springframework.cloud.contract.verifier.messaging.internal.ContractVerifierObjectMapper;
import
org.springframework.cloud.contract.verifier.messaging.internal.ContractVerifierMessage;
import
```




If you have multiple frameworks on the classpath, Stub Runner needs to define which one should be used. Assume that you have AMQP, Spring Cloud Stream, and Spring Integration on the classpath and that you want to use Spring AMQP. Then you need to set `stubrunner.stream.enabled=false` and `stubrunner.integration.enabled=false`. That way, the only remaining framework is Spring AMQP.

Stub triggering

To trigger a message, use the `StubTrigger` interface, as the following example shows:

```

package org.springframework.cloud.contract.stubrunner;

import java.util.Collection;
import java.util.Map;

/**
 * Contract for triggering stub messages.
 *
 * @author Marcin Grzejszczak
 */
public interface StubTrigger {

    /**
     * Triggers an event by a given label for a given {@code groupid:artifactid}
     notation.
     * You can use only {@code artifactId} too.
     *
     * Feature related to messaging.
     * @param ivyNotation ivy notation of a stub
     * @param labelName name of the label to trigger
     * @return true - if managed to run a trigger
     */
    boolean trigger(String ivyNotation, String labelName);

    /**
     * Triggers an event by a given label.
     *
     * Feature related to messaging.
     * @param labelName name of the label to trigger
     * @return true - if managed to run a trigger
     */
    boolean trigger(String labelName);

    /**
     * Triggers all possible events.
     *
     * Feature related to messaging.
     * @return true - if managed to run a trigger
     */
    boolean trigger();

    /**
     * Feature related to messaging.
     * @return a mapping of ivy notation of a dependency to all the labels it has.
     */
    Map<String, Collection<String>> labels();
}

```

For convenience, the `StubFinder` interface extends `StubTrigger`, so you only need one or the other in

your tests.

`StubTrigger` gives you the following options to trigger a message:

- [Trigger by Label](#)
- [Trigger by Group and Artifact Ids](#)
- [Trigger by Artifact IDs](#)
- [Trigger All Messages](#)

Trigger by Label

The following example shows how to trigger a message with a label:

```
stubFinder.trigger('return_book_1')
```

Trigger by Group and Artifact Ids

```
stubFinder.trigger('org.springframework.cloud.contract.verifier.stubs:streamService',  
'return_book_1')
```

Trigger by Artifact IDs

The following example shows how to trigger a message from artifact IDs:

```
stubFinder.trigger('streamService', 'return_book_1')
```

Trigger All Messages

The following example shows how to trigger all messages:

```
stubFinder.trigger()
```

Consumer Side Messaging With Apache Camel

Spring Cloud Contract Stub Runner's messaging module gives you an easy way to integrate with Apache Camel. For the provided artifacts, it automatically downloads the stubs and registers the required routes.

Adding Apache Camel to the Project

You can have both Apache Camel and Spring Cloud Contract Stub Runner on the classpath. Remember to annotate your test class with `@AutoConfigureStubRunner`.

Disabling the Functionality

If you need to disable this functionality, set the `stubrunner.camel.enabled=false` property.

Examples

Assume that we have the following Maven repository with deployed stubs for the `camelService` application.

```
├── .m2
│   ├── repository
│   │   ├── io
│   │   │   ├── codearte
│   │   │   │   ├── accurest
│   │   │   │   │   ├── stubs
│   │   │   │   │   │   ├── camelService
│   │   │   │   │   │   │   ├── 0.0.1-SNAPSHOT
│   │   │   │   │   │   │   │   ├── camelService-0.0.1-SNAPSHOT.pom
│   │   │   │   │   │   │   │   ├── camelService-0.0.1-SNAPSHOT-stubs.jar
│   │   │   │   │   │   │   │   └── maven-metadata-local.xml
│   │   │   │   │   └── maven-metadata-local.xml
```

Further assume that the stubs contain the following structure:

```
├── META-INF
│   └── MANIFEST.MF
├── repository
│   ├── accurest
│   │   ├── bookDeleted.groovy
│   │   ├── bookReturned1.groovy
│   │   └── bookReturned2.groovy
└── mappings
```

Now consider the following contracts (we number them 1 and 2):

```

Contract.make {
  label 'return_book_1'
  input {
    triggeredBy('bookReturnedTriggered()')
  }
  outputMessage {
    sentTo('jms:output')
    body(''{ "bookName" : "foo" }''')
    headers {
      header('BOOK-NAME', 'foo')
    }
  }
}

```

```

Contract.make {
  label 'return_book_2'
  input {
    messageFrom('jms:input')
    messageBody([
      bookName: 'foo'
    ])
    messageHeaders {
      header('sample', 'header')
    }
  }
  outputMessage {
    sentTo('jms:output')
    body([
      bookName: 'foo'
    ])
    headers {
      header('BOOK-NAME', 'foo')
    }
  }
}

```

Scenario 1 (No Input Message)

To trigger a message from the `return_book_1` label, we use the `StubTrigger` interface, as follows:

```
stubFinder.trigger('return_book_1')
```

Next, we want to listen to the output of the message sent to `jms:output`:


```
Exchange receivedMessage = consumerTemplate.receive('jms:output', 5000)
```

The received message would then pass the following assertions:

```
receivedMessage != null  
assertThatBodyContainsBookNameFoo(receivedMessage.in.body)  
receivedMessage.in.headers.get('BOOK-NAME') == 'foo'
```

Scenario 2 (Output Triggered by Input)

Since the route is set for you, you can send a message to the `jms:output` destination.

```
producerTemplate.  
    sendBodyAndHeaders('jms:input', new BookReturned('foo'), [sample:  
    'header'])
```

Next, we want to listen to the output of the message sent to `jms:output`, as follows:

```
Exchange receivedMessage = consumerTemplate.receive('jms:output', 5000)
```

The received message would pass the following assertions:

```
receivedMessage != null  
assertThatBodyContainsBookNameFoo(receivedMessage.in.body)  
receivedMessage.in.headers.get('BOOK-NAME') == 'foo'
```

Scenario 3 (Input with No Output)

Since the route is set for you, you can send a message to the `jms:output` destination, as follows:

```
producerTemplate.  
    sendBodyAndHeaders('jms:delete', new BookReturned('foo'), [sample:  
    'header'])
```



```
Contract.make {
  label 'return_book_1'
  input {
    triggeredBy('bookReturnedTriggered()')
  }
  outputMessage {
    sentTo('output')
    body(''{ "bookName" : "foo" }''')
    headers {
      header('BOOK-NAME', 'foo')
    }
  }
}
```

```
Contract.make {
  label 'return_book_2'
  input {
    messageFrom('input')
    messageBody([
      bookName: 'foo'
    ])
    messageHeaders {
      header('sample', 'header')
    }
  }
  outputMessage {
    sentTo('output')
    body([
      bookName: 'foo'
    ])
    headers {
      header('BOOK-NAME', 'foo')
    }
  }
}
```

Now consider the following Spring Integration Route:

```

<?xml version="1.0" encoding="UTF-8"?>
<beans:beans xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:beans="http://www.springframework.org/schema/beans"
  xmlns="http://www.springframework.org/schema/integration"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
    https://www.springframework.org/schema/beans/spring-beans.xsd
    http://www.springframework.org/schema/integration
    http://www.springframework.org/schema/integration/spring-
integration.xsd">

  <!-- REQUIRED FOR TESTING -->
  <bridge input-channel="output"
    output-channel="outputTest"/>

  <channel id="outputTest">
    <queue/>
  </channel>

</beans:beans>

```

These examples lend themselves to three scenarios:

1. [Scenario 1 \(No Input Message\)](#)
2. [Scenario 2 \(Output Triggered by Input\)](#)
3. [Scenario 3 \(Input with No Output\)](#)

Scenario 1 (No Input Message)

To trigger a message from the `return_book_1` label, use the `StubTrigger` interface, as follows:

```
stubFinder.trigger('return_book_1')
```

The following listing shows how to listen to the output of the message sent to `jms:output`:

```
Message<?> receivedMessage = messaging.receive('outputTest')
```

The received message would pass the following assertions:

```
receivedMessage != null
assertJsons(receivedMessage.payload)
receivedMessage.headers.get('BOOK-NAME') == 'foo'
```

Scenario 2 (Output Triggered by Input)

Since the route is set for you, you can send a message to the `jms:output` destination, as follows:

```
messaging.send(new BookReturned('foo'), [sample: 'header'], 'input')
```

The following listing shows how to listen to the output of the message sent to `jms:output`:

```
Message<?> receivedMessage = messaging.receive('outputTest')
```

The received message passes the following assertions:

```
receivedMessage != null
assertJsons(receivedMessage.payload)
receivedMessage.headers.get('BOOK-NAME') == 'foo'
```

Scenario 3 (Input with No Output)

Since the route is set for you, you can send a message to the `jms:input` destination, as follows:

```
messaging.send(new BookReturned('foo'), [sample: 'header'], 'delete')
```

Consumer Side Messaging With Spring Cloud Stream

Spring Cloud Contract Stub Runner's messaging module gives you an easy way to integrate with Spring Stream. For the provided artifacts, it automatically downloads the stubs and registers the required routes.



If Stub Runner's integration with the Stream `messageFrom` or `sentTo` strings are resolved first as the `destination` of a channel and no such `destination` exists, the destination is resolved as a channel name.

If you want to use Spring Cloud Stream, remember to add a dependency on `org.springframework.cloud:spring-cloud-stream` test support, as follows:



Maven

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-stream</artifactId>
  <type>test-jar</type>
  <scope>test</scope>
  <classifier>test-binder</classifier>
</dependency>
```

Gradle

```
testImplementation(group: 'org.springframework.cloud', name:
'spring-cloud-stream', classifier: 'test-binder')
```

Adding the Runner to the Project

You can have both Spring Cloud Stream and Spring Cloud Contract Stub Runner on the classpath. Remember to annotate your test class with `@AutoConfigureStubRunner`.

Disabling the Functionality

If you need to disable this functionality, set the `stubrunner.stream.enabled=false` property.

Examples

Assume that you have the following Maven repository with deployed stubs for the `streamService` application:

```
├── .m2
│   ├── repository
│   │   ├── io
│   │   │   ├── codearte
│   │   │   │   ├── accurest
│   │   │   │   │   ├── stubs
│   │   │   │   │   │   ├── streamService
│   │   │   │   │   │   │   ├── 0.0.1-SNAPSHOT
│   │   │   │   │   │   │   │   ├── streamService-0.0.1-SNAPSHOT.pom
│   │   │   │   │   │   │   │   ├── streamService-0.0.1-SNAPSHOT-stubs.jar
│   │   │   │   │   │   │   │   ├── maven-metadata-local.xml
│   │   │   │   │   │   │   └── maven-metadata-local.xml
```

Further assume the stubs contain the following structure:

```
|— META-INF
|   |— MANIFEST.MF
|   |
|   |— repository
|       |— accurest
|           |— bookDeleted.groovy
|           |— bookReturned1.groovy
|           |— bookReturned2.groovy
|           |
|           |— mappings
```

Consider the following contracts (numbered 1 and 2):

```
Contract.make {
  label 'return_book_1'
  input { triggeredBy('bookReturnedTriggered()') }
  outputMessage {
    sentTo('returnBook')
    body(''{ "bookName" : "foo" }''')
    headers { header('BOOK-NAME', 'foo') }
  }
}
```

```
Contract.make {
  label 'return_book_2'
  input {
    messageFrom('bookStorage')
    messageBody([
      bookName: 'foo'
    ])
    messageHeaders { header('sample', 'header') }
  }
  outputMessage {
    sentTo('returnBook')
    body([
      bookName: 'foo'
    ])
    headers { header('BOOK-NAME', 'foo') }
  }
}
```

Now consider the following Spring configuration:

```
stubrunner.repositoryRoot: classpath:m2repo/repository/
stubrunner.ids:
org.springframework.cloud.contract.verifier.stubs:streamService:0.0.1-
SNAPSHOT:stubs
stubrunner.stubs-mode: remote
spring:
  cloud:
    stream:
      bindings:
        output:
          destination: returnBook
        input:
          destination: bookStorage

server:
  port: 0

debug: true
```

These examples lend themselves to three scenarios:

- [Scenario 1 \(No Input Message\)](#)
- [Scenario 2 \(Output Triggered by Input\)](#)
- [Scenario 3 \(Input with No Output\)](#)

Scenario 1 (No Input Message)

To trigger a message from the `return_book_1` label, use the `StubTrigger` interface as follows:

```
stubFinder.trigger('return_book_1')
```

The following example shows how to listen to the output of the message sent to a channel whose `destination` is `returnBook`:

```
Message<?> receivedMessage = messaging.receive('returnBook')
```

The received message passes the following assertions:


```
receivedMessage != null
assertJsons(receivedMessage.payload)
receivedMessage.headers.get('BOOK-NAME') == 'foo'
```

Scenario 2 (Output Triggered by Input)

Since the route is set for you, you can send a message to the `bookStorage` destination, as follows:

```
messaging.send(new BookReturned('foo'), [sample: 'header'], 'bookStorage')
```

The following example shows how to listen to the output of the message sent to `returnBook`:

```
Message<?> receivedMessage = messaging.receive('returnBook')
```

The received message passes the following assertions:

```
receivedMessage != null
assertJsons(receivedMessage.payload)
receivedMessage.headers.get('BOOK-NAME') == 'foo'
```

Scenario 3 (Input with No Output)

Since the route is set for you, you can send a message to the `json:output` destination, as follows:

```
messaging.send(new BookReturned('foo'), [sample: 'header'], 'delete')
```

Consumer Side Messaging With Spring AMQP

Spring Cloud Contract Stub Runner's messaging module provides an easy way to integrate with Spring AMQP's Rabbit Template. For the provided artifacts, it automatically downloads the stubs and registers the required routes.

The integration tries to work standalone (that is, without interaction with a running RabbitMQ message broker). It expects a `RabbitTemplate` on the application context and uses it as a spring boot test named `@SpyBean`. As a result, it can use the Mockito spy functionality to verify and inspect messages sent by the application.

On the message consumer side, the stub runner considers all `@RabbitListener` annotated endpoints and all `SimpleMessageListenerContainer` objects on the application context.

As messages are usually sent to exchanges in AMQP, the message contract contains the exchange name as the destination. Message listeners on the other side are bound to queues. Bindings connect an exchange to a queue. If message contracts are triggered, the Spring AMQP stub runner integration looks for bindings on the application context that matches this exchange. Then it collects the queues from the Spring exchanges and tries to find message listeners bound to these queues. The message is triggered for all matching message listeners.

If you need to work with routing keys, you can pass them by using the `amqp_receivedRoutingKey` messaging header.

Adding the Runner to the Project

You can have both Spring AMQP and Spring Cloud Contract Stub Runner on the classpath and set the property `stubrunner.amqp.enabled=true`. Remember to annotate your test class with `@AutoConfigureStubRunner`.



If you already have Stream and Integration on the classpath, you need to disable them explicitly by setting the `stubrunner.stream.enabled=false` and `stubrunner.integration.enabled=false` properties.

Examples

Assume that you have the following Maven repository with a deployed stubs for the `spring-cloud-contract-amqp-test` application:

```
.m2
├── repository
│   ├── com
│   │   └── example
│   │       ├── spring-cloud-contract-amqp-test
│   │       │   ├── 0.4.0-SNAPSHOT
│   │       │   │   ├── spring-cloud-contract-amqp-test-0.4.0-SNAPSHOT.pom
│   │       │   │   └── spring-cloud-contract-amqp-test-0.4.0-SNAPSHOT-
│   │       │       ├── maven-metadata-local.xml
│   │       │       └── maven-metadata-local.xml
│   └── stubs.jar
```

Further assume that the stubs contain the following structure:

```

├── META-INF
│   └── MANIFEST.MF
└── contracts
    └── shouldProduceValidPersonData.groovy

```

Then consider the following contract:

```

Contract.make {
    // Human readable description
    description 'Should produce valid person data'
    // Label by means of which the output message can be triggered
    label 'contract-test.person.created.event'
    // input to the contract
    input {
        // the contract will be triggered by a method
        triggeredBy('createPerson()')
    }
    // output message of the contract
    outputMessage {
        // destination to which the output message will be sent
        sentTo 'contract-test.exchange'
        headers {
            header('contentType': 'application/json')
            header('__TypeId__':
'org.springframework.cloud.contract.stubrunner.messaging.amqp.Person')
        }
        // the body of the output message
        body([
            id : $(consumer(9), producer(regex("[0-9]+"))),
            name: "me"
        ])
    }
}

```

Now consider the following Spring configuration:

```
stubrunner:
  repositoryRoot: classpath:m2repo/repository/
  ids: org.springframework.cloud.contract.verifier.stubs.amqp:spring-cloud-
contract-amqp-test:0.4.0-SNAPSHOT:stubs
  stubs-mode: remote
  amqp:
    enabled: true
  server:
    port: 0
```

Triggering the Message

To trigger a message using the contract in the preceding section, use the `StubTrigger` interface as follows:

```
stubTrigger.trigger("contract-test.person.created.event")
```

The message has a destination of `contract-test.exchange`, so the Spring AMQP stub runner integration looks for bindings related to this exchange, as the following example shows:

```
@Bean
public Binding binding() {
    return BindingBuilder.bind(new Queue("test.queue"))
        .to(new DirectExchange("contract-test.exchange")).with("#");
}
```

The binding definition binds the queue called `test.queue`. As a result, the following listener definition is matched and invoked with the contract message:

```

@Bean
public SimpleMessageListenerContainer simpleMessageListenerContainer(
    ConnectionFactory connectionFactory,
    MessageListenerAdapter listenerAdapter) {
    SimpleMessageListenerContainer container = new
SimpleMessageListenerContainer();
    container.setConnectionFactory(connectionFactory);
    container.setQueueNames("test.queue");
    container.setMessageListener(listenerAdapter);

    return container;
}

```

Also, the following annotated listener matches and is invoked:

```

@RabbitListener(bindings = @QueueBinding(value = @Queue("test.queue"),
    exchange = @Exchange(value = "contract-test.exchange",
        ignoreDeclarationExceptions = "true")))
public void handlePerson(Person person) {
    this.person = person;
}

```



The message is directly handed over to the `onMessage` method of the `MessageListener` associated with the matching `SimpleMessageListenerContainer`.

Spring AMQP Test Configuration

In order to avoid Spring AMQP trying to connect to a running broker during our tests, we configure a mock `ConnectionFactory`.

To disable the mocked `ConnectionFactory`, set the following property: `stubrunner.amqp.mockConnection=false`, as follows:

```

stubrunner:
  amqp:
    mockConnection: false

```

Consumer Side Messaging With Spring JMS

Spring Cloud Contract Stub Runner's messaging module provides an easy way to integrate with Spring JMS.

The integration assumes that you have a running instance of a JMS broker (e.g. `activemq` embedded broker).

Adding the Runner to the Project

You need to have both Spring JMS and Spring Cloud Contract Stub Runner on the classpath. Remember to annotate your test class with `@AutoConfigureStubRunner`.

Examples

Assume that the stub structure looks as follows:

```
├── stubs
│   ├── bookDeleted.groovy
│   ├── bookReturned1.groovy
│   └── bookReturned2.groovy
```

Further assume the following test configuration:

```
stubrunner:
  repository-root: stubs:classpath:/stubs/
  ids: my:stubs
  stubs-mode: remote
spring:
  activemq:
    send-timeout: 1000
  jms:
    template:
      receive-timeout: 1000
```

Now consider the following contracts (we number them 1 and 2):

```

Contract.make {
  label 'return_book_1'
  input {
    triggeredBy('bookReturnedTriggered()')
  }
  outputMessage {
    sentTo('output')
    body(''{ "bookName" : "foo" }''')
    headers {
      header('BOOK-NAME', 'foo')
    }
  }
}

```

```

Contract.make {
  label 'return_book_2'
  input {
    messageFrom('input')
    messageBody([
      bookName: 'foo'
    ])
    messageHeaders {
      header('sample', 'header')
    }
  }
  outputMessage {
    sentTo('output')
    body([
      bookName: 'foo'
    ])
    headers {
      header('BOOK-NAME', 'foo')
    }
  }
}

```

Scenario 1 (No Input Message)

To trigger a message from the `return_book_1` label, we use the `StubTrigger` interface, as follows:

```
stubFinder.trigger('return_book_1')
```

Next, we want to listen to the output of the message sent to `output`:

```
TextMessage receivedMessage = (TextMessage) jmsTemplate.receive('output')
```

The received message would then pass the following assertions:

```
receivedMessage != null  
assertThatBodyContainsBookNameFoo(receivedMessage.getText())  
receivedMessage.getStringProperty('BOOK-NAME') == 'foo'
```

Scenario 2 (Output Triggered by Input)

Since the route is set for you, you can send a message to the **output** destination.

```
jmsTemplate.  
    convertAndSend('input', new BookReturned('foo'), new  
MessagePostProcessor() {  
    @Override  
    Message postProcessMessage(Message message) throws JMSEException {  
        message.setStringProperty("sample", "header")  
        return message  
    }  
})
```

Next, we want to listen to the output of the message sent to **output**, as follows:

```
TextMessage receivedMessage = (TextMessage) jmsTemplate.receive('output')
```

The received message would pass the following assertions:

```
receivedMessage != null  
assertThatBodyContainsBookNameFoo(receivedMessage.getText())  
receivedMessage.getStringProperty('BOOK-NAME') == 'foo'
```

Scenario 3 (Input with No Output)

Since the route is set for you, you can send a message to the **output** destination, as follows:


```

jmsTemplate.
    convertAndSend('delete', new BookReturned('foo'), new
MessagePostProcessor() {
    @Override
    Message postProcessMessage(Message message) throws JMSEException {
        message.setStringProperty("sample", "header")
        return message
    }
})

```

Consumer Side Messaging With Spring Kafka

Spring Cloud Contract Stub Runner's messaging module provides an easy way to integrate with Spring Kafka.

The integration assumes that you have a running instance of a embedded Kafka broker (via the `spring-kafka-test` dependency).

Adding the Runner to the Project

You need to have both Spring Kafka, Spring Kafka Test (to run the `@EmbeddedBroker`) and Spring Cloud Contract Stub Runner on the classpath. Remember to annotate your test class with `@AutoConfigureStubRunner`.

With Kafka integration, in order to poll for a single message we need to register a consumer upon Spring context startup. That may lead to a situation that, when you're on the consumer side, Stub Runner can register an additional consumer for the same group id and topic. That could lead to a situation that only one of the components would actually poll for the message. Since on the consumer side you have both the Spring Cloud Contract Stub Runner and Spring Cloud Contract Verifier classpath, we need to be able to switch off such behaviour. That's done automatically via the `stubrunner.kafka.initializer.enabled` flag, that will disable the Contact Verifier consumer registration. If your application is both the consumer and the producer of a kafka message, you might need to manually toggle that property to `false` in the base class of your generated tests.

Examples

Assume that the stub structure looks as follows:

```

├── stubs
│   ├── bookDeleted.groovy
│   ├── bookReturned1.groovy
│   └── bookReturned2.groovy

```

Further assume the following test configuration (notice the `spring.kafka.bootstrap-servers` pointing to the embedded broker's IP via `spring.embedded.kafka.brokers`):

```
stubrunner:
  repository-root: stubs:classpath:/stubs/
  ids: my:stubs
  stubs-mode: remote
spring:
  kafka:
    bootstrap-servers: ${spring.embedded.kafka.brokers}
    producer:
      properties:
        "value.serializer":
"org.springframework.kafka.support.serializer.JsonSerializer"
        "spring.json.trusted.packages": "*"
    consumer:
      properties:
        "value.deserializer":
"org.springframework.kafka.support.serializer.JsonDeserializer"
        "value.serializer":
"org.springframework.kafka.support.serializer.JsonSerializer"
        "spring.json.trusted.packages": "*"
    group-id: groupId
```

Now consider the following contracts (we number them 1 and 2):

```

Contract.make {
  label 'return_book_1'
  input {
    triggeredBy('bookReturnedTriggered()')
  }
  outputMessage {
    sentTo('output')
    body(''{ "bookName" : "foo" }''')
    headers {
      header('BOOK-NAME', 'foo')
    }
  }
}

```

```

Contract.make {
  label 'return_book_2'
  input {
    messageFrom('input')
    messageBody([
      bookName: 'foo'
    ])
    messageHeaders {
      header('sample', 'header')
    }
  }
  outputMessage {
    sentTo('output')
    body([
      bookName: 'foo'
    ])
    headers {
      header('BOOK-NAME', 'foo')
    }
  }
}

```

Scenario 1 (No Input Message)

To trigger a message from the `return_book_1` label, we use the `StubTrigger` interface, as follows:

```
stubFinder.trigger('return_book_1')
```

Next, we want to listen to the output of the message sent to `output`:

```
Message receivedMessage = receiveFromOutput()
```

The received message would then pass the following assertions:

```
assert receivedMessage != null
assert assertThatBodyContainsBookNameFoo(receivedMessage.getPayload())
assert receivedMessage.getHeaders().get('BOOK-NAME') == 'foo'
```

Scenario 2 (Output Triggered by Input)

Since the route is set for you, you can send a message to the **output** destination.

```
Message message = MessageBuilder.createMessage(new BookReturned('foo'), new
MessageHeaders([sample: "header",]))
kafkaTemplate.setDefaultTopic('input')
kafkaTemplate.send(message)
```

Next, we want to listen to the output of the message sent to **output**, as follows:

```
Message receivedMessage = receiveFromOutput()
```

The received message would pass the following assertions:

```
assert receivedMessage != null
assert assertThatBodyContainsBookNameFoo(receivedMessage.getPayload())
assert receivedMessage.getHeaders().get('BOOK-NAME') == 'foo'
```

Scenario 3 (Input with No Output)

Since the route is set for you, you can send a message to the **output** destination, as follows:

```
Message message = MessageBuilder.createMessage(new BookReturned('foo'), new
MessageHeaders([sample: "header",]))
kafkaTemplate.setDefaultTopic('delete')
kafkaTemplate.send(message)
```

14.3.5. Spring Cloud Contract Stub Runner

One of the issues that you might encounter while using Spring Cloud Contract Verifier is passing the generated WireMock JSON stubs from the server side to the client side (or to various clients). The same takes place in terms of client-side generation for messaging.

Copying the JSON files and setting the client side for messaging manually is out of the question. That is why we introduced Spring Cloud Contract Stub Runner. It can automatically download and run the stubs for you.

Snapshot Versions

You can add the additional snapshot repository to your `build.gradle` file to use snapshot versions, which are automatically uploaded after every successful build, as follows:

Maven

```
<repositories>
  <repository>
    <id>spring-snapshots</id>
    <name>Spring Snapshots</name>
    <url>https://repo.spring.io/snapshot</url>
    <snapshots>
      <enabled>true</enabled>
    </snapshots>
  </repository>
  <repository>
    <id>spring-milestones</id>
    <name>Spring Milestones</name>
    <url>https://repo.spring.io/milestone</url>
    <snapshots>
      <enabled>>false</enabled>
    </snapshots>
  </repository>
  <repository>
    <id>spring-releases</id>
    <name>Spring Releases</name>
    <url>https://repo.spring.io/release</url>
    <snapshots>
      <enabled>>false</enabled>
    </snapshots>
  </repository>
</repositories>
```

```
</repositories>
<pluginRepositories>
  <pluginRepository>
    <id>spring-snapshots</id>
    <name>Spring Snapshots</name>
    <url>https://repo.spring.io/snapshot</url>
    <snapshots>
      <enabled>true</enabled>
    </snapshots>
  </pluginRepository>
  <pluginRepository>
    <id>spring-milestones</id>
    <name>Spring Milestones</name>
    <url>https://repo.spring.io/milestone</url>
    <snapshots>
      <enabled>false</enabled>
    </snapshots>
  </pluginRepository>
  <pluginRepository>
    <id>spring-releases</id>
    <name>Spring Releases</name>
    <url>https://repo.spring.io/release</url>
    <snapshots>
      <enabled>false</enabled>
    </snapshots>
  </pluginRepository>
</pluginRepositories>
```

Gradle

```
/*  
We need to use the [buildscript {}] section when we have to modify  
the classpath for the plugins. If that's not the case this section  
can be skipped.  
  
If you don't need to modify the classpath (e.g. add a Pact dependency),  
then you can just set the [pluginManagement {}] section in [settings.gradle]  
file.
```

```
// settings.gradle  
pluginManagement {  
    repositories {  
        // for snapshots  
        maven {url "https://repo.spring.io/snapshot"}  
        // for milestones  
        maven {url "https://repo.spring.io/milestone"}  
        // for GA versions  
        gradlePluginPortal()  
    }  
}  
  
*/  
buildscript {  
    repositories {  
        mavenCentral()  
        mavenLocal()  
        maven { url "https://repo.spring.io/snapshot" }  
        maven { url "https://repo.spring.io/milestone" }  
        maven { url "https://repo.spring.io/release" }  
    }  
}
```

Publishing Stubs as JARs

The easiest approach to publishing stubs as jars is to centralize the way stubs are kept. For example, you can keep them as jars in a Maven repository.



For both Maven and Gradle, the setup comes ready to work. However, you can customize it if you want to.

The following example shows how to publish stubs as jars:

Maven

```
<!-- First disable the default jar setup in the properties section -->  
<!-- we don't want the verifier to do a jar for us -->  
<spring.cloud.contract.verifier.skip>true</spring.cloud.contract.verifier.skip>
```

```

<!-- Next add the assembly plugin to your build -->
<!-- we want the assembly plugin to generate the JAR -->
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-assembly-plugin</artifactId>
  <executions>
    <execution>
      <id>stub</id>
      <phase>prepare-package</phase>
      <goals>
        <goal>single</goal>
      </goals>
      <inherited>>false</inherited>
      <configuration>
        <attach>>true</attach>
        <descriptors>
          <descriptor>
            <id>stub</id>
            <location>
              <file>
                <url>
                  <value>
                    $/home/marcin/repo/spring-cloud-scripts/src/assembly/stub.xml
                  </value>
                </url>
              </file>
            </location>
          </descriptor>
        </descriptors>
      </configuration>
    </execution>
  </executions>
</plugin>

```

<!-- Finally setup your assembly. Below you can find the contents of src/main/assembly/stub.xml -->

```

<assembly
  xmlns="http://maven.apache.org/plugins/maven-assembly-plugin/assembly/1.1.3"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/plugins/maven-assembly-
plugin/assembly/1.1.3 http://maven.apache.org/xsd/assembly-1.1.3.xsd">
  <id>stubs</id>
  <formats>
    <format>jar</format>
  </formats>
  <includeBaseDirectory>>false</includeBaseDirectory>
  <fileSets>
    <fileSet>
      <directory>src/main/java</directory>
      <outputDirectory>/</outputDirectory>
      <includes>
        <include>**com/example/model/*.*</include>
      </includes>
    </fileSet>
    <fileSet>
      <directory>${project.build.directory}/classes</directory>
      <outputDirectory>/</outputDirectory>
      <includes>
        <include>**com/example/model/*.*</include>
      </includes>
    </fileSet>
  </fileSets>
</assembly>

```



```

    <fileSet>
      <directory>${project.build.directory}/snippets/stubs</directory>
      <outputDirectory>META-
INF/${project.groupId}/${project.artifactId}/${project.version}/mappings</outputDi
rectory>
      <includes>
        <include>**/*</include>
      </includes>
    </fileSet>
    <fileSet>
      <directory>$/home/marcin/repo/spring-cloud-
scripts/src/test/resources/contracts</directory>
      <outputDirectory>META-
INF/${project.groupId}/${project.artifactId}/${project.version}/contracts</outputD
irectory>
      <includes>
        <include>**/*.groovy</include>
      </includes>
    </fileSet>
  </fileSets>
</assembly>

```

Gradle

```

ext {
  contractsDir = file("mappings")
  stubsOutputDirRoot = file("${project.buildDir}/production/${project.name}-
stubs/")
}

// Automatically added by plugin:
// copyContracts - copies contracts to the output folder from which JAR will be
created
// verifierStubsJar - JAR with a provided stub suffix
// the presented publication is also added by the plugin but you can modify it as
you wish

publishing {
  publications {
    stubs(MavenPublication) {
      artifactId "${project.name}-stubs"
      artifact verifierStubsJar
    }
  }
}
}

```

Stub Runner Core

The stub runner core runs stubs for service collaborators. Treating stubs as contracts of services

lets you use stub-runner as an implementation of [Consumer-driven Contracts](#).

Stub Runner lets you automatically download the stubs of the provided dependencies (or pick those from the classpath), start WireMock servers for them, and feed them with proper stub definitions. For messaging, special stub routes are defined.

Retrieving stubs

You can pick from the following options of acquiring stubs:

- Aether-based solution that downloads JARs with stubs from Artifactory or Nexus
- Classpath-scanning solution that searches the classpath with a pattern to retrieve stubs
- Writing your own implementation of the `org.springframework.cloud.contract.stubrunner.StubDownloaderBuilder` for full customization

The latter example is described in the [Custom Stub Runner](#) section.

Downloading Stubs

You can control the downloading of stubs with the `stubsMode` switch. It picks value from the `StubRunnerProperties.StubsMode` enumeration. You can use the following options:

- `StubRunnerProperties.StubsMode.CLASSPATH` (default value): Picks stubs from the classpath
- `StubRunnerProperties.StubsMode.LOCAL`: Picks stubs from a local storage (for example, `.m2`)
- `StubRunnerProperties.StubsMode.REMOTE`: Picks stubs from a remote location

The following example picks stubs from a local location:

```
@AutoConfigureStubRunner(repositoryRoot="https://foo.bar", ids =
"com.example:beer-api-producer:+:stubs:8095", stubsMode =
StubRunnerProperties.StubsMode.LOCAL)
```

Classpath scanning

If you set the `stubsMode` property to `StubRunnerProperties.StubsMode.CLASSPATH` (or set nothing since `CLASSPATH` is the default value), the classpath is scanned. Consider the following example:

```
@AutoConfigureStubRunner(ids = {
    "com.example:beer-api-producer:+:stubs:8095",
    "com.example.foo:bar:1.0.0:superstubs:8096"
})
```

You can add the dependencies to your classpath, as follows:

Maven

```
<dependency>
  <groupId>com.example</groupId>
  <artifactId>beer-api-producer-restdocs</artifactId>
  <classifier>stubs</classifier>
  <version>0.0.1-SNAPSHOT</version>
  <scope>test</scope>
  <exclusions>
    <exclusion>
      <groupId>*</groupId>
      <artifactId>*</artifactId>
    </exclusion>
  </exclusions>
</dependency>
<dependency>
  <groupId>com.example.thing1</groupId>
  <artifactId>thing2</artifactId>
  <classifier>superstubs</classifier>
  <version>1.0.0</version>
  <scope>test</scope>
  <exclusions>
    <exclusion>
      <groupId>*</groupId>
      <artifactId>*</artifactId>
    </exclusion>
  </exclusions>
</dependency>
```

Gradle

```
testCompile("com.example:beer-api-producer-restdocs:0.0.1-SNAPSHOT:stubs") {
  transitive = false
}
testCompile("com.example.thing1:thing2:1.0.0:superstubs") {
  transitive = false
}
```

Then the specified locations on your classpath get scanned. For `com.example:beer-api-producer-restdocs`, the following locations are scanned:

- `/META-INF/com.example/beer-api-producer-restdocs/*/*`
- `/contracts/com.example/beer-api-producer-restdocs/*/*`
- `/mappings/com.example/beer-api-producer-restdocs/*/*`

For `com.example.thing1:thing2`, the following locations are scanned:

- /META-INF/com.example.thing1/thing2/*/*
- /contracts/com.example.thing1/thing2/*/*
- /mappings/com.example.thing1/thing2/*/*



You have to explicitly provide the group and artifact IDs when you package the producer stubs.

To achieve proper stub packaging, the producer would set up the contracts as follows:

```

├── src
│   ├── test
│   │   ├── resources
│   │   │   ├── contracts
│   │   │   │   ├── com.example
│   │   │   │   │   ├── beer-api-producer-restdocs
│   │   │   │   │   │   ├── nested
│   │   │   │   │   │   │   └── contract3.groovy

```

By using the [Maven assembly plugin](#) or [Gradle Jar](#) task, you have to create the following structure in your stubs jar:

```

├── META-INF
│   ├── com.example
│   │   ├── beer-api-producer-restdocs
│   │   │   ├── 2.0.0
│   │   │   │   ├── contracts
│   │   │   │   │   ├── nested
│   │   │   │   │   │   └── contract2.groovy
│   │   │   ├── mappings
│   │   │   │   └── mapping.json

```

By maintaining this structure, the classpath gets scanned and you can profit from the messaging or HTTP stubs without the need to download artifacts.

Configuring HTTP Server Stubs

Stub Runner has a notion of a `HttpServerStub` that abstracts the underlying concrete implementation of the HTTP server (for example, WireMock is one of the implementations). Sometimes, you need to perform some additional tuning (which is concrete for the given implementation) of the stub servers. To do that, Stub Runner gives you the `httpServerStubConfigurer` property that is available in the annotation and the JUnit rule and is accessible through system properties, where you can provide your implementation of the `org.springframework.cloud.contract.stubrunner.HttpServerStubConfigurer` interface. The

implementations can alter the configuration files for the given HTTP server stub.

Spring Cloud Contract Stub Runner comes with an implementation that you can extend for WireMock:

`org.springframework.cloud.contract.stubrunner.provider.wiremock.WireMockHttpServerStubConfigurer`. In the `configure` method, you can provide your own custom configuration for the given stub. The use case might be starting WireMock for the given artifact ID, on an HTTPS port. The following example shows how to do so:

Example 5. WireMockHttpServerStubConfigurer implementation

```
@CompileStatic
static class HttpsForFraudDetection extends WireMockHttpServerStubConfigurer {

    private static final Log log = LoggerFactory.getLog(HttpsForFraudDetection)

    @Override
    WireMockConfiguration configure(WireMockConfiguration httpStubConfiguration,
HttpServerStubConfiguration httpServerStubConfiguration) {
        if (httpServerStubConfiguration.stubConfiguration.artifactId ==
"fraudDetectionServer") {
            int httpsPort = SocketUtils.findAvailableTcpPort()
            log.info("Will set HTTPS port [" + httpsPort + "] for fraud detection
server")
            return httpStubConfiguration
                .httpsPort(httpsPort)
        }
        return httpStubConfiguration
    }
}
```

You can then reuse it with the `@AutoConfigureStubRunner` annotation, as follows:

```
@AutoConfigureStubRunner(mappingsOutputFolder = "target/outputmappings/",
    httpServerStubConfigurer = HttpsForFraudDetection)
```

Whenever an HTTPS port is found, it takes precedence over the HTTP port.

Running stubs

This section describes how to run stubs. It contains the following topics:

- [HTTP Stubs](#)
- [Viewing Registered Mappings](#)
- [Messaging Stubs](#)

HTTP Stubs

Stubs are defined in JSON documents, whose syntax is defined in [WireMock documentation](#)

The following example defines a stub in JSON:

```
{
  "request": {
    "method": "GET",
    "url": "/ping"
  },
  "response": {
    "status": 200,
    "body": "pong",
    "headers": {
      "Content-Type": "text/plain"
    }
  }
}
```

Viewing Registered Mappings

Every stubbed collaborator exposes a list of defined mappings under the `__/admin/` endpoint.

You can also use the `mappingsOutputFolder` property to dump the mappings to files. For the annotation-based approach, it would resemble the following example:

```
@AutoConfigureStubRunner(ids="a.b.c:loanIssuance,a.b.c:fraudDetectionServer",
mappingsOutputFolder = "target/outputmappings/")
```

For the JUnit approach, it resembles the following example:

```
@ClassRule @Shared StubRunnerRule rule = new StubRunnerRule()
    .repoRoot("https://some_url")
    .downloadStub("a.b.c", "loanIssuance")
    .downloadStub("a.b.c:fraudDetectionServer")
    .withMappingsOutputFolder("target/outputmappings")
```

Then, if you check out the `target/outputmappings` folder, you would see the following structure;

```
.
├── fraudDetectionServer_13705
└── loanIssuance_12255
```

That means that there were two stubs registered. `fraudDetectionServer` was registered at port `13705` and `loanIssuance` at port `12255`. If we take a look at one of the files, we would see (for WireMock) the mappings available for the given server:

```
[{
  "id" : "f9152eb9-bf77-4c38-8289-90be7d10d0d7",
  "request" : {
    "url" : "/name",
    "method" : "GET"
  },
  "response" : {
    "status" : 200,
    "body" : "fraudDetectionServer"
  },
  "uuid" : "f9152eb9-bf77-4c38-8289-90be7d10d0d7"
},
...
]
```

Messaging Stubs

Depending on the provided Stub Runner dependency and the DSL, the messaging routes are automatically set up.

Stub Runner JUnit Rule and Stub Runner JUnit5 Extension

Stub Runner comes with a JUnit rule that lets you can download and run stubs for a given group and artifact ID, as the following example shows:

```

@ClassRule
public static StubRunnerRule rule = new StubRunnerRule().repoRoot(repoRoot())
    .stubsMode(StubRunnerProperties.StubsMode.REMOTE)
    .downloadStub("org.springframework.cloud.contract.verifier.stubs",
        "loanIssuance")
    .downloadStub(

"org.springframework.cloud.contract.verifier.stubs:fraudDetectionServer");

@BeforeClass
@AfterClass
public static void setupProps() {
    System.clearProperty("stubrunner.repository.root");
    System.clearProperty("stubrunner.classifier");
}

```

A `StubRunnerExtension` is also available for JUnit 5. `StubRunnerRule` and `StubRunnerExtension` work in a very similar fashion. After the rule or extension is executed, Stub Runner connects to your Maven repository and, for the given list of dependencies, tries to:

- Download them
- Cache them locally
- Unzip them to a temporary folder
- Start a WireMock server for each Maven dependency on a random port from the provided range of ports or the provided port
- Feed the WireMock server with all JSON files that are valid WireMock definitions
- Send messages (remember to pass an implementation of `MessageVerifier` interface)

Stub Runner uses the [Eclipse Aether](#) mechanism to download the Maven dependencies. Check their [docs](#) for more information.

Since the `StubRunnerRule` and `StubRunnerExtension` implement the `StubFinder` they let you find the started stubs, as the following example shows:

```

package org.springframework.cloud.contract.stubrunner;

import java.net.URL;
import java.util.Collection;
import java.util.Map;

import org.springframework.cloud.contract.spec.Contract;

/**
 * Contract for finding registered stubs.

```



```

*
* @author Marcin Grzejszczak
*/
public interface StubFinder extends StubTrigger {

    /**
     * For the given groupId and artifactId tries to find the matching URL of the
     running
     * stub.
     * @param groupId - might be null. In that case a search only via artifactId
     takes
     * place
     * @param artifactId - artifact id of the stub
     * @return URL of a running stub or throws exception if not found
     * @throws StubNotFoundException in case of not finding a stub
     */
    URL findStubUrl(String groupId, String artifactId) throws
    StubNotFoundException;

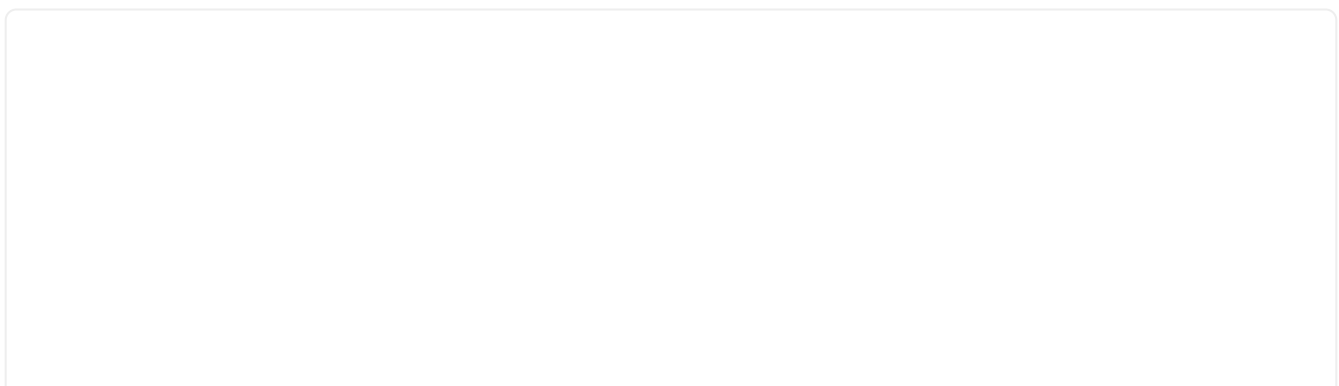
    /**
     * For the given Ivy notation {@code
    [groupId]:artifactId:[version]:[classifier]}
     * tries to find the matching URL of the running stub. You can also pass only
     * {@code artifactId}.
     * @param ivyNotation - Ivy representation of the Maven artifact
     * @return URL of a running stub or throws exception if not found
     * @throws StubNotFoundException in case of not finding a stub
     */
    URL findStubUrl(String ivyNotation) throws StubNotFoundException;

    /**
     * @return all running stubs
     */
    RunningStubs findAllRunningStubs();

    /**
     * @return the list of Contracts
     */
    Map<StubConfiguration, Collection<Contract>> getContracts();
}

```

The following examples provide more detail about using Stub Runner:



spock

```
@ClassRule
@Shared
StubRunnerRule rule = new StubRunnerRule()
    .stubsMode(StubRunnerProperties.StubsMode.REMOTE)

.repoRoot(StubRunnerRuleSpec.getResource("/m2repo/repository").toURI().toString())
    .downloadStub("org.springframework.cloud.contract.verifier.stubs",
"loanIssuance")

.downloadStub("org.springframework.cloud.contract.verifier.stubs:fraudDetectionSer
ver")
    .withMappingsOutputFolder("target/outputmappingsforrule")

def 'should start WireMock servers'() {
    expect: 'WireMocks are running'
        rule.findStubUrl('org.springframework.cloud.contract.verifier.stubs',
'loanIssuance') != null
        rule.findStubUrl('loanIssuance') != null
        rule.findStubUrl('loanIssuance') ==
rule.findStubUrl('org.springframework.cloud.contract.verifier.stubs',
'loanIssuance')

rule.findStubUrl('org.springframework.cloud.contract.verifier.stubs:fraudDetection
Server') != null
    and:
        rule.findAllRunningStubs().isPresent('loanIssuance')

rule.findAllRunningStubs().isPresent('org.springframework.cloud.contract.verifier.
stubs', 'fraudDetectionServer')

rule.findAllRunningStubs().isPresent('org.springframework.cloud.contract.verifier.
stubs:fraudDetectionServer')
    and: 'Stubs were registered'
        "${rule.findStubUrl('loanIssuance').toString()}/name".toURL().text ==
'loanIssuance'
        "${rule.findStubUrl('fraudDetectionServer').toString()}/name".toURL().text
== 'fraudDetectionServer'
}

def 'should output mappings to output folder'() {
    when:
        def url = rule.findStubUrl('fraudDetectionServer')
    then:
        new File("target/outputmappingsforrule",
"fraudDetectionServer_${url.port}").exists()
}
```

```
@Test
public void should_start_wiremock_servers() throws Exception {
    // expect: 'WireMocks are running'
    then(rule.findStubUrl("org.springframework.cloud.contract.verifier.stubs",
        "loanIssuance")).isNotNull();
    then(rule.findStubUrl("loanIssuance")).isNotNull();
    then(rule.findStubUrl("loanIssuance").isEqualTo(rule.findStubUrl(
        "org.springframework.cloud.contract.verifier.stubs", "loanIssuance")));
    then(rule.findStubUrl(
        "org.springframework.cloud.contract.verifier.stubs:fraudDetectionServer"))
        .isNotNull();
    // and:
    then(rule.findAllRunningStubs().isPresent("loanIssuance")).isTrue();
    then(rule.findAllRunningStubs().isPresent(
        "org.springframework.cloud.contract.verifier.stubs",
        "fraudDetectionServer")).isTrue();
    then(rule.findAllRunningStubs().isPresent(
        "org.springframework.cloud.contract.verifier.stubs:fraudDetectionServer"))
        .isTrue();
    // and: 'Stubs were registered'
    then(httpGet(rule.findStubUrl("loanIssuance").toString() + "/name"))
        .isEqualTo("loanIssuance");
    then(httpGet(rule.findStubUrl("fraudDetectionServer").toString() + "/name"))
        .isEqualTo("fraudDetectionServer");
}
```

```

// Visible for JUnit
@RegisterExtension
static StubRunnerExtension stubRunnerExtension = new StubRunnerExtension()
    .repoRoot(repoRoot()).stubsMode(StubRunnerProperties.StubsMode.REMOTE)
    .downloadStub("org.springframework.cloud.contract.verifier.stubs",
        "loanIssuance")
    .downloadStub(
        "org.springframework.cloud.contract.verifier.stubs:fraudDetectionServer")
    .withMappingsOutputFolder("target/outputmappingsforrule");

@BeforeAll
@AfterAll
static void setupProps() {
    System.clearProperty("stubrunner.repository.root");
    System.clearProperty("stubrunner.classifier");
}

private static String repoRoot() {
    try {
        return StubRunnerRuleJUnitTest.class.getResource("/m2repo/repository/")
            .toURI().toString();
    }
    catch (Exception e) {
        return "";
    }
}
}

```

See the [Common Properties for JUnit and Spring](#) for more information on how to apply global configuration of Stub Runner.



To use the JUnit rule or JUnit 5 extension together with messaging, you have to provide an implementation of the `MessageVerifier` interface to the rule builder (for example, `rule.messageVerifier(new MyMessageVerifier())`). If you do not do this, then, whenever you try to send a message, an exception is thrown.

Maven Settings

The stub downloader honors Maven settings for a different local repository folder. Authentication details for repositories and profiles are currently not taken into account, so you need to specify it by using the properties mentioned above.

Providing Fixed Ports

You can also run your stubs on fixed ports. You can do it in two different ways. One is to pass it in the properties, and the other is to use the fluent API of JUnit rule.

Fluent API

When using the `StubRunnerRule` or `StubRunnerExtension`, you can add a stub to download and then pass the port for the last downloaded stub. The following example shows how to do so:

```
@ClassRule
public static StubRunnerRule rule = new StubRunnerRule().repoRoot(repoRoot())
    .stubsMode(StubRunnerProperties.StubsMode.REMOTE)
    .downloadStub("org.springframework.cloud.contract.verifier.stubs",
        "loanIssuance")
    .withPort(12345).downloadStub(

"org.springframework.cloud.contract.verifier.stubs:fraudDetectionServer:12346");

@BeforeClass
@AfterClass
public static void setupProps() {
    System.clearProperty("stubrunner.repository.root");
    System.clearProperty("stubrunner.classifier");
}
```

For the preceding example, the following test is valid:

```
then(rule.findStubUrl("loanIssuance"))
    .isEqualTo(URI.create("http://localhost:12345").toURL());
then(rule.findStubUrl("fraudDetectionServer"))
    .isEqualTo(URI.create("http://localhost:12346").toURL());
```

Stub Runner with Spring

Stub Runner with Spring sets up Spring configuration of the Stub Runner project.

By providing a list of stubs inside your configuration file, Stub Runner automatically downloads and registers in WireMock the selected stubs.

If you want to find the URL of your stubbed dependency, you can autowire the `StubFinder` interface and use its methods, as follows:

```
@ContextConfiguration(classes = Config, loader = SpringBootTestLoader)
@SpringBootTest(properties = [" stubrunner.cloud.enabled=false",
    'foo=${stubrunner.runningstubs.fraudDetectionServer.port}',

'fooWithGroup=${stubrunner.runningstubs.org.springframework.cloud.contract.verifie
r.stubs.fraudDetectionServer.port}'])
```

```

@AutoConfigureStubRunner(mappingsOutputFolder = "target/outputmappings/",
    httpServerStubConfigurer = HttpsForFraudDetection)
@ActiveProfiles("test")
class StubRunnerConfigurationSpec extends Specification {

    @Autowired
    StubFinder stubFinder
    @Autowired
    Environment environment
    @StubRunnerPort("fraudDetectionServer")
    int fraudDetectionServerPort

    @StubRunnerPort("org.springframework.cloud.contract.verifier.stubs:fraudDetectionServer")
    int fraudDetectionServerPortWithGroupId
    @Value('${foo}')
    Integer foo

    @BeforeClass
    @AfterClass
    void setupProps() {
        System.clearProperty("stubrunner.repository.root")
        System.clearProperty("stubrunner.classifier")
        WireMockHttpServerStubAccessor.clear()
    }

    def 'should mark all ports as random'() {
        expect:
            WireMockHttpServerStubAccessor.everyPortRandom()
    }

    def 'should start WireMock servers'() {
        expect: 'WireMocks are running'
    }

    stubFinder.findStubUrl('org.springframework.cloud.contract.verifier.stubs',
        'loanIssuance') != null
        stubFinder.findStubUrl('loanIssuance') != null
        stubFinder.findStubUrl('loanIssuance') ==
    stubFinder.findStubUrl('org.springframework.cloud.contract.verifier.stubs',
        'loanIssuance')
        stubFinder.findStubUrl('loanIssuance') ==
    stubFinder.findStubUrl('org.springframework.cloud.contract.verifier.stubs:loanIssuance')

    stubFinder.findStubUrl('org.springframework.cloud.contract.verifier.stubs:loanIssuance:0.0.1-SNAPSHOT') ==
    stubFinder.findStubUrl('org.springframework.cloud.contract.verifier.stubs:loanIssuance:0.0.1-SNAPSHOT:stubs')

    stubFinder.findStubUrl('org.springframework.cloud.contract.verifier.stubs:fraudDetectionServer') != null

```

```

        and:
            stubFinder.findAllRunningStubs().isPresent('loanIssuance')

stubFinder.findAllRunningStubs().isPresent('org.springframework.cloud.contract.ver
ifier.stubs', 'fraudDetectionServer')

stubFinder.findAllRunningStubs().isPresent('org.springframework.cloud.contract.ver
ifier.stubs:fraudDetectionServer')
    and: 'Stubs were registered'

"${stubFinder.findStubUrl('loanIssuance').toString()}/name".toURL().text ==
'loanIssuance'

"${stubFinder.findStubUrl('fraudDetectionServer').toString()}/name".toURL().text
== 'fraudDetectionServer'
    and: 'Fraud Detection is an HTTPS endpoint'

stubFinder.findStubUrl('fraudDetectionServer').toString().startsWith("https")
}

def 'should throw an exception when stub is not found'() {
    when:
        stubFinder.findStubUrl('nonExistingService')
    then:
        thrown(StubNotFoundException)
    when:
        stubFinder.findStubUrl('nonExistingGroupId', 'nonExistingArtifactId')
    then:
        thrown(StubNotFoundException)
}

def 'should register started servers as environment variables'() {
    expect:
        environment.getProperty("stubrunner.runningstubs.loanIssuance.port")
!= null
        stubFinder.findAllRunningStubs().getPort("loanIssuance") ==
(environment.getProperty("stubrunner.runningstubs.loanIssuance.port") as Integer)
        and:

environment.getProperty("stubrunner.runningstubs.fraudDetectionServer.port") !=
null
        stubFinder.findAllRunningStubs().getPort("fraudDetectionServer") ==
(environment.getProperty("stubrunner.runningstubs.fraudDetectionServer.port") as
Integer)
        and:

environment.getProperty("stubrunner.runningstubs.fraudDetectionServer.port") !=
null
        stubFinder.findAllRunningStubs().getPort("fraudDetectionServer") ==
(environment.getProperty("stubrunner.runningstubs.org.springframework.cloud.contra
ct.verifier.stubs.fraudDetectionServer.port") as Integer)

```

```

    }

    def 'should be able to interpolate a running stub in the passed test
property'() {
        given:
            int fraudPort =
stubFinder.findAllRunningStubs().getPort("fraudDetectionServer")
        expect:
            fraudPort > 0
            environment.getProperty("foo", Integer) == fraudPort
            environment.getProperty("fooWithGroup", Integer) == fraudPort
            foo == fraudPort
    }

    @Issue("#573")
    def 'should be able to retrieve the port of a running stub via an
annotation'() {
        given:
            int fraudPort =
stubFinder.findAllRunningStubs().getPort("fraudDetectionServer")
        expect:
            fraudPort > 0
            fraudDetectionServerPort == fraudPort
            fraudDetectionServerPortWithGroupId == fraudPort
    }

    def 'should dump all mappings to a file'() {
        when:
            def url = stubFinder.findStubUrl("fraudDetectionServer")
        then:
            new File("target/outputmappings/",
"fraudDetectionServer_${url.port}").exists()
    }

    @Configuration
    @EnableAutoConfiguration
    static class Config {}

    @CompileStatic
    static class HttpsForFraudDetection extends WireMockHttpServerStubConfigurer {

        private static final Log log = LoggerFactory.getLog(HttpsForFraudDetection)

        @Override
        WireMockConfiguration configure(WireMockConfiguration
httpStubConfiguration, HttpServerStubConfiguration httpServerStubConfiguration) {
            if (httpServerStubConfiguration.stubConfiguration.artifactId ==
"fraudDetectionServer") {
                int httpsPort = SocketUtils.findAvailableTcpPort()
                log.info("Will set HTTPS port [" + httpsPort + "] for fraud
detection server")
            }
        }
    }

```



```

        return httpStubConfiguration
            .httpsPort(httpsPort)
    }
    return httpStubConfiguration
}
}
}

```

Doing so depends on the following configuration file:

```

stubrunner:
  repositoryRoot: classpath:m2repo/repository/
  ids:
    - org.springframework.cloud.contract.verifier.stubs:loanIssuance
    - org.springframework.cloud.contract.verifier.stubs:fraudDetectionServer
    - org.springframework.cloud.contract.verifier.stubs:bootService
  stubs-mode: remote

```

Instead of using the properties, you can also use the properties inside the `@AutoConfigureStubRunner`. The following example achieves the same result by setting values on the annotation:

```

@AutoConfigureStubRunner(
    ids = ["org.springframework.cloud.contract.verifier.stubs:loanIssuance",
"org.springframework.cloud.contract.verifier.stubs:fraudDetectionServer",
"org.springframework.cloud.contract.verifier.stubs:bootService"],
    stubsMode = StubRunnerProperties.StubsMode.REMOTE,
    repositoryRoot = "classpath:m2repo/repository/")

```

Stub Runner Spring registers environment variables in the following manner for every registered WireMock server. The following example shows Stub Runner IDs for `com.example:thing1` and `com.example:thing2`:

- `stubrunner.runningstubs.thing1.port`
- `stubrunner.runningstubs.com.example.thing1.port`
- `stubrunner.runningstubs.thing2.port`
- `stubrunner.runningstubs.com.example.thing2.port`

You can reference these values in your code.

You can also use the `@StubRunnerPort` annotation to inject the port of a running stub. The value of the annotation can be the `groupid:artifactid` or just the `artifactid`. The following example works shows Stub Runner IDs for `com.example:thing1` and `com.example:thing2`.

```
@StubRunnerPort("thing1")
int thing1Port;
@StubRunnerPort("com.example:thing2")
int thing2Port;
```

Stub Runner Spring Cloud

Stub Runner can integrate with Spring Cloud.

For real life examples, see:

- [The producer app sample](#)
- [The consumer app sample](#)

Stubbing Service Discovery

The most important feature of **Stub Runner Spring Cloud** is the fact that it stubs:

- **DiscoveryClient**
- **Ribbon ServerList**

That means that, regardless of whether you use Zookeeper, Consul, Eureka, or anything else, you do not need that in your tests. We are starting WireMock instances of your dependencies and we are telling your application, whenever you use **Feign**, to load a balanced **RestTemplate** or **DiscoveryClient** directly, to call those stubbed servers instead of calling the real Service Discovery tool.

For example, the following test passes:

```
def 'should make service discovery work'() {
    expect: 'WireMocks are running'
           "${stubFinder.findStubUrl('loanIssuance').toString()}/name".toURL().text
    == 'loanIssuance'

    "${stubFinder.findStubUrl('fraudDetectionServer').toString()}/name".toURL().text
    == 'fraudDetectionServer'
    and: 'Stubs can be reached via load service discovery'
        restTemplate.getForObject('http://loanIssuance/name', String) ==
    'loanIssuance'

    restTemplate.getForObject('http://someNameThatShouldMapFraudDetectionServer/name',
    String) == 'fraudDetectionServer'
}
```

Note that the preceding example requires the following configuration file:

```
stubrunner:
  idsToServiceIds:
    ivyNotation: someValueInsideYourCode
    fraudDetectionServer: someNameThatShouldMapFraudDetectionServer
```

Test Profiles and Service Discovery

In your integration tests, you typically do not want to call either a discovery service (such as Eureka) or Config Server. That is why you create an additional test configuration in which you want to disable these features.

Due to certain limitations of `spring-cloud-commons`, to achieve this, you have to disable these properties in a static block such as the following example (for Eureka):

```
//Hack to work around https://github.com/spring-cloud/spring-cloud-
commons/issues/156
static {
    System.setProperty("eureka.client.enabled", "false");
    System.setProperty("spring.cloud.config.failFast", "false");
}
```

Additional Configuration

You can match the `artifactId` of the stub with the name of your application by using the `stubrunner.idsToServiceIds`: map. You can disable Stub Runner Ribbon support by setting `stubrunner.cloud.ribbon.enabled` to `false`. You can disable Stub Runner support by setting `stubrunner.cloud.enabled` to `false`.



By default, all service discovery is stubbed. This means that, regardless of whether you have an existing `DiscoveryClient`, its results are ignored. However, if you want to reuse it, you can set `stubrunner.cloud.delegate.enabled` to `true`, and then your existing `DiscoveryClient` results are merged with the stubbed ones.

The default Maven configuration used by Stub Runner can be tweaked either by setting the following system properties or by setting the corresponding environment variables:

- `maven.repo.local`: Path to the custom maven local repository location
- `org.apache.maven.user-settings`: Path to custom maven user settings location
- `org.apache.maven.global-settings`: Path to maven global settings location

Using the Stub Runner Boot Application

Spring Cloud Contract Stub Runner Boot is a Spring Boot application that exposes REST endpoints to

trigger the messaging labels and to access WireMock servers.

One of the use cases is to run some smoke (end-to-end) tests on a deployed application. You can check out the [Spring Cloud Pipelines](#) project for more information.

Stub Runner Server

To use the Stub Runner Server, add the following dependency:

```
compile "org.springframework.cloud:spring-cloud-starter-stub-runner"
```

Then annotate a class with `@EnableStubRunnerServer`, build a fat jar, and it is ready to work.

For the properties, see the [Stub Runner Spring](#) section.

Stub Runner Server Fat Jar

You can download a standalone JAR from Maven (for example, for version 2.0.1.RELEASE) by running the following commands:

```
$ wget -O stub-runner.jar  
'https://search.maven.org/remotecontent?filepath=org/springframework/cloud/spring-  
cloud-contract-stub-runner-boot/2.0.1.RELEASE/spring-cloud-contract-stub-runner-  
boot-2.0.1.RELEASE.jar'  
$ java -jar stub-runner.jar --stubrunner.ids=... --stubrunner.repositoryRoot=...
```

Spring Cloud CLI

Starting from the 1.4.0.RELEASE version of the [Spring Cloud CLI](#) project, you can start Stub Runner Boot by running `spring cloud stubrunner`.

In order to pass the configuration, you can create a `stubrunner.yml` file in the current working directory, in a subdirectory called `config`, or in `~/.spring-cloud`. The file could resemble the following example for running stubs installed locally:

Example 6. `stubrunner.yml`

```
stubrunner:  
  stubsMode: LOCAL  
  ids:  
    - com.example:beer-api-producer:+:9876
```

Then you can call `spring cloud stubrunner` from your terminal window to start the Stub Runner server. It is available at port 8750.

Endpoints

Stub Runner Boot offers two endpoints:

- [HTTP](#)
- [Messaging](#)

HTTP

For HTTP, Stub Runner Boot makes the following endpoints available:

- GET `/stubs`: Returns a list of all running stubs in `ivy:integer` notation
- GET `/stubs/{ivy}`: Returns a port for the given `ivy` notation (when calling the endpoint `ivy` can also be `artifactId` only)

Messaging

For Messaging, Stub Runner Boot makes the following endpoints available:

- GET `/triggers`: Returns a list of all running labels in `ivy : [label1, label2 ...]` notation
- POST `/triggers/{label}`: Runs a trigger with `label`
- POST `/triggers/{ivy}/{label}`: Runs a trigger with a `label` for the given `ivy` notation (when calling the endpoint, `ivy` can also be `artifactId` only)

Example

The following example shows typical usage of Stub Runner Boot:

```
@ContextConfiguration(classes = StubRunnerBoot, loader = SpringBootTestLoader)
@SpringBootTest(properties = "spring.cloud.zookeeper.enabled=false")
@ActiveProfiles("test")
class StubRunnerBootSpec extends Specification {

    @Autowired
    StubRunning stubRunning

    def setup() {
        RestAssuredMockMvc.standaloneSetup(new HttpStubsController(stubRunning),
            new TriggerController(stubRunning))
    }

    def 'should return a list of running stub servers in "full ivy:port" notation'() {
        when:
            String response = RestAssuredMockMvc.get('/stubs').body.asString()
        then:
            def root = new JsonSlurper().parseText(response)
            root.'org.springframework.cloud.contract.verifier.stubs:bootService:0.0.1-SNAPSHOT:stubs' instanceof Integer
    }
}
```

```

def 'should return a port on which a [#stubId] stub is running'() {
  when:
    def response = RestAssuredMockMvc.get("/stubs/${stubId}")
  then:
    response.statusCode == 200
    Integer.valueOf(response.body.asString()) > 0
  where:
    stubId <<
['org.springframework.cloud.contract.verifier.stubs:bootService:++stubs',
'org.springframework.cloud.contract.verifier.stubs:bootService:0.0.1-SNAPSHOT:stubs',
'org.springframework.cloud.contract.verifier.stubs:bootService:+',
'org.springframework.cloud.contract.verifier.stubs:bootService',
  'bootService']
}

def 'should return 404 when missing stub was called'() {
  when:
    def response = RestAssuredMockMvc.get("/stubs/a:b:c:d")
  then:
    response.statusCode == 404
}

def 'should return a list of messaging labels that can be triggered when version
and classifier are passed'() {
  when:
    String response = RestAssuredMockMvc.get('/triggers').body.asString()
  then:
    def root = new JsonSlurper().parseText(response)
    root.'org.springframework.cloud.contract.verifier.stubs:bootService:0.0.1-
SNAPSHOT:stubs'?.containsAll(["delete_book", "return_book_1", "return_book_2"])
}

def 'should trigger a messaging label'() {
  given:
    StubRunning stubRunning = Mock()
    RestAssuredMockMvc.standaloneSetup(new HttpStubsController(stubRunning),
new TriggerController(stubRunning))
  when:
    def response = RestAssuredMockMvc.post("/triggers/delete_book")
  then:
    response.statusCode == 200
  and:
    1 * stubRunning.trigger('delete_book')
}

def 'should trigger a messaging label for a stub with [#stubId] ivy notation'() {
  given:

```

```

        StubRunning stubRunning = Mock()
        RestAssuredMockMvc.standaloneSetup(new HttpStubsController(stubRunning),
new TriggerController(stubRunning))
        when:
            def response = RestAssuredMockMvc.post("/triggers/$stubId/delete_book")
        then:
            response.statusCode == 200
        and:
            1 * stubRunning.trigger(stubId, 'delete_book')
        where:
            stubId <<
['org.springframework.cloud.contract.verifier.stubs:bootService:stubs',
'org.springframework.cloud.contract.verifier.stubs:bootService', 'bootService']
    }

    def 'should throw exception when trigger is missing'() {
        when:
            RestAssuredMockMvc.post("/triggers/missing_label")
        then:
            Exception e = thrown(Exception)
            e.message.contains("Exception occurred while trying to return
[missing_label] label.")
            e.message.contains("Available labels are")

            e.message.contains("org.springframework.cloud.contract.verifier.stubs:loanIssuance:0.0
.1-SNAPSHOT:stubs=[])")

            e.message.contains("org.springframework.cloud.contract.verifier.stubs:bootService:0.0
.1-SNAPSHOT:stubs=")
    }
}
}

```

Stub Runner Boot with Service Discovery

One way to use Stub Runner Boot is to use it as a feed of stubs for “smoke tests”. What does that mean? Assume that you do not want to deploy 50 microservices to a test environment in order to see whether your application works. You have already executed a suite of tests during the build process, but you would also like to ensure that the packaging of your application works. You can deploy your application to an environment, start it, and run a couple of tests on it to see whether it works. We can call those tests “smoke tests”, because their purpose is to check only a handful of testing scenarios.

The problem with this approach is thatm if you use microservices, you most likely also use a service discovery tool. Stub Runner Boot lets you solve this issue by starting the required stubs and registering them in a service discovery tool. Consider the following example of such a setup with Eureka (assume that Eureka is already running):

```

@SpringBootApplication
@EnableStubRunnerServer
@EnableEurekaClient
@AutoConfigureStubRunner
public class StubRunnerBootEurekaExample {

    public static void main(String[] args) {
        SpringApplication.run(StubRunnerBootEurekaExample.class, args);
    }

}

```

We want to start a Stub Runner Boot server (`@EnableStubRunnerServer`), enable the Eureka client (`@EnableEurekaClient`), and have the stub runner feature turned on (`@AutoConfigureStubRunner`).

Now assume that we want to start this application so that the stubs get automatically registered. We can do so by running the application with `java -jar ${SYSTEM_PROPS} stub-runner-boot-eureka-example.jar`, where `${SYSTEM_PROPS}` contains the following list of properties:

```

* -Dstubrunner.repositoryRoot=https://repo.spring.io/snapshot (1)
* -Dstubrunner.cloud.stubbed.discovery.enabled=false (2)
*
-Dstubrunner.ids=org.springframework.cloud.contract.verifier.stubs:loanIssuance,or
g.
*
springframework.cloud.contract.verifier.stubs:fraudDetectionServer,org.springframe
work.
* cloud.contract.verifier.stubs:bootService (3)
* -Dstubrunner.idsToServiceIds.fraudDetectionServer=
* someNameThatShouldMapFraudDetectionServer (4)
*
* (1) - we tell Stub Runner where all the stubs reside (2) - we don't want the
default
* behaviour where the discovery service is stubbed. That's why the stub
registration will
* be picked (3) - we provide a list of stubs to download (4) - we provide a list
of

```

That way, your deployed application can send requests to started WireMock servers through service discovery. Most likely, points 1 through 3 could be set by default in `application.yml`, because they are not likely to change. That way, you can provide only the list of stubs to download whenever you start the Stub Runner Boot.

Consumer-Driven Contracts: Stubs Per Consumer

There are cases in which two consumers of the same endpoint want to have two different responses.



This approach also lets you immediately know which consumer uses which part of your API. You can remove part of a response that your API produces and see which of your autogenerated tests fails. If none fails, you can safely delete that part of the response, because nobody uses it.

Consider the following example of a contract defined for the producer called `producer`, which has two consumers (`foo-consumer` and `bar-consumer`):

Consumer `foo-service`

```
request {
  url '/foo'
  method GET()
}
response {
  status OK()
  body(
    foo: "foo"
  )
}
```

Consumer `bar-service`

```
request {
  url '/bar'
  method GET()
}
response {
  status OK()
  body(
    bar: "bar"
  )
}
```

You cannot produce two different responses for the same request. That is why you can properly package the contracts and then profit from the `stubsPerConsumer` feature.

On the producer side, the consumers can have a folder that contains contracts related only to them. By setting the `stubrunner.stubs-per-consumer` flag to `true`, we no longer register all stubs but only those that correspond to the consumer application's name. In other words, we scan the path of every stub and, if it contains a subfolder with name of the consumer in the path, only then is it registered.

On the `foo` producer side the contracts would look like this

```
.
├── contracts
│   ├── bar-consumer
│   │   ├── bookReturnedForBar.groovy
│   │   └── shouldCallBar.groovy
│   └── foo-consumer
│       ├── bookReturnedForFoo.groovy
│       └── shouldCallFoo.groovy
```

The `bar-consumer` consumer can either set the `spring.application.name` or the `stubrunner.consumer-name` to `bar-consumer`. Alternatively, you can set the test as follows:

```
@ContextConfiguration(classes = Config, loader = SpringBootTestLoader)
@SpringBootTest(properties = ["spring.application.name=bar-consumer"])
@AutoConfigureStubRunner(ids =
    "org.springframework.cloud.contract.verifier.stubs:producerWithMultipleConsumers",
    repositoryRoot = "classpath:m2repo/repository/",
    stubsMode = StubRunnerProperties.StubsMode.REMOTE,
    stubsPerConsumer = true)
@ActiveProfiles("streamconsumer")
class StubRunnerStubsPerConsumerSpec extends Specification {
    ...
}
```

Then only the stubs registered under a path that contains `bar-consumer` in its name (that is, those from the `src/test/resources/contracts/bar-consumer/some/contracts/...` folder) are allowed to be referenced.

You can also set the consumer name explicitly, as follows:

```
@ContextConfiguration(classes = Config, loader = SpringBootTestLoader)
@SpringBootTest
@AutoConfigureStubRunner(ids =
    "org.springframework.cloud.contract.verifier.stubs:producerWithMultipleConsumers",
    repositoryRoot = "classpath:m2repo/repository/",
    consumerName = "foo-consumer",
    stubsMode = StubRunnerProperties.StubsMode.REMOTE,
    stubsPerConsumer = true)
@ActiveProfiles("streamconsumer")
class StubRunnerStubsPerConsumerWithConsumerNameSpec extends Specification {
    ...
}
```

Then only the stubs registered under a path that contains the `foo-consumer` in its name (that is, those from the `src/test/resources/contracts/foo-consumer/some/contracts/...` folder) are allowed to be referenced.

See [issue 224](#) for more information about the reasons behind this change.

Fetching Stubs or Contract Definitions From A Location

Instead of picking the stubs or contract definitions from Artifactory / Nexus or Git, one can just want to point to a location on drive or classpath. This can be especially useful in a multimodule project, where one module wants to reuse stubs or contracts from another module without the need to actually install those in a local maven repository or commit those changes to Git.

In order to achieve this it's enough to use the `stubs://` protocol when the repository root parameter is set either in Stub Runner or in a Spring Cloud Contract plugin.

In this example the `producer` project has been successfully built and stubs were generated under the `target/stubs` folder. As a consumer one can setup the Stub Runner to pick the stubs from that location using the `stubs://` protocol.

Annotation

```
@AutoConfigureStubRunner(  
    stubsMode = StubRunnerProperties.StubsMode.REMOTE,  
    repositoryRoot = "stubs://file://location/to/the/producer/target/stubs/",  
    ids = "com.example:some-producer")
```

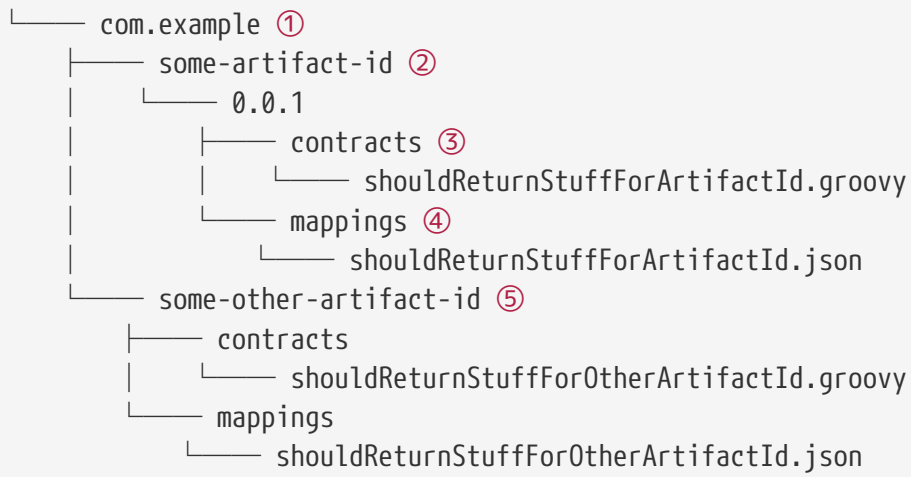
JUnit 4 Rule

```
@Rule  
public StubRunnerRule rule = new StubRunnerRule()  
    .downloadStub("com.example:some-producer")  
    .repoRoot("stubs://file://location/to/the/producer/target/stubs/")  
    .stubsMode(StubRunnerProperties.StubsMode.REMOTE);
```

JUnit 5 Extension

```
@RegisterExtension  
public StubRunnerExtension stubRunnerExtension = new StubRunnerExtension()  
    .downloadStub("com.example:some-producer")  
    .repoRoot("stubs://file://location/to/the/producer/target/stubs/")  
    .stubsMode(StubRunnerProperties.StubsMode.REMOTE);
```

Contracts and stubs may be stored in a location, where each producer has its own, dedicated folder for contracts and stub mappings. Under that folder each consumer can have its own setup. To make Stub Runner find the dedicated folder from the provided ids one can pass a property `stubs.find-producer=true` or a system property `stubrunner.stubs.find-producer=true`.



① group id of the consumers

② consumer with artifact id [some-artifact-id]

③ contracts for the consumer with artifact id [some-artifact-id]

④ mappings for the consumer with artifact id [some-artifact-id]

⑤ consumer with artifact id [some-other-artifact-id]

Annotation

```
@AutoConfigureStubRunner(  
    stubsMode = StubRunnerProperties.StubsMode.REMOTE,  
    repositoryRoot = "stubs://file://location/to/the/contracts/directory",  
    ids = "com.example:some-producer",  
    properties="stubs.find-producer=true")
```

JUnit 4 Rule

```
static Map<String, String> contractProperties() {  
    Map<String, String> map = new HashMap<>();  
    map.put("stubs.find-producer", "true");  
    return map;  
}  
  
@Rule  
public StubRunnerRule rule = new StubRunnerRule()  
    .downloadStub("com.example:some-producer")  
    .repoRoot("stubs://file://location/to/the/contracts/directory")  
    .stubsMode(StubRunnerProperties.StubsMode.REMOTE)  
    .properties(contractProperties());
```

JUnit 5 Extension

```
static Map<String, String> contractProperties() {  
    Map<String, String> map = new HashMap<>();  
    map.put("stubs.find-producer", "true");  
    return map;  
}  
  
@RegisterExtension  
public StubRunnerExtension stubRunnerExtension = new StubRunnerExtension()  
    .downloadStub("com.example:some-producer")  
    .repoRoot("stubs://file://location/to/the/contracts/directory")  
    .stubsMode(StubRunnerProperties.StubsMode.REMOTE)  
    .properties(contractProperties());
```

Generating Stubs at Runtime

As a consumer, you might not want to wait for the producer to finish its implementation and then publish their stubs. A solution to this problem can be generation of stubs at runtime.

As a producer, when a contract is defined, you are required to make the generated tests pass in order for the stubs to be published. There are cases where you would like to unblock the consumers so that they can fetch the stubs before your tests are actually passing. In this case you should set such contracts as in progress. You can read more about this under the [Contracts in Progress](#) section.

That way your tests will not be generated, but the stubs will.

As a consumer, you can toggle a switch to generate stubs at runtime. Stub Runner will ignore all the existing stub mappings and will generate new ones for all the contract definitions. Another option is to pass the `stubrunner.generate-stubs` system property. Below you can find an example of such setup.

Annotation

```
@AutoConfigureStubRunner(  
    stubsMode = StubRunnerProperties.StubsMode.REMOTE,  
    repositoryRoot = "stubs://file://location/to/the/contracts",  
    ids = "com.example:some-producer",  
    generateStubs = true)
```

JUnit 4 Rule

```
@Rule  
public StubRunnerRule rule = new StubRunnerRule()  
    .downloadStub("com.example:some-producer")  
    .repoRoot("stubs://file://location/to/the/contracts")  
    .stubsMode(StubRunnerProperties.StubsMode.REMOTE)  
    .withGenerateStubs(true);
```

JUnit 5 Extension

```
@RegisterExtension  
public StubRunnerExtension stubRunnerExtension = new StubRunnerExtension()  
    .downloadStub("com.example:some-producer")  
    .repoRoot("stubs://file://location/to/the/contracts")  
    .stubsMode(StubRunnerProperties.StubsMode.REMOTE)  
    .withGenerateStubs(true);
```

Fail On No Stubs

By default Stub Runner will fail if no stubs were found. In order to change that behaviour, just set to `false` the `failOnNoStubs` property in the annotation or call the `withFailOnNoStubs(false)` method on a JUnit Rule or Extension.

Annotation

```
@AutoConfigureStubRunner(  
    stubsMode = StubRunnerProperties.StubsMode.REMOTE,  
    repositoryRoot = "stubs://file://location/to/the/contracts",  
    ids = "com.example:some-producer",  
    failOnNoStubs = false)
```

JUnit 4 Rule

```
@Rule  
public StubRunnerRule rule = new StubRunnerRule()  
    .downloadStub("com.example:some-producer")  
    .repoRoot("stubs://file://location/to/the/contracts")  
    .stubsMode(StubRunnerProperties.StubsMode.REMOTE)  
    .withFailOnNoStubs(false);
```

JUnit 5 Extension

```
@RegisterExtension  
public StubRunnerExtension stubRunnerExtension = new StubRunnerExtension()  
    .downloadStub("com.example:some-producer")  
    .repoRoot("stubs://file://location/to/the/contracts")  
    .stubsMode(StubRunnerProperties.StubsMode.REMOTE)  
    .withFailOnNoStubs(false);
```

Common Properties

This section briefly describes common properties, including:

- [Common Properties for JUnit and Spring](#)
- [Stub Runner Stubs IDs](#)

Common Properties for JUnit and Spring

You can set repetitive properties by using system properties or Spring configuration properties. The following table shows their names with their default values:

Property name	Default value	Description
stubrunner.minPort	10000	Minimum value of a port for a started WireMock with stubs.
stubrunner.maxPort	15000	Maximum value of a port for a started WireMock with stubs.
stubrunner.repositoryRoot		Maven repo URL. If blank, then call the local Maven repo.

Property name	Default value	Description
stubrunner.classifier	stubs	Default classifier for the stub artifacts.
stubrunner.stubsMode	CLASSPATH	The way you want to fetch and register the stubs
stubrunner.ids		Array of Ivy notation stubs to download.
stubrunner.username		Optional username to access the tool that stores the JARs with stubs.
stubrunner.password		Optional password to access the tool that stores the JARs with stubs.
stubrunner.stubsPerConsumer	false	Set to true if you want to use different stubs for each consumer instead of registering all stubs for every consumer.
stubrunner.consumerName		If you want to use a stub for each consumer and want to override the consumer name, change this value.

Stub Runner Stubs IDs

You can set the stubs to download in the `stubrunner.ids` system property. They use the following pattern:

```
groupId:artifactId:version:classifier:port
```

Note that `version`, `classifier`, and `port` are optional.

- If you do not provide the `port`, a random one is picked.
- If you do not provide the `classifier`, the default is used. (Note that you can pass an empty classifier this way: `groupId:artifactId:version:`).
- If you do not provide the `version`, then `+` is passed, and the latest one is downloaded.

`port` means the port of the WireMock server.



Starting with version 1.0.4, you can provide a range of versions that you would like the Stub Runner to take into consideration. You can read more about the [Aether versioning ranges here](#).

14.3.6. Spring Cloud Contract WireMock

The Spring Cloud Contract WireMock modules let you use [WireMock](#) in a Spring Boot application. Check out the [samples](#) for more details.

If you have a Spring Boot application that uses Tomcat as an embedded server (which is the default with `spring-boot-starter-web`), you can add `spring-cloud-starter-contract-stub-runner` to your classpath and add `@AutoConfigureWireMock` to use Wiremock in your tests. Wiremock runs as a stub server, and you can register stub behavior by using a Java API or by using static JSON declarations as part of your test. The following code shows an example:

```
@SpringBootTest(webEnvironment = WebEnvironment.RANDOM_PORT)
@AutoConfigureWireMock(port = 0)
public class WiremockForDocsTests {

    // A service that calls out over HTTP
    @Autowired
    private Service service;

    @BeforeEach
    public void setup() {
        this.service.setBase("http://localhost:"
            + this.environment.getProperty("wiremock.server.port"));
    }

    // Using the WireMock APIs in the normal way:
    @Test
    public void contextLoads() throws Exception {
        // Stubbing WireMock
        stubFor(get(urlEqualTo("/resource")).willReturn(aResponse()
            .withHeader("Content-Type", "text/plain").withBody("Hello
World!"))));
        // We're asserting if WireMock responded properly
        assertThat(this.service.go()).isEqualTo("Hello World!");
    }
}
```

To start the stub server on a different port, use (for example), `@AutoConfigureWireMock(port=9999)`. For a random port, use a value of `0`. The stub server port can be bound in the test application context with the "wiremock.server.port" property. Using `@AutoConfigureWireMock` adds a bean of type `WiremockConfiguration` to your test application context, where it is cached between methods and classes having the same context. The same is true for Spring integration tests. Also, you can inject a bean of type `WireMockServer` into your test. The registered WireMock server is reset after each test class, however, if you need to reset it after each test method, just set the `wiremock.reset-mappings-after-each-test` property to `true`.

Registering Stubs Automatically

If you use `@AutoConfigureWireMock`, it registers WireMock JSON stubs from the file system or classpath (by default, from `file:src/test/resources/mappings`). You can customize the locations by using the `stubs` attribute in the annotation, which can be an Ant-style resource pattern or a directory. In the case of a directory, `*/*.json` is appended. The following code shows an example:

```
@RunWith(SpringRunner.class)
@SpringBootTest
@AutoConfigureWireMock(stubs="classpath:/stubs")
public class WiremockImportApplicationTests {

    @Autowired
    private Service service;

    @Test
    public void contextLoads() throws Exception {
        assertThat(this.service.go()).isEqualTo("Hello World!");
    }
}
```



Actually, WireMock always loads mappings from `src/test/resources/mappings` **as well as** the custom locations in the `stubs` attribute. To change this behavior, you can also specify a files root, as described in the next section of this document.

If you use Spring Cloud Contract's default stub jars, your stubs are stored in the `/META-INF/group-id/artifact-id/versions/mappings/` folder. If you want to register all stubs from that location, from all embedded JARs, you can use the following syntax:

```
@AutoConfigureWireMock(port = 0, stubs = "classpath*/META-INF/**/mappings/**/*.*.json")
```

Using Files to Specify the Stub Bodies

WireMock can read response bodies from files on the classpath or the file system. In the case of the file system, you can see in the JSON DSL that the response has a `bodyFileName` instead of a (literal) `body`. The files are resolved relative to a root directory (by default, `src/test/resources/__files`). To customize this location, you can set the `files` attribute in the `@AutoConfigureWireMock` annotation to the location of the parent directory (in other words, `__files` is a subdirectory). You can use Spring resource notation to refer to `file:...` or `classpath:...` locations. Generic URLs are not supported. A list of values can be given—in which case, WireMock resolves the first file that exists when it needs to find a response body.



When you configure the `files` root, it also affects the automatic loading of stubs, because they come from the root location in a subdirectory called `mappings`. The value of `files` has no effect on the stubs loaded explicitly from the `stubs` attribute.

Alternative: Using JUnit Rules

For a more conventional WireMock experience, you can use JUnit `@Rules` to start and stop the server. To do so, use the `WireMockSpring` convenience class to obtain an `Options` instance, as the following example shows:

```

@SpringBootTest(webEnvironment = WebEnvironment.RANDOM_PORT)
public class WiremockForDocsClassRuleTests {

    // Start WireMock on some dynamic port
    // for some reason `dynamicPort()` is not working properly
    public static WireMockServer wiremock = new
WireMockServer(WireMockSpring.options().dynamicPort());

    @BeforeAll
    static void setupClass() {
        wiremock.start();
    }

    @AfterEach
    void after() {
        wiremock.resetAll();
    }

    @AfterAll
    static void clean() {
        wiremock.shutdown();
    }

    // A service that calls out over HTTP to wiremock's port
    @Autowired
    private Service service;

    @BeforeEach
    public void setup() {
        this.service.setBase("http://localhost:" + wiremock.port());
    }

    // Using the WireMock APIs in the normal way:
    @Test
    public void contextLoads() throws Exception {
        // Stubbing WireMock
        wiremock.stubFor(get(urlEqualTo("/resource")).willReturn(aResponse()
            .withHeader("Content-Type", "text/plain").withBody("Hello
World!"))));
        // We're asserting if WireMock responded properly
        assertThat(this.service.go()).isEqualTo("Hello World!");
    }
}

```

The `@ClassRule` means that the server shuts down after all the methods in this class have been run.

Relaxed SSL Validation for Rest Template

WireMock lets you stub a “secure” server with an `https` URL protocol. If your application wants to contact that stub server in an integration test, it will find that the SSL certificates are not valid (the usual problem with self-installed certificates). The best option is often to re-configure the client to use `http`. If that is not an option, you can ask Spring to configure an HTTP client that ignores SSL validation errors (do so only for tests, of course).

To make this work with minimum fuss, you need to use the Spring Boot `RestTemplateBuilder` in your application, as the following example shows:

```
@Bean
public RestTemplate restTemplate(RestTemplateBuilder builder) {
    return builder.build();
}
```

You need `RestTemplateBuilder` because the builder is passed through callbacks to initialize it, so the SSL validation can be set up in the client at that point. This happens automatically in your test if you use the `@AutoConfigureWireMock` annotation or the stub runner. If you use the JUnit `@Rule` approach, you need to add the `@AutoConfigureHttpClient` annotation as well, as the following example shows:

```
@RunWith(SpringRunner.class)
@SpringBootTest("app.baseUrl=https://localhost:6443")
@AutoConfigureHttpClient
public class WiremockHttpsServerApplicationTests {

    @ClassRule
    public static WireMockClassRule wiremock = new WireMockClassRule(
        WireMockSpring.options().httpsPort(6443));
    ...
}
```

If you use `spring-boot-starter-test`, you have the Apache HTTP client on the classpath, and it is selected by the `RestTemplateBuilder` and configured to ignore SSL errors. If you use the default `java.net` client, you do not need the annotation (but it does no harm). There is currently no support for other clients, but it may be added in future releases.

To disable the custom `RestTemplateBuilder`, set the `wiremock.rest-template-ssl-enabled` property to `false`.

WireMock and Spring MVC Mocks

Spring Cloud Contract provides a convenience class that can load JSON WireMock stubs into a Spring `MockRestServiceServer`. The following code shows an example:

```

@SpringBootTest(webEnvironment = WebEnvironment.NONE)
public class WiremockForDocsMockServerApplicationTests {

    @Autowired
    private RestTemplate restTemplate;

    @Autowired
    private Service service;

    @Test
    public void contextLoads() throws Exception {
        // will read stubs classpath
        MockRestServiceServer server =
WireMockRestServiceServer.with(this.restTemplate)

        .baseUrl("https://example.org").stubs("classpath:/stubs/resource.json")
            .build();
        // We're asserting if WireMock responded properly
        assertThat(this.service.go()).isEqualTo("Hello World");
        server.verify();
    }
}

```

The `baseUrl` value is prepended to all mock calls, and the `stubs()` method takes a stub path resource pattern as an argument. In the preceding example, the stub defined at `/stubs/resource.json` is loaded into the mock server. If the `RestTemplate` is asked to visit `example.org/`, it gets the responses as being declared at that URL. More than one stub pattern can be specified, and each one can be a directory (for a recursive list of all `.json`), a fixed filename (as in the preceding example), or an Ant-style pattern. The JSON format is the normal WireMock format, which you can read about at the [WireMock website](#).

Currently, the Spring Cloud Contract Verifier supports Tomcat, Jetty, and Undertow as Spring Boot embedded servers, and Wiremock itself has “native” support for a particular version of Jetty (currently 9.2). To use the native Jetty, you need to add the native Wiremock dependencies and exclude the Spring Boot container (if there is one).

14.3.7. Build Tools Integration

You can run test generation and stub execution in various ways. The most common ones are as follows:

- [Maven](#)
- [Gradle](#)
- [Docker](#)

14.3.8. What to Read Next

If you want to learn more about any of the classes discussed in this section, you can browse the [source code directly](#). If you have specific questions, see the [how-to](#) section.

If you are comfortable with Spring Cloud Contract's core features, you can continue on and read about [Spring Cloud Contract's advanced features](#).

14.4. Maven Project

To learn how to set up the Maven project for Spring Cloud Contract Verifier, read the following sections:

- [Adding the Maven Plugin](#)
- [Maven and Rest Assured 2.0](#)
- [Using Snapshot and Milestone Versions for Maven](#)
- [Adding stubs](#)
- [Run plugin](#)
- [Configure plugin](#)
- [Configuration Options](#)
- [Single Base Class for All Tests](#)
- [Using Different Base Classes for Contracts](#)
- [Invoking Generated Tests](#)
- [Pushing Stubs to SCM](#)
- [Maven Plugin and STS](#)

You can also check the plugin's documentation [here](#).

14.4.1. Adding the Maven Plugin

Add the Spring Cloud Contract BOM in a fashion similar to the following:

```

<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>org.springframework.cloud</groupId>
      <artifactId>spring-cloud-dependencies</artifactId>
      <version>${spring-cloud-release.version}</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
  </dependencies>
</dependencyManagement>

```

Next, add the **Spring Cloud Contract Verifier** Maven plugin, as follows:

```

<plugin>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-contract-maven-plugin</artifactId>
  <version>${spring-cloud-contract.version}</version>
  <extensions>true</extensions>
  <configuration>

<packageWithBaseClasses>com.example.fraud</packageWithBaseClasses>
<!--          <convertToYaml>true</convertToYaml>-->
  </configuration>
  <!-- if additional dependencies are needed e.g. for Pact -->
  <dependencies>
    <dependency>
      <groupId>org.springframework.cloud</groupId>
      <artifactId>spring-cloud-contract-pact</artifactId>
      <version>${spring-cloud-contract.version}</version>
    </dependency>
  </dependencies>
</plugin>

```

You can read more in the [spring-cloud-contract-maven-plugin/index.html](https://spring.io/projects/spring-cloud-contract-maven-plugin) [Spring Cloud Contract Maven Plugin Documentation].

Sometimes, regardless of the picked IDE, you can see that the **target/generated-test-source** folder is not visible on the IDE's classpath. To ensure that it's always there, you can add the following entry to your **pom.xml**


```
<plugin>
  <groupId>org.codehaus.mojo</groupId>
  <artifactId>build-helper-maven-plugin</artifactId>
  <executions>
    <execution>
      <id>add-source</id>
      <phase>generate-test-sources</phase>
      <goals>
        <goal>add-test-source</goal>
      </goals>
      <configuration>
        <sources>
          <source>${project.build.directory}/generated-test-
sources/contracts</source>
        </sources>
      </configuration>
    </execution>
  </executions>
</plugin>
```

14.4.2. Maven and Rest Assured 2.0

By default, Rest Assured 3.x is added to the classpath. However, you can use Rest Assured 2.x by adding it to the plugins classpath, as follows:

```

<plugin>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-contract-maven-plugin</artifactId>
  <version>${spring-cloud-contract.version}</version>
  <extensions>>true</extensions>
  <configuration>
    <packageWithBaseClasses>com.example</packageWithBaseClasses>
  </configuration>
  <dependencies>
    <dependency>
      <groupId>org.springframework.cloud</groupId>
      <artifactId>spring-cloud-contract-verifier</artifactId>
      <version>${spring-cloud-contract.version}</version>
    </dependency>
    <dependency>
      <groupId>com.jayway.restassured</groupId>
      <artifactId>rest-assured</artifactId>
      <version>2.5.0</version>
      <scope>compile</scope>
    </dependency>
    <dependency>
      <groupId>com.jayway.restassured</groupId>
      <artifactId>spring-mock-mvc</artifactId>
      <version>2.5.0</version>
      <scope>compile</scope>
    </dependency>
  </dependencies>
</plugin>

<dependencies>
  <!-- all dependencies -->
  <!-- you can exclude rest-assured from spring-cloud-contract-verifier -->
  <dependency>
    <groupId>com.jayway.restassured</groupId>
    <artifactId>rest-assured</artifactId>
    <version>2.5.0</version>
    <scope>test</scope>
  </dependency>
  <dependency>
    <groupId>com.jayway.restassured</groupId>
    <artifactId>spring-mock-mvc</artifactId>
    <version>2.5.0</version>
    <scope>test</scope>
  </dependency>
</dependencies>

```

That way, the plugin automatically sees that Rest Assured 2.x is present on the classpath and modifies the imports accordingly.

14.4.3. Using Snapshot and Milestone Versions for Maven

To use Snapshot and Milestone versions, you have to add the following section to your `pom.xml`:

```
<repositories>
  <repository>
    <id>spring-snapshots</id>
    <name>Spring Snapshots</name>
    <url>https://repo.spring.io/snapshot</url>
    <snapshots>
      <enabled>true</enabled>
    </snapshots>
  </repository>
  <repository>
    <id>spring-milestones</id>
    <name>Spring Milestones</name>
    <url>https://repo.spring.io/milestone</url>
    <snapshots>
      <enabled>false</enabled>
    </snapshots>
  </repository>
  <repository>
    <id>spring-releases</id>
    <name>Spring Releases</name>
    <url>https://repo.spring.io/release</url>
    <snapshots>
      <enabled>false</enabled>
    </snapshots>
  </repository>
</repositories>
<pluginRepositories>
  <pluginRepository>
    <id>spring-snapshots</id>
    <name>Spring Snapshots</name>
    <url>https://repo.spring.io/snapshot</url>
    <snapshots>
      <enabled>true</enabled>
    </snapshots>
  </pluginRepository>
  <pluginRepository>
    <id>spring-milestones</id>
    <name>Spring Milestones</name>
    <url>https://repo.spring.io/milestone</url>
    <snapshots>
      <enabled>false</enabled>
    </snapshots>
  </pluginRepository>
  <pluginRepository>
    <id>spring-releases</id>
    <name>Spring Releases</name>
```

```
<url>https://repo.spring.io/release</url>
<snapshots>
  <enabled>>false</enabled>
</snapshots>
</pluginRepository>
</pluginRepositories>
```

14.4.4. Adding stubs

By default, Spring Cloud Contract Verifier looks for stubs in the `src/test/resources/contracts` directory. The directory containing stub definitions is treated as a class name, and each stub definition is treated as a single test. We assume that it contains at least one directory to be used as the test class name. If there is more than one level of nested directories, all except the last one is used as the package name. Consider the following structure:

```
src/test/resources/contracts/myservice/shouldCreateUser.groovy
src/test/resources/contracts/myservice/shouldReturnUser.groovy
```

Given that structure, Spring Cloud Contract Verifier creates a test class named `defaultBasePackage.MyService` with two methods:

- `shouldCreateUser()`
- `shouldReturnUser()`

14.4.5. Run plugin

The `generateTests` plugin goal is assigned to be invoked in the phase called `generate-test-sources`. If you want it to be part of your build process, you need not do anything. If you want only to generate tests, invoke the `generateTests` goal.

If you want to run stubs via Maven it's enough to call the `run` goal with the stubs to run as the `spring.cloud.contract.verifier.stubs` system property as follows:

```
mvn          org.springframework.cloud:spring-cloud-contract-maven-plugin:run \
-Dspring.cloud.contract.verifier.stubs="com.acme:service-name"
```

14.4.6. Configure plugin

To change the default configuration, you can add a `configuration` section to the plugin definition or the `execution` definition, as follows:

```

<plugin>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-contract-maven-plugin</artifactId>
  <executions>
    <execution>
      <goals>
        <goal>convert</goal>
        <goal>generateStubs</goal>
        <goal>generateTests</goal>
      </goals>
    </execution>
  </executions>
  <configuration>

  <basePackageForTests>org.springframework.cloud.verifier.twitter.place</basePackage
  ForTests>

  <baseClassForTests>org.springframework.cloud.verifier.twitter.place.BaseMockMvcSpe
  c</baseClassForTests>
  </configuration>
</plugin>

```

14.4.7. Configuration Options

- **testMode**: Defines the mode for acceptance tests. By default, the mode is **MockMvc**, which is based on Spring's MockMvc. You can also change it to **WebTestClient**, **JaxRsClient**, or **Explicit** (for real HTTP calls).
- **basePackageForTests**: Specifies the base package for all generated tests. If not set, the value is picked from the package of **baseClassForTests** and from **packageWithBaseClasses**. If neither of these values are set, the value is set to **org.springframework.cloud.contract.verifier.tests**.
- **ruleClassForTests**: Specifies a rule that should be added to the generated test classes.
- **baseClassForTests**: Creates a base class for all generated tests. By default, if you use Spock classes, the class is **spock.lang.Specification**.
- **contractsDirectory**: Specifies a directory that contains contracts written with the Groovyn DSL. The default directory is **/src/test/resources/contracts**.
- **generatedTestSourcesDir**: Specifies the test source directory where tests generated from the Groovy DSL should be placed. By default, its value is **\$buildDir/generated-test-sources/contracts**.
- **generatedTestResourcesDir**: Specifies the test resource directory for resources used by the generated tests.
- **testFramework**: Specifies the target test framework to be used. Currently, Spock, JUnit 4 (**TestFramework.JUNIT**), and JUnit 5 are supported, with JUnit 4 being the default framework.
- **packageWithBaseClasses**: Defines a package where all the base classes reside. This setting takes

precedence over `baseClassForTests`. The convention is such that, if you have a contract under (for example) `src/test/resources/contract/foo/bar/baz/` and set the value of the `packageWithBaseClasses` property to `com.example.base`, Spring Cloud Contract Verifier assumes that there is a `BarBazBase` class under the `com.example.base` package. In other words, the system takes the last two parts of the package, if they exist, and forms a class with `Base` as a suffix.

- `baseClassMappings`: Specifies a list of base class mappings that provide `contractPackageRegex` (which is checked against the package where the contract is located) and `baseClassFQN` (which maps to the fully qualified name of the base class for the matched contract). For example, if you have a contract under `src/test/resources/contract/foo/bar/baz/` and map the `.* → com.example.base.BaseClass` property, the test class generated from these contracts extends `com.example.base.BaseClass`. This setting takes precedence over `packageWithBaseClasses` and `baseClassForTests`.
- `contractsProperties`: A map that contains properties to be passed to Spring Cloud Contract components. Those properties might be used by (for example) built-in or custom Stub Downloaders.
- `failOnNoContracts`: When enabled, will throw an exception when no contracts were found. Defaults to `true`.
- `failOnInProgress`: If set to `true` then if any contracts that are in progress are found, will break the build. On the producer side you need to be explicit about the fact that you have contracts in progress and take into consideration that you might be causing false positive test execution results on the consumer side.. Defaults to `true`.
- `incrementalContractTests`: When enabled, tests are created only when contracts have changed since last build. Defaults to `true`.
- `incrementalContractStubs`: When enabled, stubs are created only when contracts have changed since last build. Defaults to `true`.
- `incrementalContractStubsJar`: When enabled, stubs jar is created only when stubs have changed since last build. Defaults to `true`.

If you want to download your contract definitions from a Maven repository, you can use the following options:

- `contractDependency`: The contract dependency that contains all the packaged contracts.
- `contractsPath`: The path to the concrete contracts in the JAR with packaged contracts. Defaults to `groupid/artifactid` where `groupid` is slash separated.
- `contractsMode`: Picks the mode in which stubs are found and registered.
- `deleteStubsAfterTest`: If set to `false` will not remove any downloaded contracts from temporary directories.
- `contractsRepositoryUrl`: URL to a repository with the artifacts that have contracts. If it is not provided, use the current Maven ones.
- `contractsRepositoryUsername`: The user name to be used to connect to the repo with contracts.
- `contractsRepositoryPassword`: The password to be used to connect to the repo with contracts.
- `contractsRepositoryProxyHost`: The proxy host to be used to connect to the repo with contracts.

- `contractsRepositoryProxyPort`: The proxy port to be used to connect to the repo with contracts.

We cache only non-snapshot, explicitly provided versions (for example `+` or `1.0.0.BUILD-SNAPSHOT` do not get cached). By default, this feature is turned on.

The following list describes experimental features that you can turn on in the plugin:

- `convertToYaml`: Converts all DSLs to the declarative YAML format. This can be extremely useful when you use external libraries in your Groovy DSLs. By turning this feature on (by setting it to `true`) you need not add the library dependency on the consumer side.
- `assertJsonSize`: You can check the size of JSON arrays in the generated tests. This feature is disabled by default.

14.4.8. Single Base Class for All Tests

When using Spring Cloud Contract Verifier in the default (`MockMvc`), you need to create a base specification for all generated acceptance tests. In this class, you need to point to an endpoint, which should be verified. The following example shows how to do so:

```
package org.mycompany.tests

import org.mycompany.ExampleSpringController
import com.jayway.restassured.module.mockmvc.RestAssuredMockMvc
import spock.lang.Specification

class MvcSpec extends Specification {
    def setup() {
        RestAssuredMockMvc.standaloneSetup(new ExampleSpringController())
    }
}
```

You can also setup the whole context if necessary, as the following example shows:

```

import io.restassured.module.mockmvc.RestAssuredMockMvc;
import org.junit.Before;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.test.context.junit4.SpringRunner;
import org.springframework.web.context.WebApplicationContext;

@RunWith(SpringRunner.class)
@SpringBootTest(webEnvironment = WebEnvironment.RANDOM_PORT, classes =
SomeConfig.class, properties="some=property")
public abstract class BaseTestClass {

    @Autowired
    WebApplicationContext context;

    @Before
    public void setup() {
        RestAssuredMockMvc.webAppContextSetup(this.context);
    }
}

```

If you use **EXPLICIT** mode, you can use a base class to initialize the whole tested app, similar to what you might do in regular integration tests. The following example shows how to do so:


```

import io.restassured.RestAssured;
import org.junit.Before;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.boot.web.server.LocalServerPort
import org.springframework.test.context.junit4.SpringRunner;
import org.springframework.web.context.WebApplicationContext;

@RunWith(SpringRunner.class)
@SpringBootTest(webEnvironment = WebEnvironment.RANDOM_PORT, classes =
SomeConfig.class, properties="some=property")
public abstract class BaseTestClass {

    @LocalServerPort
    int port;

    @Before
    public void setup() {
        RestAssured.baseURI = "http://localhost:" + this.port;
    }
}

```

If you use the **JAXRSCLIENT** mode, this base class should also contain a **protected WebTarget webTarget** field. Right now, the only way to test the JAX-RS API is to start a web server.

14.4.9. Using Different Base Classes for Contracts

If your base classes differ between contracts, you can tell the Spring Cloud Contract plugin which class should get extended by the autogenerated tests. You have two options:

- Follow a convention by providing a value for **packageWithBaseClasses**
- Provide explicit mapping with **baseClassMappings**

By Convention

The convention is such that if you have a contract under (for example) **src/test/resources/contract/foo/bar/baz/** and set the value of the **packageWithBaseClasses** property to **com.example.base**, then Spring Cloud Contract Verifier assumes that there is a **BarBazBase** class under the **com.example.base** package. In other words, the system takes the last two parts of the package, if they exist, and forms a class with a **Base** suffix. This rule takes precedence over **baseClassForTests**. The following example shows how it works in the **contracts** closure:

```

<plugin>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-contract-maven-plugin</artifactId>
  <configuration>
    <packageWithBaseClasses>hello</packageWithBaseClasses>
  </configuration>
</plugin>

```

By Mapping

You can manually map a regular expression of the contract's package to the fully qualified name of the base class for the matched contract. You have to provide a list called `baseClassMappings` that consists of `baseClassMapping` objects that each take a `contractPackageRegex` to `baseClassFQN` mapping. Consider the following example:

```

<plugin>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-contract-maven-plugin</artifactId>
  <configuration>
    <baseClassForTests>com.example.FooBase</baseClassForTests>
    <baseClassMappings>
      <baseClassMapping>
        <contractPackageRegex>.*com.*</contractPackageRegex>
        <baseClassFQN>com.example.TestBase</baseClassFQN>
      </baseClassMapping>
    </baseClassMappings>
  </configuration>
</plugin>

```

Assume that you have contracts under these two locations: `* src/test/resources/contract/com/ *` `src/test/resources/contract/foo/`

By providing the `baseClassForTests`, we have a fallback in case mapping did not succeed. (You can also provide the `packageWithBaseClasses` as a fallback.) That way, the tests generated from `src/test/resources/contract/com/` contracts extend the `com.example.ComBase`, whereas the rest of the tests extend `com.example.FooBase`.

14.4.10. Invoking Generated Tests

The Spring Cloud Contract Maven Plugin generates verification code in a directory called `/generated-test-sources/contractVerifier` and attaches this directory to `testCompile` goal.

For Groovy Spock code, you can use the following:

```

<plugin>
  <groupId>org.codehaus.gmavenplus</groupId>
  <artifactId>gmavenplus-plugin</artifactId>
  <version>1.5</version>
  <executions>
    <execution>
      <goals>
        <goal>testCompile</goal>
      </goals>
    </execution>
  </executions>
  <configuration>
    <testSources>
      <testSource>
        <directory>${project.basedir}/src/test/groovy</directory>
        <includes>
          <include>**/*.groovy</include>
        </includes>
      </testSource>
      <testSource>
        <directory>${project.build.directory}/generated-test-
sources/contractVerifier</directory>
        <includes>
          <include>**/*.groovy</include>
        </includes>
      </testSource>
    </testSources>
  </configuration>
</plugin>

```

To ensure that the provider side is compliant with defined contracts, you need to invoke `mvn generateTest test`.

14.4.11. Pushing Stubs to SCM

If you use the SCM (Source Control Management) repository to keep the contracts and stubs, you might want to automate the step of pushing stubs to the repository. To do that, you can add the `pushStubsToScm` goal. The following example shows how to do so:

```

<plugin>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-contract-maven-plugin</artifactId>
  <version>${spring-cloud-contract.version}</version>
  <extensions>>true</extensions>
  <configuration>
    <!-- Base class mappings etc. -->

    <!-- We want to pick contracts from a Git repository -->
    <contractsRepositoryUrl>git://https://github.com/spring-cloud-
samples/spring-cloud-contract-nodejs-contracts-git.git</contractsRepositoryUrl>

    <!-- We reuse the contract dependency section to set up the path
to the folder that contains the contract definitions. In our case the
path will be /groupId/artifactId/version/contracts -->
    <contractDependency>
      <groupId>${project.groupId}</groupId>
      <artifactId>${project.artifactId}</artifactId>
      <version>${project.version}</version>
    </contractDependency>

    <!-- The contracts mode can't be classpath -->
    <contractsMode>REMOTE</contractsMode>
  </configuration>
  <executions>
    <execution>
      <phase>package</phase>
      <goals>
        <!-- By default we will not push the stubs back to SCM,
you have to explicitly add it as a goal -->
        <goal>pushStubsToScm</goal>
      </goals>
    </execution>
  </executions>
</plugin>

```

Under [Using the SCM Stub Downloader](#), you can find all possible configuration options that you can pass through the `<configuration><contractProperties>` map, a system property, or an environment variable.

14.4.12. Maven Plugin and STS

The following image shows an exception that you may see when you use STS:

[STS Exception] | <https://raw.githubusercontent.com/spring-cloud/spring-cloud->

When you click on the error marker you should see something like the following:

```
plugin:1.1.0.M1:convert:default-convert:process-test-resources)
org.apache.maven.plugin.PluginExecutionException: Execution default-convert of
goal org.springframework.cloud:spring-
cloud-contract-maven-plugin:1.1.0.M1:convert failed. at
org.apache.maven.plugin.DefaultBuildPluginManager.executeMojo(DefaultBuildPluginMa
nager.java:145) at
org.eclipse.m2e.core.internal.embedder.MavenImpl.execute(MavenImpl.java:331) at
org.eclipse.m2e.core.internal.embedder.MavenImpl$11.call(MavenImpl.java:1362) at
...
org.eclipse.core.internal.jobs.Worker.run(Worker.java:55) Caused by:
java.lang.NullPointerException at
org.eclipse.m2e.core.internal.builder.plexusbuildapi.EclipseIncrementalBuildConte
xt.hasDelta(EclipseIncrementalBuildContext.java:53) at
org.sonatype.plexus.build.incremental.ThreadBuildContext.hasDelta(ThreadBuildCont
ext.java:59) at
```

In order to fix this issue, provide the following section in your `pom.xml`:

```

<build>
  <pluginManagement>
    <plugins>
      <!--This plugin's configuration is used to store Eclipse m2e settings
        only. It has no influence on the Maven build itself. -->
      <plugin>
        <groupId>org.eclipse.m2e</groupId>
        <artifactId>lifecycle-mapping</artifactId>
        <version>1.0.0</version>
        <configuration>
          <lifecycleMappingMetadata>
            <pluginExecutions>
              <pluginExecution>
                <pluginExecutionFilter>
                  <groupId>org.springframework.cloud</groupId>
                  <artifactId>spring-cloud-contract-maven-
plugin</artifactId>
                  <versionRange>[1.0,)</versionRange>
                  <goals>
                    <goal>convert</goal>
                  </goals>
                </pluginExecutionFilter>
                <action>
                  <execute />
                </action>
              </pluginExecution>
            </pluginExecutions>
          </lifecycleMappingMetadata>
        </configuration>
      </plugin>
    </plugins>
  </pluginManagement>
</build>

```

14.4.13. Maven Plugin with Spock Tests

You can select the [Spock Framework](#) for creating and running the auto-generated contract verification tests with both Maven and Gradle. However, whereas using Gradle is straightforward, in Maven, you will require some additional setup in order to make the tests compile and execute properly.

First of all, you must use a plugin, such as the [GMavenPlus](#) plugin, to add Groovy to your project. In GMavenPlus plugin, you need to explicitly set test sources, including both the path where your base test classes are defined and the path where the generated contract tests are added. The following example shows how to do so:

```

<plugin>
  <groupId>org.codehaus.gmavenplus</groupId>
  <artifactId>gmavenplus-plugin</artifactId>
  <version>1.6.1</version>
  <executions>
    <execution>
      <goals>
        <goal>compileTests</goal>
        <goal>addTestSources</goal>
      </goals>
    </execution>
  </executions>
  <configuration>
    <testSources>
      <testSource>
        <directory>${project.basedir}/src/test/groovy</directory>
        <includes>
          <include>**/*.groovy</include>
        </includes>
      </testSource>
      <testSource>
        <directory>
          ${project.basedir}/target/generated-test-
sources/contracts/com/example/beer
        </directory>
        <includes>
          <include>**/*.groovy</include>
          <include>**/*.gvy</include>
        </includes>
      </testSource>
    </testSources>
  </configuration>
  <dependencies>
    <dependency>
      <groupId>org.codehaus.groovy</groupId>
      <artifactId>groovy-all</artifactId>
      <version>${groovy.version}</version>
      <scope>runtime</scope>
      <type>pom</type>
    </dependency>
  </dependencies>

```

If you uphold the Spock convention of ending the test class names with **Spec**, you also need to adjust your Maven Surefire plugin setup, as the following example shows:

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-surefire-plugin</artifactId>
  <configuration>
    <includes>
      <include>**/*Test.java</include>
      <include>**/*Spec.java</include>
    </includes>
    <failIfNoTests>true</failIfNoTests>
  </configuration>
</plugin>
```

14.5. Gradle Project

To learn how to set up the Gradle project for Spring Cloud Contract Verifier, read the following sections:

- [Prerequisites](#)
- [Add Gradle Plugin with Dependencies](#)
- [Gradle and Rest Assured 2.0](#)
- [Snapshot Versions for Gradle](#)
- [Add stubs](#)
- [Default Setup](#)
- [Configuring the Plugin](#)
- [Configuration Options](#)
- [Single Base Class for All Tests](#)
- [Different Base Classes for Contracts](#)
- [Invoking Generated Tests](#)
- [Pushing Stubs to SCM](#)
- [Spring Cloud Contract Verifier on the Consumer Side](#)

14.5.1. Prerequisites

In order to use Spring Cloud Contract Verifier with WireMock, you must use either a Gradle or a Maven plugin.



If you want to use Spock in your projects, you must separately add the `spock-core` and `spock-spring` modules. See [Spock's documentation for more information](#)

14.5.2. Add Gradle Plugin with Dependencies

To add a Gradle plugin with dependencies, you can use code similar to the following:

Plugin DSL GA versions

```
// build.gradle
plugins {
    id "groovy"
    // this will work only for GA versions of Spring Cloud Contract
    id "org.springframework.cloud.contract" version "${GAVerifierVersion}"
}

dependencyManagement {
    imports {
        mavenBom "org.springframework.cloud:spring-cloud-contract-
dependencies:${GAVerifierVersion}"
    }
}

dependencies {
    testCompile "org.codehaus.groovy:groovy-all:${groovyVersion}"
    // example with adding Spock core and Spock Spring
    testCompile "org.spockframework:spock-core:${spockVersion}"
    testCompile "org.spockframework:spock-spring:${spockVersion}"
    testCompile 'org.springframework.cloud:spring-cloud-starter-contract-verifier'
}
```

Plugin DSL non GA versions

```
// settings.gradle
pluginManagement {
    plugins {
        id "org.springframework.cloud.contract" version "${verifierVersion}"
    }
    repositories {
        // to pick from local .m2
        mavenLocal()
        // for snapshots
        maven { url "https://repo.spring.io/snapshot" }
        // for milestones
        maven { url "https://repo.spring.io/milestone" }
        // for GA versions
        gradlePluginPortal()
    }
}

// build.gradle
plugins {
    id "groovy"
    id "org.springframework.cloud.contract"
}

dependencyManagement {
    imports {
        mavenBom "org.springframework.cloud:spring-cloud-contract-
dependencies:${verifierVersion}"
    }
}

dependencies {
    testCompile "org.codehaus.groovy:groovy-all:${groovyVersion}"
    // example with adding Spock core and Spock Spring
    testCompile "org.spockframework:spock-core:${spockVersion}"
    testCompile "org.spockframework:spock-spring:${spockVersion}"
    testCompile 'org.springframework.cloud:spring-cloud-starter-contract-verifier'
}
```

```
// build.gradle
buildscript {
    repositories {
        mavenCentral()
    }
    dependencies {
        classpath "org.springframework.boot:spring-boot-gradle-
plugin:${springboot_version}"
        classpath "org.springframework.cloud:spring-cloud-contract-gradle-
plugin:${verifier_version}"
        // here you can also pass additional dependencies such as Pact or Kotlin
spec e.g.:
        // classpath "org.springframework.cloud:spring-cloud-contract-spec-
kotlin:${verifier_version}"
    }
}

apply plugin: 'groovy'
apply plugin: 'spring-cloud-contract'

dependencyManagement {
    imports {
        mavenBom "org.springframework.cloud:spring-cloud-contract-
dependencies:${verifier_version}"
    }
}

dependencies {
    testCompile "org.codehaus.groovy:groovy-all:${groovyVersion}"
    // example with adding Spock core and Spock Spring
    testCompile "org.spockframework:spock-core:${spockVersion}"
    testCompile "org.spockframework:spock-spring:${spockVersion}"
    testCompile 'org.springframework.cloud:spring-cloud-starter-contract-verifier'
}
```

14.5.3. Gradle and Rest Assured 2.0

By default, Rest Assured 3.x is added to the classpath. However, to use Rest Assured 2.x you can add it to the plugins classpath, as the following listing shows:

```

buildscript {
    repositories {
        mavenCentral()
    }
    dependencies {
        classpath "org.springframework.boot:spring-boot-gradle-
plugin:${springboot_version}"
        classpath "org.springframework.cloud:spring-cloud-contract-gradle-
plugin:${verifier_version}"
        classpath "com.jayway.restassured:rest-assured:2.5.0"
        classpath "com.jayway.restassured:spring-mock-mvc:2.5.0"
    }
}

dependencies {
    // all dependencies
    // you can exclude rest-assured from spring-cloud-contract-verifier
    testCompile "com.jayway.restassured:rest-assured:2.5.0"
    testCompile "com.jayway.restassured:spring-mock-mvc:2.5.0"
}

```

That way, the plugin automatically sees that Rest Assured 2.x is present on the classpath and modifies the imports accordingly.

14.5.4. Snapshot Versions for Gradle

You can add the additional snapshot repository to your `build.gradle` to use snapshot versions, which are automatically uploaded after every successful build, as the following listing shows:

```

/*
We need to use the [buildscript {}] section when we have to modify
the classpath for the plugins. If that's not the case this section
can be skipped.

If you don't need to modify the classpath (e.g. add a Pact dependency),
then you can just set the [pluginManagement {}] section in [settings.gradle]
file.

// settings.gradle
pluginManagement {
    repositories {
        // for snapshots
        maven {url "https://repo.spring.io/snapshot"}
        // for milestones
        maven {url "https://repo.spring.io/milestone"}
        // for GA versions
        gradlePluginPortal()
    }
}

*/
buildscript {
    repositories {
        mavenCentral()
        mavenLocal()
        maven { url "https://repo.spring.io/snapshot" }
        maven { url "https://repo.spring.io/milestone" }
        maven { url "https://repo.spring.io/release" }
    }
}

```

14.5.5. Add stubs

By default, Spring Cloud Contract Verifier looks for stubs in the `src/test/resources/contracts` directory.

The directory that contains stub definitions is treated as a class name, and each stub definition is treated as a single test. Spring Cloud Contract Verifier assumes that it contains at least one level of directories that are to be used as the test class name. If more than one level of nested directories is present, all except the last one is used as the package name. Consider the following structure:

```

src/test/resources/contracts/myservice/shouldCreateUser.groovy
src/test/resources/contracts/myservice/shouldReturnUser.groovy

```

Given the preceding structure, Spring Cloud Contract Verifier creates a test class named `defaultBasePackage.MyService` with two methods:

- `shouldCreateUser()`
- `shouldReturnUser()`

14.5.6. Running the Plugin

The plugin registers itself to be invoked before a `check` task. If you want it to be part of your build process, you need do nothing more. If you just want to generate tests, invoke the `generateContractTests` task.

14.5.7. Default Setup

The default Gradle Plugin setup creates the following Gradle part of the build (in pseudocode):

```
contracts {
    testFramework = 'JUNIT'
    testMode = 'MockMvc'
    generatedTestSourcesDir = project.file("${project.buildDir}/generated-test-sources/contracts")
    generatedTestResourcesDir = project.file("${project.buildDir}/generated-test-resources/contracts")
    contractsDslDir =
project.file("${project.rootDir}/src/test/resources/contracts")
    basePackageForTests = 'org.springframework.cloud.verifier.tests'
    stubsOutputDir = project.file("${project.buildDir}/stubs")
    sourceSet = null

    // the following properties are used when you want to provide where the JAR
with contract lays
    contractDependency {
        stringNotation = ''
    }
    contractsPath = ''
    contractsWorkOffline = false
    contractRepository {
        cacheDownloadedContracts(true)
    }
}

tasks.create(type: Jar, name: 'verifierStubsJar', dependsOn:
'generateClientStubs') {
    baseName = project.name
    classifier = contracts.stubsSuffix
    from contractVerifier.stubsOutputDir
}

project.artifacts {
```

```

    archives task
  }

  tasks.create(type: Copy, name: 'copyContracts') {
    from contracts.contractsDsldir
    into contracts.stubsOutputDir
  }

  verifierStubsJar.dependsOn 'copyContracts'

  publishing {
    publications {
      stubs(MavenPublication) {
        artifactId project.name
        artifact verifierStubsJar
      }
    }
  }
}

```

14.5.8. Configuring the Plugin

To change the default configuration, you can add a `contracts` snippet to your Gradle configuration, as the following listing shows:

```

contracts {
  testMode = 'MockMvc'
  baseClassForTests = 'org.mycompany.tests'
  generatedTestSourcesDir = project.file('src/generatedContract')
}

```

14.5.9. Configuration Options

- `testMode`: Defines the mode for acceptance tests. By default, the mode is `MockMvc`, which is based on Spring's `MockMvc`. It can also be changed to `WebTestClient`, `JaxRsClient`, or `Explicit` (for real HTTP calls).
- `imports`: Creates an array with imports that should be included in the generated tests (for example, `['org.myorg.Matchers']`). By default, it creates an empty array.
- `staticImports`: Creates an array with static imports that should be included in generated tests (for example, `['org.myorg.Matchers.*']`). By default, it creates an empty array.
- `basePackageForTests`: Specifies the base package for all generated tests. If not set, the value is picked from the package of `baseClassForTests` and from `packageWithBaseClasses`. If neither of these values are set, the value is set to `org.springframework.cloud.contract.verifier.tests`.
- `baseClassForTests`: Creates a base class for all generated tests. By default, if you use Spock classes, the class is `spock.lang.Specification`.

- `packageWithBaseClasses`: Defines a package where all the base classes reside. This setting takes precedence over `baseClassForTests`.
- `baseClassMappings`: Explicitly maps a contract package to a FQN of a base class. This setting takes precedence over `packageWithBaseClasses` and `baseClassForTests`.
- `ruleClassForTests`: Specifies a rule that should be added to the generated test classes.
- `ignoredFiles`: Uses an `Antmatcher` to allow defining stub files for which processing should be skipped. By default, it is an empty array.
- `contractsDsldir`: Specifies the directory that contains contracts written by using the GroovyDSL. By default, its value is `$rootDir/src/test/resources/contracts`.
- `generatedTestSourcesDir`: Specifies the test source directory where tests generated from the Groovy DSL should be placed. By default, its value is `$buildDir/generated-test-sources/contracts`.
- `generatedTestResourcesDir`: Specifies the test resource directory where resources used by the tests generated from the Groovy DSL should be placed. By default, its value is `$buildDir/generated-test-resources/contracts`.
- `stubsOutputDir`: Specifies the directory where the generated WireMock stubs from the Groovy DSL should be placed.
- `testFramework`: Specifies the target test framework to be used. Currently, Spock, JUnit 4 (`TestFramework.JUNIT`), and JUnit 5 are supported, with JUnit 4 being the default framework.
- `contractsProperties`: A map that contains properties to be passed to Spring Cloud Contract components. Those properties might be used by (for example) built-in or custom Stub Downloaders.
- `sourceSet`: Source set where the contracts are stored. If not provided will assume `test` (e.g. `project.sourceSets.test.java` for JUnit or `project.sourceSets.test.groovy` for Spock).

You can use the following properties when you want to specify the location of the JAR that contains the contracts:

- `contractDependency`: Specifies the Dependency that provides `groupid:artifactid:version:classifier` coordinates. You can use the `contractDependency` closure to set it up.
- `contractsPath`: Specifies the path to the jar. If contract dependencies are downloaded, the path defaults to `groupid/artifactid` where `groupid` is slash separated. Otherwise, it scans contracts under the provided directory.
- `contractsMode`: Specifies the mode for downloading contracts (whether the JAR is available offline, remotely, and so on).
- `deleteStubsAfterTest`: If set to `false`, do not remove any downloaded contracts from temporary directories.
- `failOnNoContracts`: When enabled, will throw an exception when no contracts were found. Defaults to `true`.
- `failOnInProgress`: If set to `true` then if any contracts that are in progress are found, will break the build. On the producer side you need to be explicit about the fact that you have contracts in

progress and take into consideration that you might be causing false positive test execution results on the consumer side.. Defaults to `true`.

There is also the `contractRepository { ... }` closure that contains the following properties

- `repositoryUrl`: the URL to the repository with contract definitions
- `username` : Repository username
- `password` : Repository password
- `proxyPort` : the port of the proxy
- `proxyHost` : the host of the proxy
- `cacheDownloadedContracts` : If set to `true` then will cache the folder where non snapshot contract artifacts got downloaded. Defaults to `true`.

You can also turn on the following experimental features in the plugin:

- `convertToYaml`: Converts all DSLs to the declarative YAML format. This can be extremely useful when you use external libraries in your Groovy DSLs. By turning this feature on (by setting it to `true`) you need not add the library dependency on the consumer side.
- `assertJsonSize`: You can check the size of JSON arrays in the generated tests. This feature is disabled by default.

14.5.10. Single Base Class for All Tests

When using Spring Cloud Contract Verifier in default MockMvc, you need to create a base specification for all generated acceptance tests. In this class, you need to point to an endpoint, which should be verified. The following example shows how to do so:

```
abstract class BaseMockMvcSpec extends Specification {

    def setup() {
        RestAssuredMockMvc.standaloneSetup(new PairIdController())
    }

    void isProperCorrelationId(Integer correlationId) {
        assert correlationId == 123456
    }

    void isEmpty(String value) {
        assert value == null
    }

}
```

If you use `Explicit` mode, you can use a base class to initialize the whole tested application, as you might see in regular integration tests. If you use the `JAXRSCLIENT` mode, this base class should also

contain a `protected WebTarget webTarget` field. Right now, the only option to test the JAX-RS API is to start a web server.

14.5.11. Different Base Classes for Contracts

If your base classes differ between contracts, you can tell the Spring Cloud Contract plugin which class should get extended by the autogenerated tests. You have two options:

- Follow a convention by providing the `packageWithBaseClasses`
- Provide explicit mapping by using `baseClassMappings`

By Convention

The convention is such that if you have a contract in (for example) `src/test/resources/contract/foo/bar/baz/` and set the value of the `packageWithBaseClasses` property to `com.example.base`, then Spring Cloud Contract Verifier assumes that there is a `BarBazBase` class under the `com.example.base` package. In other words, the system takes the last two parts of the package, if they exist, and forms a class with a `Base` suffix. This rule takes precedence over `baseClassForTests`. The following example shows how it works in the `contracts` closure:

```
packageWithBaseClasses = 'com.example.base'
```

By Mapping

You can manually map a regular expression of the contract's package to the fully qualified name of the base class for the matched contract. You have to provide a list called `baseClassMappings` that consists of `baseClassMapping` objects that take a `contractPackageRegex` to `baseClassFQN` mapping. Consider the following example:

```
baseClassForTests = "com.example.FooBase"
baseClassMappings {
    baseClassMapping('.*com/.*', 'com.example.ComBase')
    baseClassMapping('.*bar/.*': 'com.example.BarBase')
}
```

Let's assume that you have contracts in the following directories: - `src/test/resources/contract/com/` - `src/test/resources/contract/foo/`

By providing `baseClassForTests`, we have a fallback in case mapping did not succeed. (You could also provide the `packageWithBaseClasses` as a fallback.) That way, the tests generated from `src/test/resources/contract/com/` contracts extend the `com.example.ComBase`, whereas the rest of the tests extend `com.example.FooBase`.

14.5.12. Invoking Generated Tests

To ensure that the provider side is compliant with your defined contracts, you need to run the following command:

```
./gradlew generateContractTests test
```

14.5.13. Pushing Stubs to SCM

If you use the SCM repository to keep the contracts and stubs, you might want to automate the step of pushing stubs to the repository. To do that, you can call the `pushStubsToScm` task by running the following command:

```
$ ./gradlew pushStubsToScm
```

Under [Using the SCM Stub Downloader](#) you can find all possible configuration options that you can pass either through the `contractsProperties` field (for example, `contracts { contractsProperties = [foo:"bar"] }`), through the `contractsProperties` method (for example, `contracts { contractsProperties([foo:"bar"]) }`), or through a system property or an environment variable.

14.5.14. Spring Cloud Contract Verifier on the Consumer Side

In a consuming service, you need to configure the Spring Cloud Contract Verifier plugin in exactly the same way as in the case of a provider. If you do not want to use Stub Runner, you need to copy the contracts stored in `src/test/resources/contracts` and generate WireMock JSON stubs by using the following command:

```
./gradlew generateClientStubs
```



The `stubsOutputDir` option has to be set for stub generation to work.

When present, JSON stubs can be used in automated tests to consume a service. The following example shows how to do so:

```

@ContextConfiguration(loader == SpringApplicationContextLoader, classes ==
Application)
class LoanApplicationServiceSpec extends Specification {

    @ClassRule
    @Shared
    WireMockClassRule wireMockRule == new WireMockClassRule()

    @Autowired
    LoanApplicationService sut

    def 'should successfully apply for loan'() {
        given:
            LoanApplication application =
                new LoanApplication(client: new Client(clientPesel: '12345678901'),
amount: 123.123)
        when:
            LoanApplicationResult loanApplication == sut.loanApplication(application)
        then:
            loanApplication.loanApplicationStatus == LoanApplicationStatus.LOAN_APPLIED
            loanApplication.rejectionReason == null
    }
}

```

In the preceding example, `LoanApplication` makes a call to the `FraudDetection` service. This request is handled by a WireMock server configured with stubs that were generated by Spring Cloud Contract Verifier.

14.6. Docker Project

In this section, we publish a `springcloud/spring-cloud-contract` Docker image that contains a project that generates tests and runs them in `EXPLICIT` mode against a running application.



The `EXPLICIT` mode means that the tests generated from contracts send real requests and not the mocked ones.

We also publish a `spring-cloud/spring-cloud-contract-stub-runner` Docker image that starts the standalone version of Stub Runner.

14.6.1. A Short Introduction to Maven, JARs and Binary storage

Since non-JVM projects can use the Docker image, it is good to explain the basic terms behind Spring Cloud Contract packaging defaults.

Parts of the following definitions were taken from the [Maven Glossary](#):

- **Project:** Maven thinks in terms of projects. Projects are all you build. Those projects follow a

well defined “Project Object Model”. Projects can depend on other projects, in which case the latter are called “dependencies”. A project may consist of several subprojects. However, these subprojects are still treated equally as projects.

- **Artifact:** An artifact is something that is either produced or used by a project. Examples of artifacts produced by Maven for a project include JAR files and source and binary distributions. Each artifact is uniquely identified by a group ID and an artifact ID that is unique within a group.
- **JAR:** JAR stands for Java ARchive. Its format is based on the ZIP file format. Spring Cloud Contract packages the contracts and generated stubs in a JAR file.
- **GroupId:** A group ID is a universally unique identifier for a project. While this is often just the project name (for example, `commons-collections`), it is helpful to use a fully-qualified package name to distinguish it from other projects with a similar name (for example, `org.apache.maven`). Typically, when published to the Artifact Manager, the `GroupId` gets slash separated and forms part of the URL. For example, for a group ID of `com.example` and an artifact ID of `application`, the result would be `/com/example/application/`.
- **Classifier:** The Maven dependency notation looks as follows: `groupId:artifactId:version:classifier`. The classifier is an additional suffix passed to the dependency—for example, `stubs` or `sources`. The same dependency (for example, `com.example:application`) can produce multiple artifacts that differ from each other with the classifier.
- **Artifact manager:** When you generate binaries, sources, or packages, you would like them to be available for others to download, reference, or reuse. In the case of the JVM world, those artifacts are generally JARs. For Ruby, those artifacts are gems. For Docker, those artifacts are Docker images. You can store those artifacts in a manager. Examples of such managers include [Artifactory](#) or [Nexus](#).

14.6.2. Generating Tests on the Producer Side

The image searches for contracts under the `/contracts` folder. The output from running the tests is available in the `/spring-cloud-contract/build` folder (useful for debugging purposes).

You can mount your contracts and pass the environment variables. The image then:

- Generates the contract tests
- Runs the tests against the provided URL
- Generates the [WireMock](#) stubs
- Publishes the stubs to a Artifact Manager (optional - turned on by default)

Environment Variables

The Docker image requires some environment variables to point to your running application, to the Artifact manager instance, and so on. The following list describes the environment variables:

- **PROJECT_GROUP:** Your project’s group ID. Defaults to `com.example`.
- **PROJECT_VERSION:** Your project’s version. Defaults to `0.0.1-SNAPSHOT`.

- `PROJECT_NAME`: Your project's artifact id. Defaults to `example`.
- `PRODUCER_STUBS_CLASSIFIER`: Archive classifier used for generated producer stubs. Defaults to `stubs`.
- `REPO_WITH_BINARIES_URL`: URL of your Artifact Manager. Defaults to `localhost:8081/artifactory/libs-release-local`, which is the default URL of `Artifactory` running locally.
- `REPO_WITH_BINARIES_USERNAME`: (optional) Username when the Artifact Manager is secured. Defaults to `admin`.
- `REPO_WITH_BINARIES_PASSWORD`: (optional) Password when the Artifact Manager is secured. Defaults to `password`.
- `PUBLISH_ARTIFACTS`: If set to `true`, publishes the artifact to binary storage. Defaults to `true`.
- `PUBLISH_ARTIFACTS_OFFLINE`: If set to `true`, it will publish the artifacts to local `.m2`. Defaults to `false`.

These environment variables are used when contracts lay in an external repository. To enable this feature, you must set the `EXTERNAL_CONTRACTS_ARTIFACT_ID` environment variable.

- `EXTERNAL_CONTRACTS_GROUP_ID`: Group ID of the project with contracts. Defaults to `com.example`
- `EXTERNAL_CONTRACTS_ARTIFACT_ID`: Artifact ID of the project with contracts.
- `EXTERNAL_CONTRACTS_CLASSIFIER`: Classifier of the project with contracts. Empty by default.
- `EXTERNAL_CONTRACTS_VERSION`: Version of the project with contracts. Defaults to `+`, equivalent to picking the latest.
- `EXTERNAL_CONTRACTS_REPO_WITH_BINARIES_URL`: URL of your Artifact Manager. It defaults to the value of `REPO_WITH_BINARIES_URL` environment variable. If that is not set, it defaults to `localhost:8081/artifactory/libs-release-local`, which is the default URL of `Artifactory` running locally.
- `EXTERNAL_CONTRACTS_REPO_WITH_BINARIES_USERNAME`: (optional) Username if the `EXTERNAL_CONTRACTS_REPO_WITH_BINARIES_URL` requires authentication. It defaults to `REPO_WITH_BINARIES_USERNAME`. If that is not set, it defaults to `admin`.
- `EXTERNAL_CONTRACTS_REPO_WITH_BINARIES_PASSWORD`: (optional) Password if the `EXTERNAL_CONTRACTS_REPO_WITH_BINARIES_URL` requires authentication. It defaults to `REPO_WITH_BINARIES_PASSWORD`. If that is not set, it defaults to `password`.
- `EXTERNAL_CONTRACTS_PATH`: Path to contracts for the given project, inside the project with contracts. Defaults to slash-separated `EXTERNAL_CONTRACTS_GROUP_ID` concatenated with `/` and `EXTERNAL_CONTRACTS_ARTIFACT_ID`. For example, for group id `cat-server-side.dog` and artifact id `fish`, would result in `cat/dog/fish` for the contracts path.
- `EXTERNAL_CONTRACTS_WORK_OFFLINE`; If set to `true`, retrieves the artifact with contracts from the container's `.m2`. Mount your local `.m2` as a volume available at the container's `/root/.m2` path.



You must not set both `EXTERNAL_CONTRACTS_WORK_OFFLINE` and `EXTERNAL_CONTRACTS_REPO_WITH_BINARIES_URL`.

The following environment variables are used when tests are executed:

- **APPLICATION_BASE_URL**: URL against which tests should be run. Remember that it has to be accessible from the Docker container (for example, `localhost` does not work)
- **APPLICATION_USERNAME**: (optional) Username for basic authentication to your application.
- **APPLICATION_PASSWORD**: (optional) Password for basic authentication to your application.

Example of Usage

In this section, we explore a simple MVC application. To get started, clone the following git repository and cd to the resulting directory, by running the following commands:

```
$ git clone https://github.com/spring-cloud-samples/spring-cloud-contract-nodejs
$ cd bookstore
```

The contracts are available in the `/contracts` folder.

Since we want to run tests, we can run the following command:

```
$ npm test
```

However, for learning purposes, we split it into pieces, as follows:

```

# Stop docker infra (nodejs, artifactory)
$ ./stop_infra.sh
# Start docker infra (nodejs, artifactory)
$ ./setup_infra.sh

# Kill & Run app
$ pkill -f "node app"
$ nohup node app &

# Prepare environment variables
$ SC_CONTRACT_DOCKER_VERSION="..."
$ APP_IP="192.168.0.100"
$ APP_PORT="3000"
$ ARTIFACTORY_PORT="8081"
$ APPLICATION_BASE_URL="http://${APP_IP}:${APP_PORT}"
$ ARTIFACTORY_URL="http://${APP_IP}:${ARTIFACTORY_PORT}/artifactory/libs-release-
local"
$ CURRENT_DIR="$( pwd )"
$ CURRENT_FOLDER_NAME=${PWD##*/}
$ PROJECT_VERSION="0.0.1.RELEASE"

# Execute contract tests
$ docker run --rm -e "APPLICATION_BASE_URL=${APPLICATION_BASE_URL}" -e
"PUBLISH_ARTIFACTS=true" -e "PROJECT_NAME=${CURRENT_FOLDER_NAME}" -e
"REPO_WITH_BINARIES_URL=${ARTIFACTORY_URL}" -e
"PROJECT_VERSION=${PROJECT_VERSION}" -v "${CURRENT_DIR}/contracts:/contracts:ro"
-v "${CURRENT_DIR}/node_modules/spring-cloud-contract/output:/spring-cloud-
contract-output/" springcloud/spring-cloud-
contract:"${SC_CONTRACT_DOCKER_VERSION}"

# Kill app
$ pkill -f "node app"

```

Through bash scripts, the following happens:

- The infrastructure (MongoDb and Artifactory) is set up. In a real-life scenario, you would run the NodeJS application with a mocked database. In this example, we want to show how we can benefit from Spring Cloud Contract in very little time.
- Due to those constraints, the contracts also represent the stateful situation.
 - The first request is a **POST** that causes data to get inserted to the database.
 - The second request is a **GET** that returns a list of data with 1 previously inserted element.
- The NodeJS application is started (on port **3000**).
- The contract tests are generated through Docker, and tests are run against the running application.
 - The contracts are taken from **/contracts** folder.

- The output of the test execution is available under `node_modules/spring-cloud-contract/output`.
- The stubs are uploaded to Artifactory. You can find them in `localhost:8081/artifactory/libs-release-local/com/example/bookstore/0.0.1.RELEASE/` . The stubs are at `localhost:8081/artifactory/libs-release-local/com/example/bookstore/0.0.1.RELEASE/bookstore-0.0.1.RELEASE-stubs.jar`.

14.6.3. Running Stubs on the Consumer Side

This section describes how to use Docker on the consumer side to fetch and run stubs.

We publish a `spring-cloud/spring-cloud-contract-stub-runner` Docker image that starts the standalone version of Stub Runner.

Environment Variables

You can run the docker image and pass any of the [Common Properties for JUnit and Spring](#) as environment variables. The convention is that all the letters should be upper case. The dot (.) should be replaced with underscore (_) characters. For example, the `stubrunner.repositoryRoot` property should be represented as a `STUBRUNNER_REPOSITORY_ROOT` environment variable.

Example of Usage

We want to use the stubs created in this [\[docker-server-side\]](#) step. Assume that we want to run the stubs on port `9876`. You can see the NodeJS code by cloning the repository and changing to the directory indicated in the following commands:

```
$ git clone https://github.com/spring-cloud-samples/spring-cloud-contract-nodejs
$ cd bookstore
```

Now we can run the Stub Runner Boot application with the stubs, by running the following commands:

```

# Provide the Spring Cloud Contract Docker version
$ SC_CONTRACT_DOCKER_VERSION="..."
# The IP at which the app is running and Docker container can reach it
$ APP_IP="192.168.0.100"
# Spring Cloud Contract Stub Runner properties
$ STUBRUNNER_PORT="8083"
# Stub coordinates 'groupId:artifactId:version:classifier:port'
$ STUBRUNNER_IDS="com.example:bookstore:0.0.1.RELEASE:stubs:9876"
$ STUBRUNNER_REPOSITORY_ROOT="http://${APP_IP}:8081/artifactory/libs-release-local"
# Run the docker with Stub Runner Boot
$ docker run --rm -e "STUBRUNNER_IDS=${STUBRUNNER_IDS}" -e
"STUBRUNNER_REPOSITORY_ROOT=${STUBRUNNER_REPOSITORY_ROOT}" -e
"STUBRUNNER_STUBS_MODE=REMOTE" -p "${STUBRUNNER_PORT}:${STUBRUNNER_PORT}" -p
"9876:9876" springcloud/spring-cloud-contract-stub-
runner:${SC_CONTRACT_DOCKER_VERSION}"

```

When the preceding commands run,

- A standalone Stub Runner application gets started.
- It downloads the stub with coordinates `com.example:bookstore:0.0.1.RELEASE:stubs` on port `9876`.
- It gets downloads from Artifactory running at `192.168.0.100:8081/artifactory/libs-release-local`.
- After a while, Stub Runner is running on port `8083`.
- The stubs are running at port `9876`.

On the server side, we built a stateful stub. We can use curl to assert that the stubs are setup properly. To do so, run the following commands:

```

# let's execute the first request (no response is returned)
$ curl -H "Content-Type:application/json" -X POST --data '{ "title" : "Title",
"genre" : "Genre", "description" : "Description", "author" : "Author", "publisher"
: "Publisher", "pages" : 100, "image_url" :
"https://d213dhlpdb53mu.cloudfront.net/assets/pivotal-square-logo-
41418bd391196c3022f3cd9f3959b3f6d7764c47873d858583384e759c7db435.svg", "buy_url" :
"https://pivotal.io" }' http://localhost:9876/api/books
# Now time for the second request
$ curl -X GET http://localhost:9876/api/books
# You will receive contents of the JSON

```



If you want use the stubs that you have built locally, on your host, you should set the `-e STUBRUNNER_STUBS_MODE=LOCAL` environment variable and mount the volume of your local m2 (`-v "${HOME}/.m2:/root/.m2:ro`).

14.7. Spring Cloud Contract customization

In this section, we describe how to customize various parts of Spring Cloud Contract.

14.7.1. DSL Customization



This section is valid only for the Groovy DSL

You can customize the Spring Cloud Contract Verifier by extending the DSL, as shown in the remainder of this section.

Extending the DSL

You can provide your own functions to the DSL. The key requirement for this feature is to maintain the static compatibility. Later in this document, you can see examples of:

- Creating a JAR with reusable classes.
- Referencing of these classes in the DSLs.

You can find the full example [here](#).

Common JAR

The following examples show three classes that can be reused in the DSLs.

`PatternUtils` contains functions used by both the consumer and the producer. The following listing shows the `PatternUtils` class:

```

package com.example;

import java.util.regex.Pattern;

/**
 * If you want to use {@link Pattern} directly in your tests
 * then you can create a class resembling this one. It can
 * contain all the {@link Pattern} you want to use in the DSL.
 *
 * <pre>
 * {@code
 * request {
 *     body(
 *         [ age: $(c(PatternUtils.oldEnough()))]
 *     )
 * }
 * </pre>
 *
 * Notice that we're using both {@code $()} for dynamic values
 * and {@code c()} for the consumer side.
 *
 * @author Marcin Grzejszczak
 */
//tag::impl[]
public class PatternUtils {

    public static String tooYoung() {
        //remove::start[]
        return "[0-1][0-9]";
        //remove::end[return]
    }

    public static Pattern oldEnough() {
        //remove::start[]
        return Pattern.compile("[2-9][0-9]");
        //remove::end[return]
    }

    /**
     * Makes little sense but it's just an example ;)
     */
    public static Pattern ok() {
        //remove::start[]
        return Pattern.compile("OK");
        //remove::end[return]
    }
}
//end::impl[]

```

`ConsumerUtils` contains functions used by the consumer. The following listing shows the `ConsumerUtils` class:

```

package com.example;

import org.springframework.cloud.contract.spec.internal.ClientDslProperty;

/**
 * DSL Properties passed to the DSL from the consumer's perspective.
 * That means that on the input side {@code Request} for HTTP
 * or {@code Input} for messaging you can have a regular expression.
 * On the {@code Response} for HTTP or {@code Output} for messaging
 * you have to have a concrete value.
 *
 * @author Marcin Grzejszczak
 */
//tag::impl[]
public class ConsumerUtils {
    /**
     * Consumer side property. By using the {@link ClientDslProperty}
     * you can omit most of boilerplate code from the perspective
     * of dynamic values. Example
     *
     * <pre>
     * {@code
     * request {
     *     body(
     *         [ age: $(ConsumerUtils.oldEnough()) ]
     *     )
     * }
     * </pre>
     *
     * That way it's in the implementation that we decide what value we will pass
     to the consumer
     * and which one to the producer.
     *
     * @author Marcin Grzejszczak
     */
    public static ClientDslProperty oldEnough() {
        //remove::start[]
        // this example is not the best one and
        // theoretically you could just pass the regex instead of
`ServerDslProperty` but
        // it's just to show some new tricks :)
        return new ClientDslProperty(PatternUtils.oldEnough(), 40);
        //remove::end[return]
    }
}

//end::impl[]

```

`ProducerUtils` contains functions used by the producer. The following listing shows the `ProducerUtils` class:

```
package com.example;

import org.springframework.cloud.contract.spec.internal.ServerDslProperty;

/**
 * DSL Properties passed to the DSL from the producer's perspective.
 * That means that on the input side {@code Request} for HTTP
 * or {@code Input} for messaging you have to have a concrete value.
 * On the {@code Response} for HTTP or {@code Output} for messaging
 * you can have a regular expression.
 *
 * @author Marcin Grzejszczak
 */
//tag::impl[]
public class ProducerUtils {

    /**
     * Producer side property. By using the {@link ProducerUtils}
     * you can omit most of boilerplate code from the perspective
     * of dynamic values. Example
     *
     * <pre>
     * {@code
     * response {
     *     body(
     *         [ status: $(ProducerUtils.ok())]
     *     )
     * }
     * </pre>
     *
     * That way it's in the implementation that we decide what value we will pass
     to the consumer
     * and which one to the producer.
     */
    public static ServerDslProperty ok() {
        // this example is not the best one and
        // theoretically you could just pass the regex instead of
        `ServerDslProperty` but
        // it's just to show some new tricks :)
        return new ServerDslProperty( PatternUtils.ok(), "OK");
    }
}
//end::impl[]
```

Adding a Test Dependency in the Project's Dependencies

To add a test dependency in the project's dependencies, you must first add the common jar dependency as a test dependency. Because your contracts files are available on the test resources path, the common jar classes automatically become visible in your Groovy files. The following examples show how to test the dependency:

Maven

```
<dependency>
  <groupId>com.example</groupId>
  <artifactId>beer-common</artifactId>
  <version>${project.version}</version>
  <scope>test</scope>
</dependency>
```

Gradle

```
testCompile("com.example:beer-common:0.0.1.BUILD-SNAPSHOT")
```

Adding a Test Dependency in the Plugin's Dependencies

Now, you must add the dependency for the plugin to reuse at runtime, as the following example shows:

Maven

```
<plugin>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-contract-maven-plugin</artifactId>
  <version>${spring-cloud-contract.version}</version>
  <extensions>>true</extensions>
  <configuration>
    <packageWithBaseClasses>com.example</packageWithBaseClasses>
    <baseClassMappings>
      <baseClassMapping>
        <contractPackageRegex>.*intoxication.*</contractPackageRegex>

<baseClassFQN>com.example.intoxication.BeerIntoxicationBase</baseClassFQN>
      </baseClassMapping>
    </baseClassMappings>
  </configuration>
  <dependencies>
    <dependency>
      <groupId>com.example</groupId>
      <artifactId>beer-common</artifactId>
      <version>${project.version}</version>
      <scope>compile</scope>
    </dependency>
  </dependencies>
</plugin>
```

Gradle

```
classpath "com.example:beer-common:0.0.1.BUILD-SNAPSHOT"
```

Referencing Classes in DSLs

You can now reference your classes in your DSL, as the following example shows:

```

package contracts.beer.rest

import com.example.ConsumerUtils
import com.example.ProducerUtils
import org.springframework.cloud.contract.spec.Contract

Contract.make {
    description("""
Represents a successful scenario of getting a beer
    """)
    given:
        client is old enough
    when:
        he applies for a beer
    then:
        we'll grant him the beer
    """)
    request {
        method 'POST'
        url '/check'
        body(
            age: $(ConsumerUtils.oldEnough())
        )
        headers {
            contentType(applicationJson())
        }
    }
    response {
        status 200
        body("""
            {
                "status": "${value(ProducerUtils.ok())}"
            }
        """)
        headers {
            contentType(applicationJson())
        }
    }
}

```



You can set the Spring Cloud Contract plugin up by setting `convertToYaml` to `true`. That way, you do NOT have to add the dependency with the extended functionality to the consumer side, since the consumer side uses YAML contracts instead of Groovy contracts.

14.7.2. WireMock Customization

In this section, we show how to customize the way you work with [WireMock](#).

Registering Your Own WireMock Extension

WireMock lets you register custom extensions. By default, Spring Cloud Contract registers the transformer, which lets you reference a request from a response. If you want to provide your own extensions, you can register an implementation of the `org.springframework.cloud.contract.verifier.dsl.wiremock.WireMockExtensions` interface. Since we use the `spring.factories` extension approach, you can create an entry in `META-INF/spring.factories` file similar to the following:

```
org.springframework.cloud.contract.verifier.dsl.wiremock.WireMockExtensions=\
org.springframework.cloud.contract.stubrunner.provider.wiremock.TestWireMockExtensions
org.springframework.cloud.contract.spec.ContractConverter=\
org.springframework.cloud.contract.stubrunner.TestCustomYamlContractConverter
```

The following example shows a custom extension:

Example 7. TestWireMockExtensions.groovy

```
/*
 * Copyright 2013-2020 the original author or authors.
 *
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 *     https://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */

package org.springframework.cloud.contract.verifier.dsl.wiremock

import com.github.tomakehurst.wiremock.extension.Extension

/**
 * Extension that registers the default transformer and the custom one
 */
class TestWireMockExtensions implements WireMockExtensions {
    @Override
    List<Extension> extensions() {
        return [
            new DefaultResponseTransformer(),
            new CustomExtension()
        ]
    }
}

class CustomExtension implements Extension {

    @Override
    String getName() {
        return "foo-transformer"
    }
}
```



Remember to override the `applyGlobally()` method and set it to `false` if you want the transformation to be applied only for a mapping that explicitly requires it.

Customization of WireMock Configuration

You can register a bean of type `org.springframework.cloud.contract.wiremock.WireMockConfigurationCustomizer` to customize the WireMock configuration (for example, to add custom transformers). The following example shows how to do so:

```
@Bean
WireMockConfigurationCustomizer optionsCustomizer() {
    return new WireMockConfigurationCustomizer() {
        @Override
        public void customize(WireMockConfiguration options) {
// perform your customization here
        }
    };
}
```

14.7.3. Using the Pluggable Architecture

You may encounter cases where your contracts have been defined in other formats, such as YAML, RAML, or PACT. In those cases, you still want to benefit from the automatic generation of tests and stubs. You can add your own implementation for generating both tests and stubs. Also, you can customize the way tests are generated (for example, you can generate tests for other languages) and the way stubs are generated (for example, you can generate stubs for other HTTP server implementations).

Custom Contract Converter

The `ContractConverter` interface lets you register your own implementation of a contract structure converter. The following code listing shows the `ContractConverter` interface:

```

package org.springframework.cloud.contract.spec;

import java.io.File;
import java.util.Collection;

/**
 * Converter to be used to convert FROM {@link File} TO {@link Contract} and from
 * {@link Contract} to {@code T}.
 *
 * @param <T> - type to which we want to convert the contract
 * @author Marcin Grzejszczak
 * @since 1.1.0
 */
public interface ContractConverter<T> extends ContractStorer<T> {

    /**
     * Should this file be accepted by the converter. Can use the file extension
     to check
     * if the conversion is possible.
     * @param file - file to be considered for conversion
     * @return - {@code true} if the given implementation can convert the file
     */
    boolean isAccepted(File file);

    /**
     * Converts the given {@link File} to its {@link Contract} representation.
     * @param file - file to convert
     * @return - {@link Contract} representation of the file
     */
    Collection<Contract> convertFrom(File file);

    /**
     * Converts the given {@link Contract} to a {@link T} representation.
     * @param contract - the parsed contract
     * @return - {@link T} the type to which we do the conversion
     */
    T convertTo(Collection<Contract> contract);
}

```

Your implementation must define the condition on which it should start the conversion. Also, you must define how to perform that conversion in both directions.



Once you create your implementation, you must create a `/META-INF/spring.factories` file in which you provide the fully qualified name of your implementation.

The following example shows a typical `spring.factories` file:

```
org.springframework.cloud.contract.spec.ContractConverter=\
org.springframework.cloud.contract.verifier.converter.YamlContractConverter
```

Using the Custom Test Generator

If you want to generate tests for languages other than Java or you are not happy with the way the verifier builds Java tests, you can register your own implementation.

The `SingleTestGenerator` interface lets you register your own implementation. The following code listing shows the `SingleTestGenerator` interface:

```
package org.springframework.cloud.contract.verifier.builder;

import java.nio.file.Path;
import java.util.Collection;

import
org.springframework.cloud.contract.verifier.config.ContractVerifierConfigPropertie
S;
import org.springframework.cloud.contract.verifier.file.ContractMetadata;

/**
 * Builds a single test.
 *
 * @since 1.1.0
 */
public interface SingleTestGenerator {

    /**
     * Creates contents of a single test class in which all test scenarios from
     the
     * contract metadata should be placed.
     * @param properties - properties passed to the plugin
     * @param listOfFiles - list of parsed contracts with additional metadata
     * @param className - the name of the generated test class
     * @param classPackage - the name of the package in which the test class
     should be
     * stored
     * @param includedDirectoryRelativePath - relative path to the included
     directory
     * @return contents of a single test class
     * @deprecated use{@link
     SingleTestGenerator#buildClass(ContractVerifierConfigProperties, Collection,
     String, GeneratedClassData)}
     */
}
```

```

@Deprecated
String buildClass(ContractVerifierConfigProperties properties,
                 Collection<ContractMetadata> listOfFiles, String className,
                 String classPackage, String includedDirectoryRelativePath);

/**
 * Creates contents of a single test class in which all test scenarios from
the
 * contract metadata should be placed.
 * @param properties - properties passed to the plugin
 * @param listOfFiles - list of parsed contracts with additional metadata
 * @param generatedClassData - information about the generated class
 * @param includedDirectoryRelativePath - relative path to the included
directory
 * @return contents of a single test class
 */
String buildClass(ContractVerifierConfigProperties properties,
                 Collection<ContractMetadata> listOfFiles,
                 String includedDirectoryRelativePath, GeneratedClassData
generatedClassData);

class GeneratedClassData {

    public final String className;

    public final String classPackage;

    public final Path testClassPath;

    public GeneratedClassData(String className, String classPackage,
                             Path testClassPath) {
        this.className = className;
        this.classPackage = classPackage;
        this.testClassPath = testClassPath;
    }
}
}
}

```

Again, you must provide a `spring.factories` file, such as the one shown in the following example:

```

org.springframework.cloud.contract.verifier.builder.SingleTestGenerator=com.example.MyGenerator

```


Using the Custom Stub Generator

If you want to generate stubs for stub servers other than WireMock, you can plug in your own implementation of the `StubGenerator` interface. The following code listing shows the `StubGenerator` interface:

```
package org.springframework.cloud.contract.verifier.converter;

import java.util.Map;

import org.springframework.cloud.contract.spec.Contract;
import org.springframework.cloud.contract.verifier.file.ContractMetadata;

/**
 * Converts contracts into their stub representation.
 *
 * @since 1.1.0
 */
public interface StubGenerator {

    /**
     * @param fileName - file name
     * @return {@code true} if the converter can handle the file to convert it
     into a
     * stub.
     */
    default boolean canHandleFileName(String fileName) {
        return fileName.endsWith(fileExtension());
    }

    /**
     * @param rootName - root name of the contract
     * @param content - metadata of the contract
     * @return the collection of converted contracts into stubs. One contract can
     result
     * in multiple stubs.
     */
    Map<Contract, String> convertContents(String rootName, ContractMetadata
    content);

    /**
     * @param inputFileName - name of the input file
     * @return the name of the converted stub file. If you have multiple contracts
     in a
     * single file then a prefix will be added to the generated file. If you
     provide the
     * {@link Contract#name} field then that field will override the generated
     file name.
     *
     * Example: name of file with 2 contracts is {@code foo.groovy}, it will be

```

```

converted
    * by the implementation to {@code foo.json}. The recursive file converter
will create
    * two files {@code 0_foo.json} and {@code 1_foo.json}
    */
String generateOutputFileNameForInput(String inputFileName);

/**
 * Describes the file extension that this stub generator can handle.
 * @return string describing the file extension
 */
default String fileExtension() {
    return ".json";
}
}

```

Again, you must provide a `spring.factories` file, such as the one shown in the following example:

```

# Stub converters
org.springframework.cloud.contract.verifier.converter.StubGenerator=\
org.springframework.cloud.contract.verifier.wiremock.DslToWireMockClientConverter

```

The default implementation is the WireMock stub generation.



You can provide multiple stub generator implementations. For example, from a single DSL, you can produce both WireMock stubs and Pact files.

Using the Custom Stub Runner

If you decide to use a custom stub generation, you also need a custom way of running stubs with your different stub provider.

Assume that you use [Moco](#) to build your stubs and that you have written a stub generator and placed your stubs in a JAR file.

In order for Stub Runner to know how to run your stubs, you have to define a custom HTTP Stub server implementation, which might resemble the following example:

```

package org.springframework.cloud.contract.stubrunner.provider.moco

import com.github.dreamhead.moco.bootstrap.arg.HttpArgs
import com.github.dreamhead.moco.runner.JsonRunner
import com.github.dreamhead.moco.runner.RunnerSetting
import groovy.transform.CompileStatic

```

```
import groovy.util.logging.Commons

import org.springframework.cloud.contract.stubrunner.HttpServerStub
import org.springframework.util.SocketUtils

@Commons
@CompileStatic
class MocoHttpServerStub implements HttpServerStub {

    private boolean started
    private JsonRunner runner
    private int port

    @Override
    int port() {
        if (!isRunning()) {
            return -1
        }
        return port
    }

    @Override
    boolean isRunning() {
        return started
    }

    @Override
    HttpServerStub start() {
        return start(SocketUtils.findAvailableTcpPort())
    }

    @Override
    HttpServerStub start(int port) {
        this.port = port
        return this
    }

    @Override
    HttpServerStub stop() {
        if (!isRunning()) {
            return this
        }
        this.runner.stop()
        return this
    }

    @Override
    HttpServerStub registerMappings(Collection<File> stubFiles) {
        List<RunnerSetting> settings = stubFiles.findAll {
            it.name.endsWith("json") }
            .collect {
```

```

        log.info("Trying to parse [${it.name}]")
        try {
            return
            RunnerSetting.aRunnerSetting().addStream(it.newInputStream()).
                build()
        }
        catch (Exception e) {
            log.warn("Exception occurred while trying to parse file
[${it.name}]", e)
            return null
        }
    }.findAll { it }
    this.runner = JsonRunner.newJsonRunnerWithSetting(settings,
        HttpArgs.httpArgs().withPort(this.port).build())
    this.runner.run()
    this.started = true
    return this
}

@Override
String registeredMappings() {
    return ""
}

@Override
boolean isAccepted(File file) {
    return file.name.endsWith(".json")
}
}

```

Then you can register it in your `spring.factories` file, as the following example shows:

```

org.springframework.cloud.contract.stubrunner.HttpServerStub=\
org.springframework.cloud.contract.stubrunner.provider.moco.MocoHttpServerStub

```

Now you can run stubs with Moco.



If you do not provide any implementation, the default (WireMock) implementation is used. If you provide more than one, the first one on the list is used.

Using the Custom Stub Downloader

You can customize the way your stubs are downloaded by creating an implementation of the `StubDownloaderBuilder` interface, as the following example shows:

```

package com.example;

class CustomStubDownloaderBuilder implements StubDownloaderBuilder {

    @Override
    public StubDownloader build(final StubRunnerOptions stubRunnerOptions) {
        return new StubDownloader() {
            @Override
            public Map.Entry<StubConfiguration, File> downloadAndUnpackStubJar(
                StubConfiguration config) {
                File unpackedStubs = retrieveStubs();
                return new AbstractMap.SimpleEntry<>(
                    new StubConfiguration(config.getGroupId(),
config.getArtifactId(), version,
                                config.getClassifier()), unpackedStubs);
            }

            File retrieveStubs() {
                // here goes your custom logic to provide a folder where all the
stubs reside
            }
        }
    }
}

```

Then you can register it in your `spring.factories` file, as the following example shows:

```

# Example of a custom Stub Downloader Provider
org.springframework.cloud.contract.stubrunner.StubDownloaderBuilder=\
com.example.CustomStubDownloaderBuilder

```

Now you can pick a folder with the source of your stubs.



If you do not provide any implementation, the default (scanning the classpath) is used. If you provide the `stubsMode = StubRunnerProperties.StubsMode.LOCAL` or `stubsMode = StubRunnerProperties.StubsMode.REMOTE`, the Aether implementation is used. If you provide more than one, the first one on the list is used.

Using the SCM Stub Downloader

Whenever the `repositoryRoot` starts with a SCM protocol (currently, we support only `git://`), the stub downloader tries to clone the repository and use it as a source of contracts to generate tests or stubs.

Through environment variables, system properties, or properties set inside the plugin or the contracts repository configuration, you can tweak the downloader's behavior. The following table

describes the available properties:

Table 5. SCM Stub Downloader properties

Type of a property	Name of the property	Description
* <code>git.branch</code> (plugin prop) * <code>stubrunner.properties.git.branch</code> (system prop) * <code>STUBRUNNER_PROPERTIES_GIT_BRANCH</code> (env prop)	master	Which branch to checkout
* <code>git.username</code> (plugin prop) * <code>stubrunner.properties.git.username</code> (system prop) * <code>STUBRUNNER_PROPERTIES_GIT_USERNAME</code> (env prop)		Git clone username
* <code>git.password</code> (plugin prop) * <code>stubrunner.properties.git.password</code> (system prop) * <code>STUBRUNNER_PROPERTIES_GIT_PASSWORD</code> (env prop)		Git clone password
* <code>git.no-of-attempts</code> (plugin prop) * <code>stubrunner.properties.git.no-of-attempts</code> (system prop) * <code>STUBRUNNER_PROPERTIES_GIT_NO_OF_ATTEMPTS</code> (env prop)	10	Number of attempts to push the commits to <code>origin</code>

* <code>git.wait-between-attempts</code> (Plugin prop)	1000	Number of milliseconds to wait between attempts to push the commits to <code>origin</code>
* <code>stubrunner.properties.git.wait-between-attempts</code> (system prop)		
* <code>STUBRUNNER_PROPERTIES_GIT_WAIT_BETWEEN_ATTEMPTS</code> (env prop)		

14.8. “How-to” Guides

This section provides answers to some common “how do I do that...” questions that often arise when using Spring Cloud Contract. Its coverage is not exhaustive, but it does cover quite a lot.

If you have a specific problem that we do not cover here, you might want to check out stackoverflow.com to see if someone has already provided an answer. Stack Overflow is also a great place to ask new questions (please use the `spring-cloud` tag).

We are also more than happy to extend this section. If you want to add a “how-to”, send us a [pull request](#).

14.8.1. Why use Spring Cloud Contract?

Spring Cloud Contract works great in a polyglot environment. This project has a lot of really interesting features. Quite a few of these features definitely make Spring Cloud Contract Verifier stand out on the market of Consumer Driven Contract (CDC) tooling. The most interesting features include the following:

- Ability to do CDC with messaging.
- Clear and easy to use, statically typed DSL.
- Ability to copy-paste your current JSON file to the contract and only edit its elements.
- Automatic generation of tests from the defined Contract.
- Stub Runner functionality: The stubs are automatically downloaded at runtime from Nexus/Artifactory.
- Spring Cloud integration: No discovery service is needed for integration tests.
- Spring Cloud Contract integrates with Pact and provides easy hooks to extend its functionality.
- Ability to add support for any language & framework through Docker.

14.8.2. How Can I Write Contracts in a Language Other than Groovy?

You can write a contract in YAML. See [this section](#) for more information.

We are working on allowing more ways of describing the contracts. You can check the [github-issues](#) for more information.

14.8.3. How Can I Provide Dynamic Values to a Contract?

One of the biggest challenges related to stubs is their reusability. Only if they can be widely used can they serve their purpose. The hard-coded values (such as dates and IDs) of request and response elements generally make that difficult. Consider the following JSON request:

```
{
  "time" : "2016-10-10 20:10:15",
  "id" : "9febab1c-6f36-4a0b-88d6-3b6a6d81cd4a",
  "body" : "foo"
}
```

Now consider the following JSON response:

```
{
  "time" : "2016-10-10 21:10:15",
  "id" : "c4231e1f-3ca9-48d3-b7e7-567d55f0d051",
  "body" : "bar"
}
```

Imagine the pain required to set the proper value of the `time` field (assume that this content is generated by the database) by changing the clock in the system or by providing stub implementations of data providers. The same is related to the field called `id`. You could create a stubbed implementation of UUID generator, but doing so makes little sense.

So, as a consumer, you want to send a request that matches any form of a time or any UUID. That way, your system works as usual, generating data without you having to stub out anything. Assume that, in case of the aforementioned JSON, the most important part is the `body` field. You can focus on that and provide matching for other fields. In other words, you would like the stub to work as follows:

```
{
  "time" : "SOMETHING THAT MATCHES TIME",
  "id" : "SOMETHING THAT MATCHES UUID",
  "body" : "foo"
}
```

As far as the response goes, as a consumer, you need a concrete value on which you can operate. Consequently, the following JSON is valid:


```
{
  "time" : "2016-10-10 21:10:15",
  "id" : "c4231e1f-3ca9-48d3-b7e7-567d55f0d051",
  "body" : "bar"
}
```

In the previous sections, we generated tests from contracts. So, from the producer's side, the situation looks much different. We parse the provided contract, and, in the test, we want to send a real request to your endpoints. So, for the case of a producer for the request, we cannot have any sort of matching. We need concrete values on which the producer's backend can work. Consequently, the following JSON would be valid:

```
{
  "time" : "2016-10-10 20:10:15",
  "id" : "9febab1c-6f36-4a0b-88d6-3b6a6d81cd4a",
  "body" : "foo"
}
```

On the other hand, from the point of view of the validity of the contract, the response does not necessarily have to contain concrete values for `time` or `id`. Suppose you generate those on the producer side. Again, you have to do a lot of stubbing to ensure that you always return the same values. That is why, from the producer's side you might want the following response:

```
{
  "time" : "SOMETHING THAT MATCHES TIME",
  "id" : "SOMETHING THAT MATCHES UUID",
  "body" : "bar"
}
```

How can you then provide a matcher for the consumer and a concrete value for the producer (and the opposite at some other time)? Spring Cloud Contract lets you provide a dynamic value. That means that it can differ for both sides of the communication.

You can read more about this in the [Contract DSL](#) section.



Read the [Groovy docs related to JSON](#) to understand how to properly structure the request and response bodies.

14.8.4. How to Do Stubs versioning?

This section covers version of the stubs, which you can handle in a number of different ways:

- [API Versioning](#)
- [JAR versioning](#)
- [Development or Production Stubs](#)

API Versioning

What does versioning really mean? If you refer to the API version, there are different approaches:

- Use hypermedia links and do not version your API by any means
- Pass the version through headers and URLs

We do not try to answer the question of which approach is better. You should pick whatever suits your needs and lets you generate business value.

Assume that you do version your API. In that case, you should provide as many contracts with as many versions as you support. You can create a subfolder for every version or append it to the contract name — whatever suits you best.

JAR versioning

If, by versioning, you mean the version of the JAR that contains the stubs, then there are essentially two main approaches.

Assume that you do continuous delivery and deployment, which means that you generate a new version of the jar each time you go through the pipeline and that the jar can go to production at any time. For example, your jar version looks like the following (because it got built on the 20.10.2016 at 20:15:21) :

```
1.0.0.20161020-201521-RELEASE
```

In that case your, generated stub jar should look like the following:

```
1.0.0.20161020-201521-RELEASE-stubs.jar
```

In this case, you should, inside your `application.yml` or `@AutoConfigureStubRunner` when referencing stubs, provide the latest version of the stubs. You can do that by passing the `+` sign. the following example shows how to do so:

```
@AutoConfigureStubRunner(ids = {"com.example:http-server-dsl:+:stubs:8080"})
```

If the versioning, however, is fixed (for example, `1.0.4.RELEASE` or `2.1.1`), you have to set the

concrete value of the jar version. The following example shows how to do so for version 2.1.1:

```
@AutoConfigureStubRunner(ids = {"com.example:http-server-dsl:2.1.1:stubs:8080"})
```

Development or Production Stubs

You can manipulate the classifier to run the tests against current the development version of the stubs of other services or the ones that were deployed to production. If you alter your build to deploy the stubs with the `prod-stubs` classifier once you reach production deployment, you can run tests in one case with development stubs and one with production stubs.

The following example works for tests that use the development version of the stubs:

```
@AutoConfigureStubRunner(ids = {"com.example:http-server-dsl+:stubs:8080"})
```

The following example works for tests that use the production version of stubs:

```
@AutoConfigureStubRunner(ids = {"com.example:http-server-dsl+:prod-stubs:8080"})
```

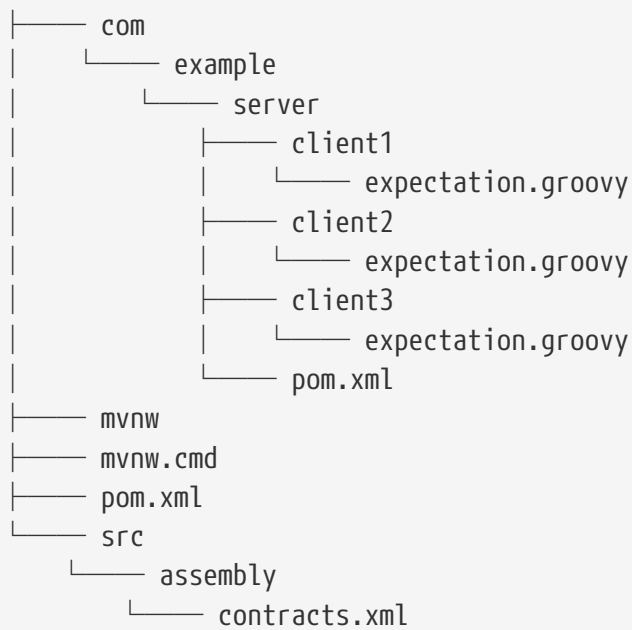
You can also pass those values also in properties from your deployment pipeline.

14.8.5. How Can I use a Common Repository with Contracts Instead of Storing Them with the Producer?

Another way of storing contracts, rather than having them with the producer, is to keep them in a common place. This situation can be related to security issues (where the consumers cannot clone the producer's code). Also if you keep contracts in a single place, then you, as a producer, know how many consumers you have and which consumer you may break with your local changes.

Repo Structure

Assume that we have a producer with coordinates of `com.example:server` and three consumers: `client1`, `client2`, and `client3`. Then, in the repository with common contracts, you could have the following setup (which you can check out [here](#)). The following listing shows such a structure:



As you can see under the slash-delimited `groupid/artifact id` folder (`com/example/server`) you have expectations of the three consumers (`client1`, `client2`, and `client3`). Expectations are the standard Groovy DSL contract files, as described throughout this documentation. This repository has to produce a JAR file that maps one-to-one to the contents of the repository.

The following example shows a `pom.xml` inside the `server` folder:

```

<?xml version="1.0" encoding="UTF-8"?>
<project xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns="http://maven.apache.org/POM/4.0.0"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
https://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <groupId>com.example</groupId>
  <artifactId>server</artifactId>
  <version>0.0.1</version>

  <name>Server Stubs</name>
  <description>POM used to install locally stubs for consumer side</description>

  <parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.3.0.M4</version>
    <relativePath/>
  </parent>

  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>

```

```

<java.version>1.8</java.version>
<spring-cloud-contract.version>3.0.0-SNAPSHOT</spring-cloud-
contract.version>
<spring-cloud-release.version>2020.0.0-SNAPSHOT</spring-cloud-
release.version>
<excludeBuildFolders>>true</excludeBuildFolders>
</properties>

<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>org.springframework.cloud</groupId>
      <artifactId>spring-cloud-dependencies</artifactId>
      <version>${spring-cloud-release.version}</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
  </dependencies>
</dependencyManagement>

<build>
  <plugins>
    <plugin>
      <groupId>org.springframework.cloud</groupId>
      <artifactId>spring-cloud-contract-maven-plugin</artifactId>
      <version>${spring-cloud-contract.version}</version>
      <extensions>>true</extensions>
      <configuration>
        <!-- By default it would search under src/test/resources/ -->
        <contractsDirectory>${project.basedir}</contractsDirectory>
      </configuration>
    </plugin>
  </plugins>
</build>

<repositories>
  <repository>
    <id>spring-snapshots</id>
    <name>Spring Snapshots</name>
    <url>https://repo.spring.io/snapshot</url>
    <snapshots>
      <enabled>>true</enabled>
    </snapshots>
  </repository>
  <repository>
    <id>spring-milestones</id>
    <name>Spring Milestones</name>
    <url>https://repo.spring.io/milestone</url>
    <snapshots>
      <enabled>>false</enabled>
    </snapshots>
  </repository>
</repositories>

```

```

</repository>
<repository>
  <id>spring-releases</id>
  <name>Spring Releases</name>
  <url>https://repo.spring.io/release</url>
  <snapshots>
    <enabled>>false</enabled>
  </snapshots>
</repository>
</repositories>
<pluginRepositories>
  <pluginRepository>
    <id>spring-snapshots</id>
    <name>Spring Snapshots</name>
    <url>https://repo.spring.io/snapshot</url>
    <snapshots>
      <enabled>>true</enabled>
    </snapshots>
  </pluginRepository>
  <pluginRepository>
    <id>spring-milestones</id>
    <name>Spring Milestones</name>
    <url>https://repo.spring.io/milestone</url>
    <snapshots>
      <enabled>>false</enabled>
    </snapshots>
  </pluginRepository>
  <pluginRepository>
    <id>spring-releases</id>
    <name>Spring Releases</name>
    <url>https://repo.spring.io/release</url>
    <snapshots>
      <enabled>>false</enabled>
    </snapshots>
  </pluginRepository>
</pluginRepositories>
</project>

```

There are no dependencies other than the Spring Cloud Contract Maven Plugin. Those pom files are necessary for the consumer side to run `mvn clean install -DskipTests` to locally install the stubs of the producer project.

The `pom.xml` in the root folder can look like the following:

```

<?xml version="1.0" encoding="UTF-8"?>
<project xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns="http://maven.apache.org/POM/4.0.0"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
https://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <groupId>com.example.standalone</groupId>
  <artifactId>contracts</artifactId>
  <version>0.0.1</version>

  <name>Contracts</name>
  <description>Contains all the Spring Cloud Contracts, well, contracts. JAR
used by the
  producers to generate tests and stubs
  </description>

  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
  </properties>

  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-assembly-plugin</artifactId>
        <executions>
          <execution>
            <id>contracts</id>
            <phase>prepare-package</phase>
            <goals>
              <goal>single</goal>
            </goals>
            <configuration>
              <attach>true</attach>
            </configuration>
          </execution>
        </executions>
      </plugin>
    </plugins>

    <descriptor>${basedir}/src/assembly/contracts.xml</descriptor>
    <!-- If you want an explicit classifier remove the
following line -->
      <appendAssemblyId>false</appendAssemblyId>
    </configuration>
  </executions>
</build>

</project>

```

It uses the assembly plugin to build the JAR with all the contracts. The following example shows such a setup:

```
<assembly xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns="http://maven.apache.org/plugins/maven-assembly-
  plugin/assembly/1.1.3"
  xsi:schemaLocation="http://maven.apache.org/plugins/maven-assembly-
  plugin/assembly/1.1.3 https://maven.apache.org/xsd/assembly-1.1.3.xsd">
  <id>project</id>
  <formats>
    <format>jar</format>
  </formats>
  <includeBaseDirectory>>false</includeBaseDirectory>
  <fileSets>
    <fileSet>
      <directory>${project.basedir}</directory>
      <outputDirectory></outputDirectory>
      <useDefaultExcludes>>true</useDefaultExcludes>
      <excludes>
        <exclude>**/${project.build.directory}/**</exclude>
        <exclude>mvnw</exclude>
        <exclude>mvnw.cmd</exclude>
        <exclude>.mvn/**</exclude>
        <exclude>src/**</exclude>
      </excludes>
    </fileSet>
  </fileSets>
</assembly>
```

Workflow

The workflow assumes that Spring Cloud Contract is set up both on the consumer and on the producer side. There is also the proper plugin setup in the common repository with contracts. The CI jobs are set for a common repository to build an artifact of all contracts and upload it to Nexus/Artifactory. The following image shows the UML for this workflow:

[how to common repo] | [how-to-common-repo.png](#)

Consumer

When the consumer wants to work on the contracts offline, instead of cloning the producer code, the consumer team clones the common repository, goes to the required producer's folder (for example, `com/example/server`) and runs `mvn clean install -DskipTests` to locally install the stubs converted from the contracts.



You need to have [Maven installed locally](#)

Producer

As a producer, you can alter the Spring Cloud Contract Verifier to provide the URL and the dependency of the JAR that contains the contracts, as follows:

```
<plugin>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-contract-maven-plugin</artifactId>
  <configuration>
    <contractsMode>REMOTE</contractsMode>
    <contractsRepositoryUrl>
      https://link/to/your/nexus/or/artifactory/or/sth
    </contractsRepositoryUrl>
    <contractDependency>
      <groupId>com.example.standalone</groupId>
      <artifactId>contracts</artifactId>
    </contractDependency>
  </configuration>
</plugin>
```

With this setup, the JAR with a groupId of `com.example.standalone` and artifactid `contracts` is downloaded from `link/to/your/nexus/or/artifactory/or/sth`. It is then unpacked in a local temporary folder, and the contracts present in `com/example/server` are picked as the ones used to generate the tests and the stubs. Due to this convention, the producer team can know which consumer teams will be broken when some incompatible changes are made.

The rest of the flow looks the same.

How Can I Define Messaging Contracts per Topic Rather than per Producer?

To avoid messaging contracts duplication in the common repository, when a few producers write messages to one topic, we could create a structure in which the REST contracts are placed in a folder per producer and messaging contracts are placed in the folder per topic.

For Maven Projects

To make it possible to work on the producer side, we should specify an inclusion pattern for filtering common repository jar files by messaging topics we are interested in. The `includedFiles` property of the Maven Spring Cloud Contract plugin lets us do so. Also, `contractsPath` need to be specified, since the default path would be the common repository `groupId/artifactid`. The following example shows a Maven plugin for Spring Cloud Contract:

```

<plugin>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-contract-maven-plugin</artifactId>
  <version>${spring-cloud-contract.version}</version>
  <configuration>
    <contractsMode>REMOTE</contractsMode>

    <contractsRepositoryUrl>https://link/to/your/nexus/or/artifactory/or/sth</contractsRepositoryUrl>
    <contractDependency>
      <groupId>com.example</groupId>
      <artifactId>common-repo-with-contracts</artifactId>
      <version>+</version>
    </contractDependency>
    <contractsPath></contractsPath>
    <baseClassMappings>
      <baseClassMapping>
        <contractPackageRegex>.*messaging.*</contractPackageRegex>
        <baseClassFQN>com.example.services.MessagingBase</baseClassFQN>
      </baseClassMapping>
      <baseClassMapping>
        <contractPackageRegex>.*rest.*</contractPackageRegex>
        <baseClassFQN>com.example.services.TestBase</baseClassFQN>
      </baseClassMapping>
    </baseClassMappings>
    <includedFiles>
      <includedFile>**/${project.artifactId}/**</includedFile>
      <includedFile>**/${first-topic}/**</includedFile>
      <includedFile>**/${second-topic}/**</includedFile>
    </includedFiles>
  </configuration>
</plugin>

```



Many of the values in the preceding Maven plugin can be changed. We included it for illustration purposes rather than trying to provide a “typical” example.

For Gradle Projects

To work with a Gradle project:

1. Add a custom configuration for the common repository dependency, as follows:

```
ext {
    contractsGroupId = "com.example"
    contractsArtifactId = "common-repo"
    contractsVersion = "1.2.3"
}

configurations {
    contracts {
        transitive = false
    }
}
```

2. Add the common repository dependency to your classpath, as follows:

```
dependencies {
    contracts "${contractsGroupId}:${contractsArtifactId}:${contractsVersion}"
    testCompile
    "${contractsGroupId}:${contractsArtifactId}:${contractsVersion}"
}
```

3. Download the dependency to an appropriate folder, as follows:

```
task getContracts(type: Copy) {
    from configurations.contracts
    into new File(project.buildDir, "downloadedContracts")
}
```

4. Unzip the JAR, as follows:

```
task unzipContracts(type: Copy) {
    def zipFile = new File(project.buildDir,
    "downloadedContracts/${contractsArtifactId}-${contractsVersion}.jar")
    def outputDir = file("${buildDir}/unpackedContracts")

    from zipTree(zipFile)
    into outputDir
}
```

5. Cleanup unused contracts, as follows:

```

task deleteUnwantedContracts(type: Delete) {
    delete fileTree(dir: "${buildDir}/unpackedContracts",
        include: "**/*",
        excludes: [
            "**/${project.name}/**",
            "**/${first-topic}/**",
            "**/${second-topic}/**"])
}

```

6. Create task dependencies, as follows:

```

unzipContracts.dependsOn("getContracts")
deleteUnwantedContracts.dependsOn("unzipContracts")
build.dependsOn("deleteUnwantedContracts")

```

7. Configure the plugin by specifying the directory that contains the contracts, by setting the `contractsDslDir` property, as follows:

```

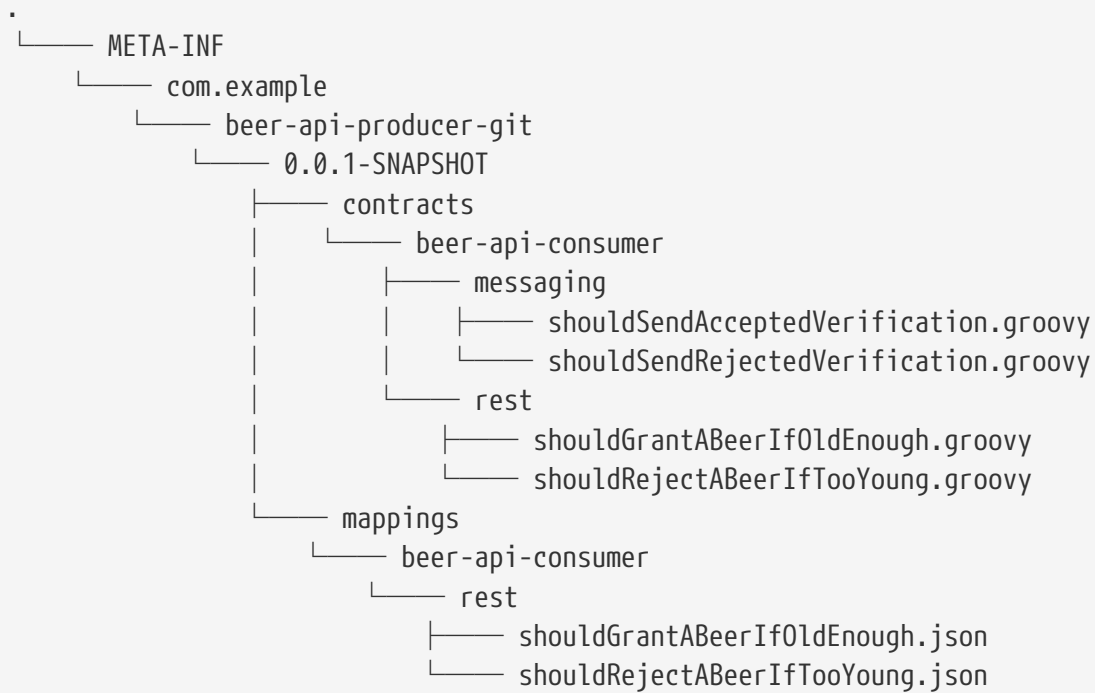
contracts {
    contractsDslDir = new File("${buildDir}/unpackedContracts")
}

```

14.8.6. How Can I Use Git as the Storage for Contracts and Stubs?

In the polyglot world, there are languages that do not use binary storages, as Artifactory or Nexus do. Starting from Spring Cloud Contract version 2.0.0, we provide mechanisms to store contracts and stubs in a SCM (Source Control Management) repository. Currently, the only supported SCM is Git.

The repository would have to have the following setup (which you can checkout from [here](#)):



Under the **META-INF** folder:

- We group applications by **groupId** (such as **com.example**).
- Each application is represented by its **artifactId** (for example, **beer-api-producer-git**).
- Next, each application is organized by its version (such as **0.0.1-SNAPSHOT**). Starting from Spring Cloud Contract version **2.1.0**, you can specify the versions as follows (assuming that your versions follow semantic versioning):
 - **+** or **latest**: To find the latest version of your stubs (assuming that the snapshots are always the latest artifact for a given revision number). That means:
 - If you have **1.0.0.RELEASE**, **2.0.0.BUILD-SNAPSHOT**, and **2.0.0.RELEASE**, we assume that the latest is **2.0.0.BUILD-SNAPSHOT**.
 - If you have **1.0.0.RELEASE** and **2.0.0.RELEASE**, we assume that the latest is **2.0.0.RELEASE**.
 - If you have a version called **latest** or **+**, we will pick that folder.
 - **release**: To find the latest release version of your stubs. That means:
 - If you have **1.0.0.RELEASE**, **2.0.0.BUILD-SNAPSHOT**, and **2.0.0.RELEASE** we assume that the latest is **2.0.0.RELEASE**.
 - If you have a version called **release**, we pick that folder.

Finally, there are two folders:

- **contracts**: The good practice is to store the contracts required by each consumer in the folder with the consumer name (such as **beer-api-consumer**). That way, you can use the **stubs-per-consumer** feature. Further directory structure is arbitrary.
- **mappings**: The Maven or Gradle Spring Cloud Contract plugins push the stub server mappings in

this folder. On the consumer side, Stub Runner scans this folder to start stub servers with stub definitions. The folder structure is a copy of the one created in the `contracts` subfolder.

Protocol Convention

To control the type and location of the source of contracts (whether binary storage or an SCM repository), you can use the protocol in the URL of the repository. Spring Cloud Contract iterates over registered protocol resolvers and tries to fetch the contracts (by using a plugin) or stubs (from Stub Runner).

For the SCM functionality, currently, we support the Git repository. To use it, in the property where the repository URL needs to be placed, you have to prefix the connection URL with `git://`. The following listing shows some examples:

```
git://file:///foo/bar
git://https://github.com/spring-cloud-samples/spring-cloud-contract-nodejs-
contracts-git.git
git://git@github.com:spring-cloud-samples/spring-cloud-contract-nodejs-contracts-
git.git
```

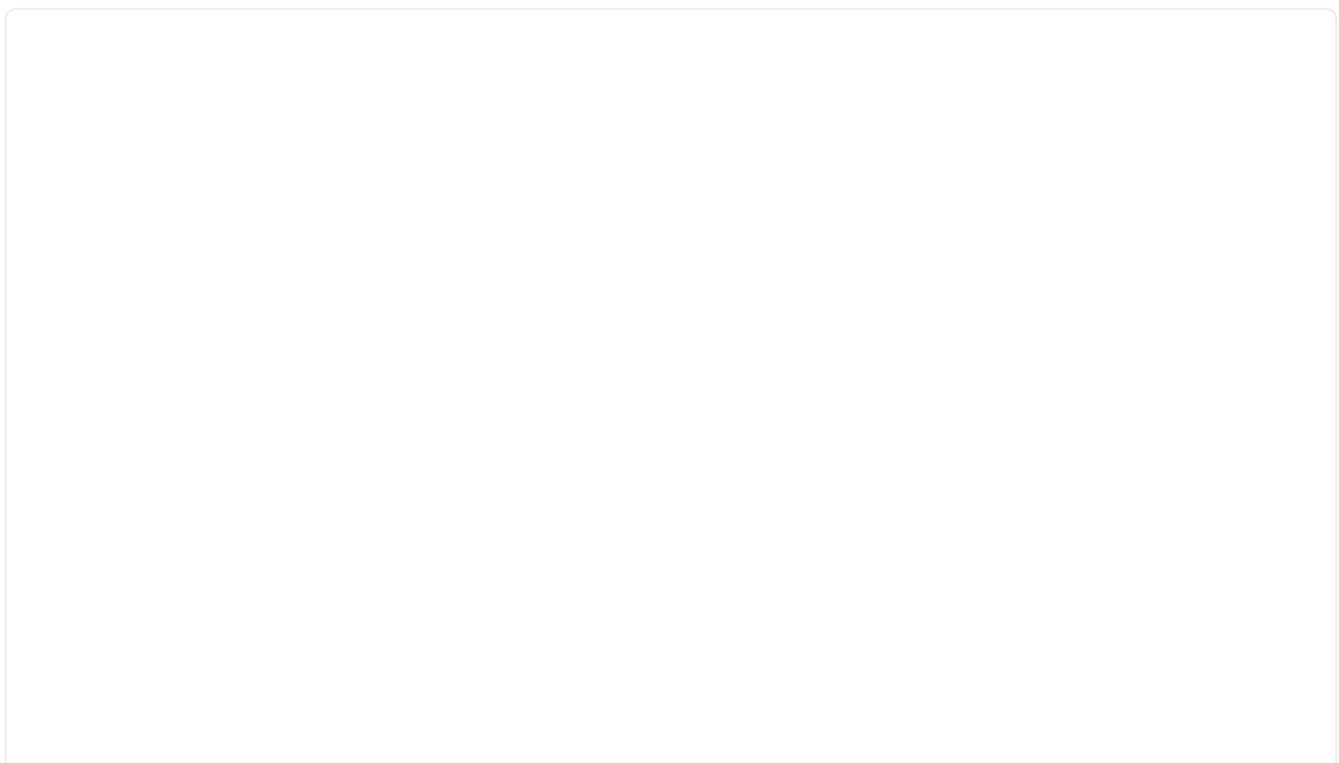
Producer

For the producer, to use the SCM (Source Control Management) approach, we can reuse the same mechanism we use for external contracts. We route Spring Cloud Contract to use the SCM implementation from the URL that starts with the `git://` protocol.



You have to manually add the `pushStubsToScm` goal in Maven or execute (bind) the `pushStubsToScm` task in Gradle. We do not push stubs to the `origin` of your git repository.

The following listing includes the relevant parts both Maven and Gradle build files:



```
<plugin>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-contract-maven-plugin</artifactId>
  <version>${spring-cloud-contract.version}</version>
  <extensions>>true</extensions>
  <configuration>
    <!-- Base class mappings etc. -->

    <!-- We want to pick contracts from a Git repository -->
    <contractsRepositoryUrl>git://https://github.com/spring-cloud-
samples/spring-cloud-contract-nodejs-contracts-git.git</contractsRepositoryUrl>

    <!-- We reuse the contract dependency section to set up the path
to the folder that contains the contract definitions. In our case the
path will be /groupId/artifactId/version/contracts -->
    <contractDependency>
      <groupId>${project.groupId}</groupId>
      <artifactId>${project.artifactId}</artifactId>
      <version>${project.version}</version>
    </contractDependency>

    <!-- The contracts mode can't be classpath -->
    <contractsMode>REMOTE</contractsMode>
  </configuration>
  <executions>
    <execution>
      <phase>package</phase>
      <goals>
        <!-- By default we will not push the stubs back to SCM,
you have to explicitly add it as a goal -->
        <goal>pushStubsToScm</goal>
      </goals>
    </execution>
  </executions>
</plugin>
```

gradle

```
contracts {
    // We want to pick contracts from a Git repository
    contractDependency {
        stringNotation = "${project.group}:${project.name}:${project.version}"
    }
    /*
    We reuse the contract dependency section to set up the path
    to the folder that contains the contract definitions. In our case the
    path will be /groupId/artifactId/version/contracts
    */
    contractRepository {
        repositoryUrl = "git://https://github.com/spring-cloud-samples/spring-
cloud-contract-nodejs-contracts-git.git"
    }
    // The mode can't be classpath
    contractsMode = "REMOTE"
    // Base class mappings etc.
}

/*
In this scenario we want to publish stubs to SCM whenever
the `publish` task is executed
*/
publish.dependsOn("publishStubsToScm")
```

It is also possible to further customise the `publishStubsToScm` gradle task. In the following example, the task is customised to pick contracts from a local git repository:

gradle

```
publishStubsToScm {
    // We want to modify the default set up of the plugin when publish stubs to
    scm is called
    // We want to pick contracts from a Git repository
    contractDependency {
        stringNotation = "${project.group}:${project.name}:${project.version}"
    }
    /*
    We reuse the contract dependency section to set up the path
    to the folder that contains the contract definitions. In our case the
    path will be /groupId/artifactId/version/contracts
    */
    contractRepository {
        repositoryUrl = "git://file://${new File(project.rootDir,
"../target")}/contract_empty_git/"
    }
    // We set the contracts mode to `LOCAL`
    contractsMode = "LOCAL"
}
```

IMPORTANT

Starting with `2.3.0.RELEASE` the `customize{}` closure previously used for the `publishStubsToScm` customization is no longer available. The settings should be applied directly within the `publishStubsToScm` closure as in the example above.

With such a setup:

- A git project is cloned to a temporary directory
- The SCM stub downloader goes to `META-INF/groupId/artifactId/version/contracts` folder to find contracts. For example, for `com.example:foo:1.0.0`, the path would be `META-INF/com.example/foo/1.0.0/contracts`.
- Tests are generated from the contracts.
- Stubs are created from the contracts.
- Once the tests pass, the stubs are committed in the cloned repository.
- Finally, a push is sent to that repo's `origin`.

Producer with Contracts Stored Locally

Another option to use the SCM as the destination for stubs and contracts is to store the contracts locally, with the producer, and only push the contracts and the stubs to SCM. The following listing shows the setup required to achieve this with Maven and Gradle:

maven

```

<plugin>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-contract-maven-plugin</artifactId>
  <version>${spring-cloud-contract.version}</version>
  <extensions>>true</extensions>
  <!-- In the default configuration, we want to use the contracts stored locally
-->
  <configuration>
    <baseClassMappings>
      <baseClassMapping>
        <contractPackageRegex>.*messaging.*</contractPackageRegex>
        <baseClassFQN>com.example.BeerMessagingBase</baseClassFQN>
      </baseClassMapping>
      <baseClassMapping>
        <contractPackageRegex>.*rest.*</contractPackageRegex>
        <baseClassFQN>com.example.BeerRestBase</baseClassFQN>
      </baseClassMapping>
    </baseClassMappings>
    <basePackageForTests>com.example</basePackageForTests>
  </configuration>
  <executions>
    <execution>
      <phase>package</phase>
      <goals>
        <!-- By default we will not push the stubs back to SCM,
        you have to explicitly add it as a goal -->
        <goal>pushStubsToScm</goal>
      </goals>
      <configuration>
        <!-- We want to pick contracts from a Git repository -->
        <contractsRepositoryUrl>git://file://${env.ROOT}/target/contract_empty_git/
        </contractsRepositoryUrl>
        <!-- Example of URL via git protocol -->
        <!--<contractsRepositoryUrl>git://git@github.com:spring-cloud-
samples/spring-cloud-contract-samples.git</contractsRepositoryUrl>-->
        <!-- Example of URL via http protocol -->
        <!--<contractsRepositoryUrl>git://https://github.com/spring-cloud-
samples/spring-cloud-contract-samples.git</contractsRepositoryUrl>-->
        <!-- We reuse the contract dependency section to set up the path
to the folder that contains the contract definitions. In our case
the
        path will be /groupId/artifactId/version/contracts -->
        <contractDependency>
          <groupId>${project.groupId}</groupId>
          <artifactId>${project.artifactId}</artifactId>
          <version>${project.version}</version>
        </contractDependency>
        <!-- The mode can't be classpath -->
        <contractsMode>LOCAL</contractsMode>
      </configuration>
    </execution>
  </executions>
</plugin>

```

```
</execution>
</executions>
</plugin>
```

gradle

```
contracts {
    // Base package for generated tests
    basePackageForTests = "com.example"
    baseClassMappings {
        baseClassMapping(".*messaging.*", "com.example.BeerMessagingBase")
        baseClassMapping(".*rest.*", "com.example.BeerRestBase")
    }
}

/*
In this scenario we want to publish stubs to SCM whenever
the `publish` task is executed
*/
publishStubsToScm {
    // We want to modify the default set up of the plugin when publish stubs to
    scm is called
    customize {
        // We want to pick contracts from a Git repository
        contractDependency {
            stringNotation = "${project.group}:${project.name}:${project.version}"
        }
        /*
        We reuse the contract dependency section to set up the path
        to the folder that contains the contract definitions. In our case the
        path will be /groupId/artifactId/version/contracts
        */
        contractRepository {
            repositoryUrl = "git://file://${new File(project.rootDir,
"../target")}/contract_empty_git/"
        }
        // The mode can't be classpath
        contractsMode = "LOCAL"
    }
}

publish.dependsOn("publishStubsToScm")
publishToMavenLocal.dependsOn("publishStubsToScm")
```

With such a setup:

- Contracts from the default `src/test/resources/contracts` directory are picked.
- Tests are generated from the contracts.

- Stubs are created from the contracts.
- Once the tests pass:
 - The git project is cloned to a temporary directory.
 - The stubs and contracts are committed in the cloned repository.
- Finally, a push is done to that repository's **origin**.

Keeping Contracts with the Producer and Stubs in an External Repository

You can also keep the contracts in the producer repository but keep the stubs in an external git repository. This is most useful when you want to use the base consumer-producer collaboration flow but cannot use an artifact repository to store the stubs.

To do so, use the usual producer setup and then add the `pushStubsToScm` goal and set `contractsRepositoryUrl` to the repository where you want to keep the stubs.

Consumer

On the consumer side, when passing the `repositoryRoot` parameter, either from the `@AutoConfigureStubRunner` annotation, the JUnit rule, JUnit 5 extension, or properties, you can pass the URL of the SCM repository, prefixed with the `git://` protocol. The following example shows how to do so:

```
@AutoConfigureStubRunner(  
    stubsMode="REMOTE",  
    repositoryRoot="git://https://github.com/spring-cloud-samples/spring-cloud-  
contract-nodejs-contracts-git.git",  
    ids="com.example:bookstore:0.0.1.RELEASE"  
)
```

With such a setup:

- The git project is cloned to a temporary directory.
- The SCM stub downloader goes to the `META-INF/groupId/artifactId/version/` folder to find stub definitions and contracts. For example, for `com.example:foo:1.0.0`, the path would be `META-INF/com.example/foo/1.0.0/`.
- Stub servers are started and fed with mappings.
- Messaging definitions are read and used in the messaging tests.

14.8.7. How Can I Use the Pact Broker?

When using `Pact`, you can use the `Pact Broker` to store and share Pact definitions. Starting from Spring Cloud Contract 2.0.0, you can fetch Pact files from the Pact Broker to generate tests and stubs.



Pact follows the consumer contract convention. That means that the consumer creates the Pact definitions first and then shares the files with the Producer. Those expectations are generated from the Consumer's code and can break the Producer if the expectations are not met.

How to Work with Pact

Spring Cloud Contract includes support for the [Pact](#) representation of contracts up until version 4. Instead of using the DSL, you can use Pact files. In this section, we show how to add Pact support for your project. Note, however, that not all functionality is supported. Starting with version 3, you can combine multiple matchers for the same element; you can use matchers for the body, headers, request and path; and you can use value generators. Spring Cloud Contract currently only supports multiple matchers that are combined by using the **AND** rule logic. Next to that, the request and path matchers are skipped during the conversion. When using a date, time, or datetime value generator with a given format, the given format is skipped and the ISO format is used.

Pact Converter

In order to properly support the Spring Cloud Contract way of doing messaging with Pact, you have to provide some additional meta data entries.

To define the destination to which a message gets sent, you have to set a `metaData` entry in the Pact file with the `sentTo` key equal to the destination to which a message is to be sent (for example, `"metaData": { "sentTo": "activemq:output" }`).

Pact Contract

Spring Cloud Contract can read the Pact JSON definition. You can place the file in the `src/test/resources/contracts` folder. Remember to put the `spring-cloud-contract-pact` dependency to your classpath. The following example shows such a Pact contract:

```
{
  "provider": {
    "name": "Provider"
  },
  "consumer": {
    "name": "Consumer"
  },
  "interactions": [
    {
      "description": "",
      "request": {
        "method": "PUT",
        "path": "/pactfraudcheck",
        "headers": {
          "Content-Type": "application/json"
        }
      },
      "body": {
```

```
    "clientId": "1234567890",
    "loanAmount": 99999
  },
  "generators": {
    "body": {
      "$.clientId": {
        "type": "Regex",
        "regex": "[0-9]{10}"
      }
    }
  },
  "matchingRules": {
    "header": {
      "Content-Type": {
        "matchers": [
          {
            "match": "regex",
            "regex": "application/json.*"
          }
        ],
        "combine": "AND"
      }
    },
    "body": {
      "$.clientId": {
        "matchers": [
          {
            "match": "regex",
            "regex": "[0-9]{10}"
          }
        ],
        "combine": "AND"
      }
    }
  }
},
"response": {
  "status": 200,
  "headers": {
    "Content-Type": "application/json"
  },
  "body": {
    "fraudCheckStatus": "FRAUD",
    "rejection.reason": "Amount too high"
  },
  "matchingRules": {
    "header": {
      "Content-Type": {
        "matchers": [
          {
            "match": "regex",
```


example shows a test and stub that came from this process:

test

```
@Test
public void validate_shouldMarkClientAsFraud() throws Exception {
    // given:
    MockMvcRequestSpecification request = given()
        .header("Content-Type", "application/vnd.fraud.v1+json")
        .body("{\"clientId\":\"1234567890\",\"loanAmount\":99999}");

    // when:
    ResponseOptions response = given().spec(request)
        .put("/fraudcheck");

    // then:
    assertThat(response.statusCode()).isEqualTo(200);
    assertThat(response.header("Content-
Type")).matches("application/vnd\\.fraud\\.v1\\.+json.*");
    // and:
    DocumentContext parsedJson =
    JsonPath.parse(response.getBody().asString());

    assertThatJson(parsedJson).field("[ 'rejectionReason' ]").isEqualTo("Amount too
high");
    // and:
    assertThat(parsedJson.read("$.fraudCheckStatus",
String.class)).matches("FRAUD");
}
```


stub

```
{
  "id" : "996ae5ae-6834-4db6-8fac-358ca187ab62",
  "uuid" : "996ae5ae-6834-4db6-8fac-358ca187ab62",
  "request" : {
    "url" : "/fraudcheck",
    "method" : "PUT",
    "headers" : {
      "Content-Type" : {
        "matches" : "application/vnd\\.fraud\\.v1\\+json.*"
      }
    },
    "bodyPatterns" : [ {
      "matchesJsonPath" : "$[?(@['loanAmount'] = 99999)]"
    }, {
      "matchesJsonPath" : "$[?(@.clientId =~ /^[0-9]{10})/]"
    } ]
  },
  "response" : {
    "status" : 200,
    "body" : "{\"fraudCheckStatus\":\"FRAUD\",\"rejectionReason\":\"Amount too high\"}",
    "headers" : {
      "Content-Type" : "application/vnd.fraud.v1+json;charset=UTF-8"
    },
    "transformers" : [ "response-template" ]
  },
}
```

Pact for Consumers

On the consumer side, you must add two additional dependencies to your project dependencies. One is the Spring Cloud Contract Pact support, and the other represents the current Pact version that you use. The following listing shows how to do so for both Maven and Gradle:

Maven

Gradle

Communicating with the Pact Broker

Whenever the `repositoryRoot` property starts with a Pact protocol (starts with `pact://`), the stub downloader tries to fetch the Pact contract definitions from the Pact Broker. Whatever is set after

`pact://` is parsed as the Pact Broker URL.

By setting environment variables, system properties, or properties set inside the plugin or contracts repository configuration, you can tweak the downloader's behavior. The following table describes the properties:

Table 6. Pact Stub Downloader properties

Name of a property	Default	Description
* <code>pactbroker.host</code> (plugin prop) * <code>stubrunner.properties.pactbroker.host</code> (system prop) * <code>STUBRUNNER_PROPERTIES_PACTBROKER_HOST</code> (env prop)	Host from URL passed to <code>repositoryRoot</code>	The URL of the Pact Broker.
* <code>pactbroker.port</code> (plugin prop) * <code>stubrunner.properties.pactbroker.port</code> (system prop) * <code>STUBRUNNER_PROPERTIES_PACTBROKER_PORT</code> (env prop)	Port from URL passed to <code>repositoryRoot</code>	The port of Pact Broker.
* <code>pactbroker.protocol</code> (plugin prop) * <code>stubrunner.properties.pactbroker.protocol</code> (system prop) * <code>STUBRUNNER_PROPERTIES_PACTBROKER_PROTOCOL</code> (env prop)	Protocol from URL passed to <code>repositoryRoot</code>	The protocol of Pact Broker.
* <code>pactbroker.tags</code> (plugin prop) * <code>stubrunner.properties.pactbroker.tags</code> (system prop) * <code>STUBRUNNER_PROPERTIES_PACTBROKER_TAGS</code> (env prop)	Version of the stub, or <code>latest</code> if version is <code>+</code>	The tags that should be used to fetch the stub.

<ul style="list-style-type: none"> * <code>pactbroker.auth.scheme</code> (plugin prop) * <code>stubrunner.properties.pactbroker.auth.scheme</code> (system prop) * <code>STUBRUNNER_PROPERTIES_PACTBROKER_AUTH_SCHEME</code> (env prop) 	Basic	The kind of authentication that should be used to connect to the Pact Broker.
<ul style="list-style-type: none"> * <code>pactbroker.auth.username</code> (plugin prop) * <code>stubrunner.properties.pactbroker.auth.username</code> (system prop) * <code>STUBRUNNER_PROPERTIES_PACTBROKER_AUTH_USERNAME</code> (env prop) 	The username passed to <code>contractsRepositoryUsername</code> (maven) or <code>contractRepository.username</code> (gradle)	The username to use when connecting to the Pact Broker.
<ul style="list-style-type: none"> * <code>pactbroker.auth.password</code> (plugin prop) * <code>stubrunner.properties.pactbroker.auth.password</code> (system prop) * <code>STUBRUNNER_PROPERTIES_PACTBROKER_AUTH_PASSWORD</code> (env prop) 	The password passed to <code>contractsRepositoryPassword</code> (maven) or <code>contractRepository.password</code> (gradle)	The password to use when connecting to the Pact Broker.
<ul style="list-style-type: none"> * <code>pactbroker.provider-name-with-group-id</code> (plugin prop) * <code>stubrunner.properties.pactbroker.provider-name-with-group-id</code> (system prop) * <code>STUBRUNNER_PROPERTIES_PACTBROKER_PROVIDER_NAME_WITH_GROUP_ID</code> (env prop) 	false	When <code>true</code> , the provider name is a combination of <code>groupId:artifactId</code> . If <code>false</code> , only <code>artifactId</code> is used.

Flow: Consumer Contract approach with Pact Broker | Consumer Side

The consumer uses the Pact framework to generate Pact files. The Pact files are sent to the Pact Broker. You can find an example of such a setup [here](#).

Flow: Consumer Contract Approach with Pact Broker on the Producer Side

For the producer to use the Pact files from the Pact Broker, we can reuse the same mechanism we use for external contracts. We route Spring Cloud Contract to use the Pact implementation with the URL that contains the `pact://` protocol. You can pass the URL to the Pact Broker. You can find an example of such a setup [here](#). The following listing shows the configuration details for both Maven and Gradle:

maven

```
<plugin>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-contract-maven-plugin</artifactId>
  <version>${spring-cloud-contract.version}</version>
  <extensions>>true</extensions>
  <configuration>
    <!-- Base class mappings etc. -->

    <!-- We want to pick contracts from a Git repository -->

    <contractsRepositoryUrl>pact://http://localhost:8085</contractsRepositoryUrl>

    <!-- We reuse the contract dependency section to set up the path
    to the folder that contains the contract definitions. In our case the
    path will be /groupId/artifactId/version/contracts -->
    <contractDependency>
      <groupId>${project.groupId}</groupId>
      <artifactId>${project.artifactId}</artifactId>
      <!-- When + is passed, a latest tag will be applied when fetching
      pacts -->
      <version>+</version>
    </contractDependency>

    <!-- The contracts mode can't be classpath -->
    <contractsMode>REMOTE</contractsMode>
  </configuration>
  <!-- Don't forget to add spring-cloud-contract-pact to the classpath! -->
  <dependencies>
    <dependency>
      <groupId>org.springframework.cloud</groupId>
      <artifactId>spring-cloud-contract-pact</artifactId>
      <version>${spring-cloud-contract.version}</version>
    </dependency>
  </dependencies>
</plugin>
```

gradle

```
buildscript {
    repositories {
        //...
    }

    dependencies {
        // ...
        // Don't forget to add spring-cloud-contract-pact to the classpath!
        classpath "org.springframework.cloud:spring-cloud-contract-
pact:${contractVersion}"
    }
}

contracts {
    // When + is passed, a latest tag will be applied when fetching pacts
    contractDependency {
        stringNotation = "${project.group}:${project.name}:+\"
    }
    contractRepository {
        repositoryUrl = "pact://http://localhost:8085\"
    }
    // The mode can't be classpath
    contractsMode = "REMOTE\"
    // Base class mappings etc.
}
```

With such a setup:

- Pact files are downloaded from the Pact Broker.
- Spring Cloud Contract converts the Pact files into tests and stubs.
- The JAR with the stubs gets automatically created, as usual.

Flow: Producer Contract approach with Pact on the Consumer Side

In the scenario where you do not want to do the consumer contract approach (for every single consumer, define the expectations) but you prefer to do producer contracts (the producer provides the contracts and publishes stubs), you can use Spring Cloud Contract with the Stub Runner option. You can find an example of such a setup [here](#).

Remember to add the Stub Runner and Spring Cloud Contract Pact modules as test dependencies.

The following listing shows the configuration details for both Maven and Gradle:

maven

```
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>org.springframework.cloud</groupId>
      <artifactId>spring-cloud-dependencies</artifactId>
      <version>${spring-cloud.version}</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
  </dependencies>
</dependencyManagement>

<!-- Don't forget to add spring-cloud-contract-pact to the classpath! -->
<dependencies>
  <!-- ... -->
  <dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-contract-stub-runner</artifactId>
    <scope>test</scope>
  </dependency>
  <dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-contract-pact</artifactId>
    <scope>test</scope>
  </dependency>
</dependencies>
```

gradle

```
dependencyManagement {
  imports {
    mavenBom "org.springframework.cloud:spring-cloud-
dependencies:${springCloudVersion}"
  }
}

dependencies {
  //...
  testCompile("org.springframework.cloud:spring-cloud-starter-contract-stub-
runner")
  // Don't forget to add spring-cloud-contract-pact to the classpath!
  testCompile("org.springframework.cloud:spring-cloud-contract-pact")
}
```

Next, you can pass the URL of the Pact Broker to `repositoryRoot`, prefixed with `pact://` protocol (for example, `pact://http://localhost:8085`), as the following example shows:

```
@RunWith(SpringRunner.class)
@SpringBootTest
@AutoConfigureStubRunner(stubsMode = StubRunnerProperties.StubsMode.REMOTE,
    ids = "com.example:beer-api-producer-pact",
    repositoryRoot = "pact://http://localhost:8085")
public class BeerControllerTest {
    //Inject the port of the running stub
    @StubRunnerPort("beer-api-producer-pact") int producerPort;
    //...
}
```

With such a setup:

- Pact files are downloaded from the Pact Broker.
- Spring Cloud Contract converts the Pact files into stub definitions.
- The stub servers are started and fed with stubs.

14.8.8. How Can I Debug the Request/Response Being Sent by the Generated Tests Client?

The generated tests all boil down to RestAssured in some form or fashion. RestAssured relies on the [Apache HttpClient](#). HttpClient has a facility called [wire logging](#), which logs the entire request and response to HttpClient. Spring Boot has a logging [common application property](#) for doing this sort of thing. To use it, add this to your application properties, as follows:

```
logging.level.org.apache.http.wire=DEBUG
```

14.8.9. How Can I Debug the Mapping, Request, or Response Being Sent by WireMock?

Starting from version [1.2.0](#), we turn on WireMock logging to [info](#) and set the WireMock notifier to being verbose. Now you can exactly know what request was received by the WireMock server and which matching response definition was picked.

To turn off this feature, set WireMock logging to [ERROR](#), as follows:

```
logging.level.com.github.tomakehurst.wiremock=ERROR
```

14.8.10. How Can I See What Got Registered in the HTTP Server Stub?

You can use the `mappingsOutputFolder` property on `@AutoConfigureStubRunner`, `StubRunnerRule`, or `StubRunnerExtension`` to dump all mappings per artifact ID. Also the port at which the given stub server was started is attached.

14.8.11. How Can I Reference Text from File?

In version 1.2.0, we added this ability. You can call a `file(...)` method in the DSL and provide a path relative to where the contract lies. If you use YAML, you can use the `bodyFromFile` property.

14.8.12. How Can I Generate Pact, YAML, or X files from Spring Cloud Contract Contracts?

Spring Cloud Contract comes with a `ToFileContractsTransformer` class that lets you dump contracts as files for the given `ContractConverter`. It contains a `static void main` method that lets you execute the transformer as an executable. It takes the following arguments:

- argument 1 : `FQN`: Fully qualified name of the `ContractConverter` (for example, `PactContractConverter`). **REQUIRED**.
- argument 2 : `path`: Path where the dumped files should be stored. **OPTIONAL**—defaults to `target/converted-contracts`.
- argument 3 : `path`: Path where the contracts should be searched for. **OPTIONAL**—defaults to `src/test/resources/contracts`.

After executing the transformer, the Spring Cloud Contract files are processed and, depending on the provided FQN of the `ContractTransformer`, the contracts are transformed to the required format and dumped to the provided folder.

The following example shows how to configure Pact integration for both Maven and Gradle:

maven

```
<plugin>
  <groupId>org.codehaus.mojo</groupId>
  <artifactId>exec-maven-plugin</artifactId>
  <version>1.6.0</version>
  <executions>
    <execution>
      <id>convert-dsl-to-pact</id>
      <phase>process-test-classes</phase>
      <configuration>
        <classpathScope>test</classpathScope>
        <mainClass>
          org.springframework.cloud.contract.verifier.util.ToFileContractsTransformer
        </mainClass>
        <arguments>
          <argument>
            org.springframework.cloud.contract.verifier.spec.pact.PactContractConverter
          </argument>
          <argument>${project.basedir}/target/pacts</argument>
          <argument>
            ${project.basedir}/src/test/resources/contracts
          </argument>
        </arguments>
      </configuration>
      <goals>
        <goal>java</goal>
      </goals>
    </execution>
  </executions>
</plugin>
```

gradle

```
task convertContracts(type: JavaExec) {
    main =
    "org.springframework.cloud.contract.verifier.util.ToFileContractsTransformer"
    classpath = sourceSets.test.compileClasspath

    args("org.springframework.cloud.contract.verifier.spec.pact.PactContractConverter"
    ,
        "${project.rootDir}/build/pacts",
        "${project.rootDir}/src/test/resources/contracts")
}

test.dependsOn("convertContracts")
```

14.8.13. How Can I Work with Transitive Dependencies?

The Spring Cloud Contract plugins add the tasks that create the stubs jar for you. One problem that arises is that, when reusing the stubs, you can mistakenly import all of that stub's dependencies. When building a Maven artifact, even though you have a couple of different jars, all of them share one pom, as the following listing shows:

```
|—— producer-0.0.1.BUILD-20160903.075506-1-stubs.jar
|—— producer-0.0.1.BUILD-20160903.075506-1-stubs.jar.sha1
|—— producer-0.0.1.BUILD-20160903.075655-2-stubs.jar
|—— producer-0.0.1.BUILD-20160903.075655-2-stubs.jar.sha1
|—— producer-0.0.1.BUILD-SNAPSHOT.jar
|—— producer-0.0.1.BUILD-SNAPSHOT.pom
|—— producer-0.0.1.BUILD-SNAPSHOT-stubs.jar
|—— ...
|—— ...
```

There are three possibilities of working with those dependencies so as not to have any issues with transitive dependencies:

- Mark all application dependencies as optional
- Create a separate artifactid for the stubs
- Exclude dependencies on the consumer side

How Can I Mark All Application Dependencies as Optional?

If, in the `producer` application, you mark all of your dependencies as optional, when you include the `producer` stubs in another application (or when that dependency gets downloaded by Stub Runner) then, since all of the dependencies are optional, they do not get downloaded.

How can I Create a Separate artifactid for the Stubs?

If you create a separate `artifactid`, you can set it up in whatever way you wish. For example, you might decide to have no dependencies at all.

How can I Exclude Dependencies on the Consumer Side?

As a consumer, if you add the stub dependency to your classpath, you can explicitly exclude the unwanted dependencies.

14.8.14. How can I Generate Spring REST Docs Snippets from the Contracts?

When you want to include the requests and responses of your API by using Spring REST Docs, you only need to make some minor changes to your setup if you are using `MockMvc` and `RestAssuredMockMvc`. To do so, include the following dependencies (if you have not already done so):

maven

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-contract-verifier</artifactId>
  <scope>test</scope>
</dependency>
<dependency>
  <groupId>org.springframework.restdocs</groupId>
  <artifactId>spring-restdocs-mockmvc</artifactId>
  <optional>true</optional>
</dependency>
```

gradle

```
testImplementation 'org.springframework.cloud:spring-cloud-starter-contract-
verifier'
testImplementation 'org.springframework.restdocs:spring-restdocs-mockmvc'
```

Next, you need to make some changes to your base class. The following examples use [WebApplicationContext](#) and the standalone option with RestAssured:

WebApplicationContext

```
package com.example.fraud;

import io.restassured.module.mockmvc.RestAssuredMockMvc;
import org.junit.Before;
import org.junit.Rule;
import org.junit.rules.TestName;
import org.junit.runner.RunWith;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.restdocs.JUnitRestDocumentation;
import org.springframework.test.context.junit4.SpringRunner;
import org.springframework.test.web.servlet.setup.MockMvcBuilders;
import org.springframework.web.context.WebApplicationContext;

import static
org.springframework.restdocs.mockmvc.MockMvcRestDocumentation.document;
import static
org.springframework.restdocs.mockmvc.MockMvcRestDocumentation.documentationConfigu
ration;

@RunWith(SpringRunner.class)
```

```

@SpringBootTest(classes = Application.class)
public abstract class FraudBaseWithWebAppSetup {

    private static final String OUTPUT = "target/generated-snippets";

    @Rule
    public JUnitRestDocumentation restDocumentation = new
JUnitRestDocumentation(OUTPUT);

    @Rule
    public TestName testName = new TestName();

    @Autowired
    private WebApplicationContext context;

    @Before
    public void setup() {

RestAssuredMockMvc.mockMvc(MockMvcBuilders.webAppContextSetup(this.context)
        .apply(documentationConfiguration(this.restDocumentation))
        .alwaysDo(document(
            getClass().getSimpleName() + "_" +
testName.getMethodName()))
        .build());
    }

    protected void assertThatRejectionReasonIsNull(Object rejectionReason) {
        assert rejectionReason == null;
    }
}

```

```
package com.example.fraud;

import io.restassured.module.mockmvc.RestAssuredMockMvc;
import org.junit.Before;
import org.junit.Rule;
import org.junit.rules.TestName;

import org.springframework.restdocs.JUnitRestDocumentation;
import org.springframework.test.web.servlet.setup.MockMvcBuilders;

import static
org.springframework.restdocs.mockmvc.MockMvcRestDocumentation.document;
import static
org.springframework.restdocs.mockmvc.MockMvcRestDocumentation.documentationConfigu
ration;

public abstract class FraudBaseWithStandaloneSetup {

    private static final String OUTPUT = "target/generated-snippets";

    @Rule
    public JUnitRestDocumentation restDocumentation = new
JUnitRestDocumentation(OUTPUT);

    @Rule
    public TestName testName = new TestName();

    @Before
    public void setup() {
        RestAssuredMockMvc.standaloneSetup(MockMvcBuilders
            .standaloneSetup(new FraudDetectionController())
            .apply(documentationConfiguration(this.restDocumentation))
            .alwaysDo(document(
                getClass().getSimpleName() + "_" +
testName.getMethodName())));
    }
}
```



You need not specify the output directory for the generated snippets (since version 1.2.0.RELEASE of Spring REST Docs).

14.8.15. How can I Use Stubs from a Location

If you want to fetch contracts or stubs from a given location without cloning a repo or fetching a JAR, just use the `stubs://` protocol when providing the repository root argument for Stub Runner or the Spring Cloud Contract plugin. You can read more about this in [this section](#) of the

documentation.

14.8.16. How can I Generate Stubs at Runtime

If you want to generate stubs at runtime for contracts, it's enough to switch the `generateStubs` property in the `@AutoConfigureStubRunner` annotation, or call the `withGenerateStubs(true)` method on the JUnit Rule or Extension. You can read more about this in [this section](#) of the documentation.

14.8.17. How can I Make The Build Pass if There Are No Contracts or Stubs

If you want Stub Runner not to fail if no stubs were found, it's enough to switch the `generateStubs` property in the `@AutoConfigureStubRunner` annotation, or call the `withFailOnNoStubs(false)` method on the JUnit Rule or Extension. You can read more about this in [this section](#) of the documentation.

If you want the plugins not to fail the build when no contracts were found, you can set the `failOnNoStubs` flag in Maven or call the `contractRepository { failOnNoStubs(false) }` Closure in Gradle.

14.8.18. How can I Mark that a Contract Is in Progress

If a contract is in progress, it means that the on the producer side tests will not be generated, but the stub will be. You can read more about this in [this section](#) of the documentation.

In a CI build, before going to production, you would like to ensure that no in progress contracts are there on the classpath. That's because you may lead to false positives. That's why, by default, in the Spring Cloud Contract plugin, we set the value of `failOnInProgress` to `true`. If you want to allow such contracts when tests are to be generated, just set the flag to `false`.

Appendix A: Common application properties

Various properties can be specified inside your `application.properties` file, inside your `application.yml` file, or as command line switches. This appendix provides a list of common Spring Cloud Contract properties and references to the underlying classes that consume them.



Property contributions can come from additional jar files on your classpath, so you should not consider this an exhaustive list. Also, you can define your own properties.

Default application properties

Name	Default	Description
<code>stubrunner.amqp.enabled</code>	false	Whether to enable support for Stub Runner and AMQP.
<code>stubrunner.amqp.mockConnection</code>	true	Whether to enable support for Stub Runner and AMQP mocked connection factory.

Name	Default	Description
stubrunner.classifier	stubs	The classifier to use by default in ivy co-ordinates for a stub.
stubrunner.cloud.consul.enabled	true	Whether to enable stubs registration in Consul.
stubrunner.cloud.delegate.enabled	true	Whether to enable DiscoveryClient's Stub Runner implementation.
stubrunner.cloud.enabled	true	Whether to enable Spring Cloud support for Stub Runner.
stubrunner.cloud.eureka.enabled	true	Whether to enable stubs registration in Eureka.
stubrunner.cloud.loadbalancer.enabled	true	Whether to enable Stub Runner's Spring Cloud Load Balancer integration.
stubrunner.cloud.stubbed.discovery.enabled	true	Whether Service Discovery should be stubbed for Stub Runner. If set to false, stubs will get registered in real service discovery.
stubrunner.cloud.zookeeper.enabled	true	Whether to enable stubs registration in Zookeeper.
stubrunner.consumer-name		You can override the default <code>spring.application.name</code> of this field by setting a value to this parameter.
stubrunner.delete-stubs-after-test	true	If set to <code>false</code> will NOT delete stubs from a temporary folder after running tests.
stubrunner.fail-on-no-stubs	true	When enabled, this flag will tell stub runner to throw an exception when no stubs / contracts were found.
stubrunner.generate-stubs	false	When enabled, this flag will tell stub runner to not load the generated stubs, but convert the found contracts at runtime to a stub format and run those stubs.
stubrunner.http-server-stub-configurer		Configuration for an HTTP server stub.

Name	Default	Description
stubrunner.ids	[]	The ids of the stubs to run in "ivy" notation ([groupId]:artifactId:[version]:[classifier][:port]). {@code groupId}, {@code classifier}, {@code version} and {@code port} can be optional.
stubrunner.ids-to-service-ids		Mapping of Ivy notation based ids to serviceIds inside your application. Example "a:b" → "myService" "artifactId" → "myOtherService"
stubrunner.integration.enabled	true	Whether to enable Stub Runner integration with Spring Integration.
stubrunner.jms.enabled	true	Whether to enable Stub Runner integration with Spring JMS.
stubrunner.kafka.enabled	true	Whether to enable Stub Runner integration with Spring Kafka.
stubrunner.kafka.initializer.enabled	true	Whether to allow Stub Runner to take care of polling for messages instead of the KafkaStubMessages component. The latter should be used only on the producer side.
stubrunner.mappings-output-folder		Dumps the mappings of each HTTP server to the selected folder.
stubrunner.max-port	15000	Max value of a port for the automatically started WireMock server.
stubrunner.min-port	10000	Min value of a port for the automatically started WireMock server.
stubrunner.password		Repository password.
stubrunner.properties		Map of properties that can be passed to custom {@link org.springframework.cloud.contract.stubrunner.StubDownloaderBuilder}.
stubrunner.proxy-host		Repository proxy host.

Name	Default	Description
stubrunner.proxy-port		Repository proxy port.
stubrunner.server-id		null
stubrunner.stream.enabled	true	Whether to enable Stub Runner integration with Spring Cloud Stream.
stubrunner.stubs-mode		Pick where the stubs should come from.
stubrunner.stubs-per-consumer	false	Should only stubs for this particular consumer get registered in HTTP server stub.
stubrunner.username		Repository username.
wiremock.placeholders.enabled	true	Flag to indicate that http URLs in generated wiremock stubs should be filtered to add or resolve a placeholder for a dynamic port.
wiremock.reset-mappings-after-each-test	false	null
wiremock.rest-template-ssl-enabled	false	null
wiremock.server.files	[]	null
wiremock.server.https-port	-1	null
wiremock.server.https-port-dynamic	false	null
wiremock.server.port	8080	null
wiremock.server.port-dynamic	false	null
wiremock.server.stubs	[]	null

Additional application properties



The following properties can be passed as a system property (e.g. `stubrunner.properties.git.branch`) or via an environment variable (e.g. `STUBRUNNER_PROPERTIES_GIT_BRANCH`) or as a property inside stub runner's annotation or a JUnit Rule / Extension. In the latter case you can pass `git.branch` property name instead of the `stubrunner.properties.git.branch` one.

Table 7. Stubrunner Properties Options

Name	Default	Description
stubrunner.properties.pactbroker.provider-name-with-group-id	false	When using the Pact Broker based approach, you can automatically group id to the provider name.
stubrunner.properties.git.branch		When using the SCM based approach, you can customize the branch name to check out.
stubrunner.properties.git.commit-message	Updating project [\$project] with stubs	When using the SCM based approach, you can customize the commit message for created stubs. The \$project text will be replaced with the project name.
stubrunner.properties.git.no-of-attempts	10	When using the SCM based approach, you can customize number of retries to push the stubs to Git.
stubrunner.properties.git.username		When using the SCM based approach, you can pass the username to connect to the Git repository.
stubrunner.properties.git.password		When using the SCM based approach, you can pass the password to connect to the Git repository.
stubrunner.properties.git.wait-between-attempts	1000	When using the SCM based approach, you can customize waiting time in ms between trying to push the stubs to Git.
stubrunner.properties.stubs.find-producer	false	When using the Stubs protocol, you can toggle this flag to search for contracts via the group id / artifact id instead of taking the stubs directly from the provided folder.

Chapter 15. Spring Cloud Vault

© 2016-2020 The original authors.



Copies of this document may be made for your own use and for distribution to others, provided that you do not charge any fee for such copies and further provided that each copy contains this Copyright Notice, whether distributed in print or electronically.

Spring Cloud Vault Config provides client-side support for externalized configuration in a distributed system. With [HashiCorp's Vault](#) you have a central place to manage external secret properties for applications across all environments. Vault can manage static and dynamic secrets such as username/password for remote applications/resources and provide credentials for external services such as MySQL, PostgreSQL, Apache Cassandra, MongoDB, Consul, AWS and more.

15.1. Quick Start

Prerequisites

To get started with Vault and this guide you need a *NIX-like operating systems that provides:

- `wget`, `openssl` and `unzip`
- at least Java 8 and a properly configured `JAVA_HOME` environment variable



This guide explains Vault setup from a Spring Cloud Vault perspective for integration testing. You can find a getting started guide directly on the Vault project site: learn.hashicorp.com/vault

Install Vault

```
$ wget
https://releases.hashicorp.com/vault/${vault_version}/vault_${vault_version}_${platform}.zip
$ unzip vault_${vault_version}_${platform}.zip
```



These steps can be achieved by downloading and running `install_vault.sh`.

Create SSL certificates for Vault

Next, you're required to generate a set of certificates:

- Root CA
- Vault Certificate (decrypted key `work/ca/private/localhost.decrypted.key.pem` and certificate `work/ca/certs/localhost.cert.pem`)

Make sure to import the Root Certificate into a Java-compliant truststore.

The easiest way to achieve this is by using OpenSSL.



`create_certificates.sh` creates certificates in `work/ca` and a JKS truststore `work/keystore.jks`. If you want to run Spring Cloud Vault using this quickstart guide you need to configure the truststore the `spring.cloud.vault.ssl.trust-store` property to `file:work/keystore.jks`.

Start Vault server

Next create a config file along the lines of:

```
backend "inmem" {  
  }  
  
listener "tcp" {  
  address = "0.0.0.0:8200"  
  tls_cert_file = "work/ca/certs/localhost.cert.pem"  
  tls_key_file = "work/ca/private/localhost.decrypted.key.pem"  
}  
  
disable_mlock = true
```



You can find an example config file at [vault.conf](#).

```
$ vault server -config=vault.conf
```

Vault is started listening on `0.0.0.0:8200` using the `inmem` storage and `https`. Vault is sealed and not initialized when starting up.



If you want to run tests, leave Vault uninitialized. The tests will initialize Vault and create a root token `00000000-0000-0000-0000-000000000000`.

If you want to use Vault for your application or give it a try then you need to initialize it first.

```
$ export VAULT_ADDR="https://localhost:8200"  
$ export VAULT_SKIP_VERIFY=true # Don't do this for production  
$ vault init
```

You should see something like:

```
Key 1: 7149c6a2e16b8833f6eb1e76df03e47f6113a3288b3093faf5033d44f0e70fe701
Key 2: 901c534c7988c18c20435a85213c683bdcf0efcd82e38e2893779f152978c18c02
Key 3: 03ff3948575b1165a20c20ee7c3e6edf04f4cdbe0e82dbff5be49c63f98bc03a03
Key 4: 216ae5cc3ddaf93ceb8e1d15bb9fc3176653f5b738f5f3d1ee00cd7dccbe926e04
Key 5: b2898fc8130929d569c1677ee69dc5f3be57d7c4b494a6062693ce0b1c4d93d805
Initial Root Token: 19aefa97-cccc-bbbb-aaaa-225940e63d76
```

Vault initialized with 5 keys and a key threshold of 3. Please securely distribute the above keys. When the Vault is re-sealed, restarted, or stopped, you must provide at least 3 of these keys to unseal it again.

Vault does not store the master key. Without at least 3 keys, your Vault will remain permanently sealed.

Vault will initialize and return a set of unsealing keys and the root token. Pick 3 keys and unseal Vault. Store the Vault token in the `VAULT_TOKEN` environment variable.

```
$ vault unseal (Key 1)
$ vault unseal (Key 2)
$ vault unseal (Key 3)
$ export VAULT_TOKEN=(Root token)
# Required to run Spring Cloud Vault tests after manual initialization
$ vault token-create -id="00000000-0000-0000-0000-000000000000" -policy="root"
```

Spring Cloud Vault accesses different resources. By default, the secret backend is enabled which accesses secret config settings via JSON endpoints.

The HTTP service has resources in the form:

```
/secret/{application}/{profile}
/secret/{application}
/secret/{defaultContext}/{profile}
/secret/{defaultContext}
```

where the "application" is injected as the `spring.application.name` in the `SpringApplication` (i.e. what is normally "application" in a regular Spring Boot app), "profile" is an active profile (or comma-separated list of properties). Properties retrieved from Vault will be used "as-is" without further prefixing of the property names.

15.2. Client Side Usage

To use these features in an application, just build it as a Spring Boot application that depends on `spring-cloud-vault-config` (e.g. see the test cases). Example Maven configuration:

Example 8. pom.xml

```
<parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-parent</artifactId>
  <version>2.0.0.RELEASE</version>
  <relativePath /> <!-- lookup parent from repository -->
</parent>

<dependencies>
  <dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-vault-config</artifactId>
    <version>3.0.0-M2</version>
  </dependency>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
    <scope>test</scope>
  </dependency>
</dependencies>

<build>
  <plugins>
    <plugin>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-maven-plugin</artifactId>
    </plugin>
  </plugins>
</build>

<!-- repositories also needed for snapshots and milestones -->
```

Then you can create a standard Spring Boot application, like this simple HTTP server:

```

@SpringBootApplication
@RestController
public class Application {

    @RequestMapping("/")
    public String home() {
        return "Hello World!";
    }

    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }
}

```

When it runs it will pick up the external configuration from the default local Vault server on port **8200** if it is running. To modify the startup behavior you can change the location of the Vault server using **bootstrap.properties** (like **application.properties** but for the bootstrap phase of an application context), e.g.

Example 9. bootstrap.yml

```

spring.cloud.vault:
  host: localhost
  port: 8200
  scheme: https
  uri: https://localhost:8200
  connection-timeout: 5000
  read-timeout: 15000
  config:
    order: -10

```

- **host** sets the hostname of the Vault host. The host name will be used for SSL certificate validation
- **port** sets the Vault port
- **scheme** setting the scheme to **http** will use plain HTTP. Supported schemes are **http** and **https**.
- **uri** configure the Vault endpoint with an URI. Takes precedence over host/port/scheme configuration
- **connection-timeout** sets the connection timeout in milliseconds
- **read-timeout** sets the read timeout in milliseconds
- **config.order** sets the order for the property source

Enabling further integrations requires additional dependencies and configuration. Depending on how you have set up Vault you might need additional configuration like [SSL](#) and [authentication](#).

If the application imports the `spring-boot-starter-actuator` project, the status of the vault server will be available via the `/health` endpoint.

The vault health indicator can be enabled or disabled through the property `management.health.vault.enabled` (default to `true`).

15.2.1. Authentication

Vault requires an [authentication mechanism](#) to [authorize client requests](#).

Spring Cloud Vault supports multiple [authentication mechanisms](#) to authenticate applications with Vault.

For a quickstart, use the root token printed by the [Vault initialization](#).

Example 10. bootstrap.yml

```
spring.cloud.vault:  
  token: 19aefa97-cccc-bbbb-aaaa-225940e63d76
```



Consider carefully your security requirements. Static token authentication is fine if you want quickly get started with Vault, but a static token is not protected any further. Any disclosure to unintended parties allows Vault use with the associated token roles.

15.3. Authentication methods

Different organizations have different requirements for security and authentication. Vault reflects that need by shipping multiple authentication methods. Spring Cloud Vault supports token and AppId authentication.

15.3.1. Token authentication

Tokens are the core method for authentication within Vault. Token authentication requires a static token to be provided using the [Bootstrap Application Context](#).



Token authentication is the default authentication method. If a token is disclosed an unintended party gains access to Vault and can access secrets for the intended client.

Example 11. bootstrap.yml

```
spring.cloud.vault:  
  authentication: TOKEN  
  token: 00000000-0000-0000-0000-000000000000
```

- `authentication` setting this value to `TOKEN` selects the Token authentication method
- `token` sets the static token to use

See also: [Vault Documentation: Tokens](#)

15.3.2. Vault Agent authentication

Vault ships a sidecar utility with Vault Agent since version 0.11.0. Vault Agent implements the functionality of Spring Vault's `SessionManager` with its Auto-Auth feature. Applications can reuse cached session credentials by relying on Vault Agent running on `localhost`. Spring Vault can send requests without the `X-Vault-Token` header. Disable Spring Vault's authentication infrastructure to disable client authentication and session management.

Example 12. bootstrap.yml

```
spring.cloud.vault:  
  authentication: NONE
```

- `authentication` setting this value to `NONE` disables `ClientAuthentication` and `SessionManager`.

See also: [Vault Documentation: Agent](#)

15.3.3. AppId authentication

Vault supports `AppId` authentication that consists of two hard to guess tokens. The `AppId` defaults to `spring.application.name` that is statically configured. The second token is the `UserId` which is a part determined by the application, usually related to the runtime environment. IP address, Mac address or a Docker container name are good examples. Spring Cloud Vault Config supports IP address, Mac address and static `UserId`'s (e.g. supplied via System properties). The IP and Mac address are represented as Hex-encoded SHA256 hash.

IP address-based `UserId`'s use the local host's IP address.

Example 13. bootstrap.yml using SHA256 IP-Address UserId's

```
spring.cloud.vault:  
  authentication: APPID  
  app-id:  
    user-id: IP_ADDRESS
```

- **authentication** setting this value to **APPID** selects the AppId authentication method
- **app-id-path** sets the path of the AppId mount to use
- **user-id** sets the UserId method. Possible values are **IP_ADDRESS**, **MAC_ADDRESS** or a class name implementing a custom **AppIdUserIdMechanism**

The corresponding command to generate the IP address UserId from a command line is:

```
$ echo -n 192.168.99.1 | sha256sum
```



Including the line break of **echo** leads to a different hash value so make sure to include the **-n** flag.

Mac address-based UserId's obtain their network device from the localhost-bound device. The configuration also allows specifying a **network-interface** hint to pick the right device. The value of **network-interface** is optional and can be either an interface name or interface index (0-based).

Example 14. bootstrap.yml using SHA256 Mac-Address UserId's

```
spring.cloud.vault:  
  authentication: APPID  
  app-id:  
    user-id: MAC_ADDRESS  
    network-interface: eth0
```

- **network-interface** sets network interface to obtain the physical address

The corresponding command to generate the IP address UserId from a command line is:

```
$ echo -n 0AFEDE1234AC | sha256sum
```



The Mac address is specified uppercase and without colons. Including the line break of **echo** leads to a different hash value so make sure to include the **-n** flag.

Custom UserId

The UserId generation is an open mechanism. You can set `spring.cloud.vault.app-id.user-id` to any string and the configured value will be used as static UserId.

A more advanced approach lets you set `spring.cloud.vault.app-id.user-id` to a classname. This class must be on your classpath and must implement the `org.springframework.cloud.vault.AppIdUserIdMechanism` interface and the `createUserId` method. Spring Cloud Vault will obtain the UserId by calling `createUserId` each time it authenticates using AppId to obtain a token.

Example 15. bootstrap.yml

```
spring.cloud.vault:
  authentication: APPID
  app-id:
    user-id: com.example.MyUserIdMechanism
```

Example 16. MyUserIdMechanism.java

```
public class MyUserIdMechanism implements AppIdUserIdMechanism {

    @Override
    public String createUserId() {
        String userId = ...
        return userId;
    }
}
```

See also: [Vault Documentation: Using the App ID auth backend](#)

15.3.4. AppRole authentication

[AppRole](#) is intended for machine authentication, like the deprecated (since Vault 0.6.1) [AppId authentication](#). AppRole authentication consists of two hard to guess (secret) tokens: RoleId and SecretId.

Spring Vault supports various AppRole scenarios (push/pull mode and wrapped).

RoleId and optionally SecretId must be provided by configuration, Spring Vault will not look up these or create a custom SecretId.

Example 17. bootstrap.yml with AppRole authentication properties

```
spring.cloud.vault:
  authentication: APPROLE
  app-role:
    role-id: bde2076b-cccb-3cf0-d57e-bca7b1e83a52
```

The following scenarios are supported along the required configuration details:

Table 8. Configuration

Method	RoleId	SecretId	RoleName	Token
Provided RoleId/SecretId	Provided	Provided		
Provided RoleId without SecretId	Provided			
Provided RoleId, Pull SecretId	Provided	Provided	Provided	Provided
Pull RoleId, provided SecretId		Provided	Provided	Provided
Full Pull Mode			Provided	Provided
Wrapped				Provided
Wrapped RoleId, provided SecretId	Provided			Provided
Provided RoleId, wrapped SecretId		Provided		Provided

Table 9. Pull/Push/Wrapped Matrix

RoleId	SecretId	Supported
Provided	Provided	
Provided	Pull	
Provided	Wrapped	
Provided	Absent	
Pull	Provided	
Pull	Pull	
Pull	Wrapped	
Pull	Absent	
Wrapped	Provided	
Wrapped	Pull	

Wrapped	Wrapped	
Wrapped	Absent	



You can use still all combinations of push/pull/wrapped modes by providing a configured `AppRoleAuthentication` bean within the bootstrap context. Spring Cloud Vault cannot derive all possible AppRole combinations from the configuration properties.



AppRole authentication is limited to simple pull mode using reactive infrastructure. Full pull mode is not yet supported. Using Spring Cloud Vault with the Spring WebFlux stack enables Vault's reactive auto-configuration which can be disabled by setting `spring.cloud.vault.reactive.enabled=false`.

Example 18. bootstrap.yml with all AppRole authentication properties

```
spring.cloud.vault:
  authentication: APPROLE
  app-role:
    role-id: bde2076b-cccb-3cf0-d57e-bca7b1e83a52
    secret-id: 1696536f-1976-73b1-b241-0b4213908d39
    role: my-role
    app-role-path: approle
```

- `role-id` sets the RoleId.
- `secret-id` sets the SecretId. SecretId can be omitted if AppRole is configured without requiring SecretId (See `bind_secret_id`).
- `role`: sets the AppRole name for pull mode.
- `app-role-path` sets the path of the approle authentication mount to use.

See also: [Vault Documentation: Using the AppRole auth backend](#)

15.3.5. AWS-EC2 authentication

The `aws-ec2` auth backend provides a secure introduction mechanism for AWS EC2 instances, allowing automated retrieval of a Vault token. Unlike most Vault authentication backends, this backend does not require first-deploying, or provisioning security-sensitive credentials (tokens, username/password, client certificates, etc.). Instead, it treats AWS as a Trusted Third Party and uses the cryptographically signed dynamic metadata information that uniquely represents each EC2 instance.

Example 19. bootstrap.yml using AWS-EC2 Authentication

```
spring.cloud.vault:  
  authentication: AWS_EC2
```

AWS-EC2 authentication enables nonce by default to follow the Trust On First Use (TOFU) principle. Any unintended party that gains access to the PKCS#7 identity metadata can authenticate against Vault.

During the first login, Spring Cloud Vault generates a nonce that is stored in the auth backend aside the instance Id. Re-authentication requires the same nonce to be sent. Any other party does not have the nonce and can raise an alert in Vault for further investigation.

The nonce is kept in memory and is lost during application restart. You can configure a static nonce with `spring.cloud.vault.aws-ec2.nonce`.

AWS-EC2 authentication roles are optional and default to the AMI. You can configure the authentication role by setting the `spring.cloud.vault.aws-ec2.role` property.

Example 20. bootstrap.yml with configured role

```
spring.cloud.vault:  
  authentication: AWS_EC2  
  aws-ec2:  
    role: application-server
```

Example 21. bootstrap.yml with all AWS EC2 authentication properties

```
spring.cloud.vault:  
  authentication: AWS_EC2  
  aws-ec2:  
    role: application-server  
    aws-ec2-path: aws-ec2  
    identity-document: http://...  
    nonce: my-static-nonce
```

- `authentication` setting this value to `AWS_EC2` selects the AWS EC2 authentication method
- `role` sets the name of the role against which the login is being attempted.
- `aws-ec2-path` sets the path of the AWS EC2 mount to use
- `identity-document` sets URL of the PKCS#7 AWS EC2 identity document
- `nonce` used for AWS-EC2 authentication. An empty nonce defaults to nonce generation

See also: [Vault Documentation: Using the aws auth backend](#)

15.3.6. AWS-IAM authentication

The `aws` backend provides a secure authentication mechanism for AWS IAM roles, allowing the automatic authentication with vault based on the current IAM role of the running application. Unlike most Vault authentication backends, this backend does not require first-deploying, or provisioning security-sensitive credentials (tokens, username/password, client certificates, etc.). Instead, it treats AWS as a Trusted Third Party and uses the 4 pieces of information signed by the caller with their IAM credentials to verify that the caller is indeed using that IAM role.

The current IAM role the application is running in is automatically calculated. If you are running your application on AWS ECS then the application will use the IAM role assigned to the ECS task of the running container. If you are running your application naked on top of an EC2 instance then the IAM role used will be the one assigned to the EC2 instance.

When using the AWS-IAM authentication you must create a role in Vault and assign it to your IAM role. An empty `role` defaults to the friendly name the current IAM role.

Example 22. bootstrap.yml with required AWS-IAM Authentication properties

```
spring.cloud.vault:  
  authentication: AWS_IAM
```

Example 23. bootstrap.yml with all AWS-IAM Authentication properties

```
spring.cloud.vault:  
  authentication: AWS_IAM  
  aws-iam:  
    role: my-dev-role  
    aws-path: aws  
    server-id: some.server.name  
    endpoint-uri: https://sts.eu-central-1.amazonaws.com
```

- `role` sets the name of the role against which the login is being attempted. This should be bound to your IAM role. If one is not supplied then the friendly name of the current IAM user will be used as the vault role.
- `aws-path` sets the path of the AWS mount to use
- `server-id` sets the value to use for the `X-Vault-AWS-IAM-Server-ID` header preventing certain types of replay attacks.
- `endpoint-uri` sets the value to use for the AWS STS API used for the `iam_request_url` parameter.

AWS-IAM requires the AWS Java SDK dependency (`com.amazonaws:aws-java-sdk-core`) as the authentication implementation uses AWS SDK types for credentials and request signing.

See also: [Vault Documentation: Using the aws auth backend](#)

15.3.7. Azure MSI authentication

The `azure` auth backend provides a secure introduction mechanism for Azure VM instances, allowing automated retrieval of a Vault token. Unlike most Vault authentication backends, this backend does not require first-deploying, or provisioning security-sensitive credentials (tokens, username/password, client certificates, etc.). Instead, it treats Azure as a Trusted Third Party and uses the managed service identity and instance metadata information that can be bound to a VM instance.

Example 24. bootstrap.yml with required Azure Authentication properties

```
spring.cloud.vault:  
  authentication: AZURE_MSI  
  azure-msi:  
    role: my-dev-role
```

Example 25. bootstrap.yml with all Azure Authentication properties

```
spring.cloud.vault:  
  authentication: AZURE_MSI  
  azure-msi:  
    role: my-dev-role  
    azure-path: azure
```

- `role` sets the name of the role against which the login is being attempted.
- `azure-path` sets the path of the Azure mount to use

Azure MSI authentication fetches environmental details about the virtual machine (subscription Id, resource group, VM name) from the instance metadata service.

See also: [Vault Documentation: Using the azure auth backend](#)

15.3.8. TLS certificate authentication

The `cert` auth backend allows authentication using SSL/TLS client certificates that are either signed by a CA or self-signed.

To enable `cert` authentication you need to:

1. Use SSL, see [Vault Client SSL configuration](#)
2. Configure a Java `Keystore` that contains the client certificate and the private key
3. Set the `spring.cloud.vault.authentication` to `CERT`

Example 26. bootstrap.yml

```
spring.cloud.vault:  
  authentication: CERT  
  ssl:  
    key-store: classpath:keystore.jks  
    key-store-password: changeit  
    cert-auth-path: cert
```

See also: [Vault Documentation: Using the Cert auth backend](#)

15.3.9. Cubbyhole authentication

Cubbyhole authentication uses Vault primitives to provide a secured authentication workflow. Cubbyhole authentication uses tokens as primary login method. An ephemeral token is used to obtain a second, login VaultToken from Vault's Cubbyhole secret backend. The login token is usually longer-lived and used to interact with Vault. The login token will be retrieved from a wrapped response stored at `/cubbyhole/response`.

Creating a wrapped token



Response Wrapping for token creation requires Vault 0.6.0 or higher.

Example 27. Creating and storing tokens

```
$ vault token-create -wrap-ttl="10m"  
Key                               Value  
---                               -  
wrapping_token:                   397ccb93-ff6c-b17b-9389-380b01ca2645  
wrapping_token_ttl:                0h10m0s  
wrapping_token_creation_time:      2016-09-18 20:29:48.652957077 +0200 CEST  
wrapped_accessor:                  46b6aebb-187f-932a-26d7-4f3d86a68319
```

Example 28. bootstrap.yml

```
spring.cloud.vault:  
  authentication: CUBBYHOLE  
  token: 397ccb93-ff6c-b17b-9389-380b01ca2645
```

See also:

- [Vault Documentation: Tokens](#)
- [Vault Documentation: Cubbyhole Secret Backend](#)

- [Vault Documentation: Response Wrapping](#)

15.3.10. GCP-GCE authentication

The `gcp` auth backend allows Vault login by using existing GCP (Google Cloud Platform) IAM and GCE credentials.

GCP GCE (Google Compute Engine) authentication creates a signature in the form of a JSON Web Token (JWT) for a service account. A JWT for a Compute Engine instance is obtained from the GCE metadata service using [Instance identification](#). This API creates a JSON Web Token that can be used to confirm the instance identity.

Unlike most Vault authentication backends, this backend does not require first-deploying, or provisioning security-sensitive credentials (tokens, username/password, client certificates, etc.). Instead, it treats GCP as a Trusted Third Party and uses the cryptographically signed dynamic metadata information that uniquely represents each GCP service account.

Example 29. bootstrap.yml with required GCP-GCE Authentication properties

```
spring.cloud.vault:  
  authentication: GCP_GCE  
  gcp-gce:  
    role: my-dev-role
```

Example 30. bootstrap.yml with all GCP-GCE Authentication properties

```
spring.cloud.vault:  
  authentication: GCP_GCE  
  gcp-gce:  
    gcp-path: gcp  
    role: my-dev-role  
    service-account: my-service@projectid.iam.gserviceaccount.com
```

- `role` sets the name of the role against which the login is being attempted.
- `gcp-path` sets the path of the GCP mount to use
- `service-account` allows overriding the service account Id to a specific value. Defaults to the `default` service account.

See also:

- [Vault Documentation: Using the GCP auth backend](#)
- [GCP Documentation: Verifying the Identity of Instances](#)

15.3.11. GCP-IAM authentication

The `gcp` auth backend allows Vault login by using existing GCP (Google Cloud Platform) IAM and GCE credentials.

GCP IAM authentication creates a signature in the form of a JSON Web Token (JWT) for a service account. A JWT for a service account is obtained by calling GCP IAM's `projects.serviceAccounts.signJwt` API. The caller authenticates against GCP IAM and proves thereby its identity. This Vault backend treats GCP as a Trusted Third Party.

IAM credentials can be obtained from either the runtime environment, specifically the `GOOGLE_APPLICATION_CREDENTIALS` environment variable, the Google Compute metadata service, or supplied externally as e.g. JSON or base64 encoded. JSON is the preferred form as it carries the project id and service account identifier required for calling `projects.serviceAccounts.signJwt`.

Example 31. bootstrap.yml with required GCP-IAM Authentication properties

```
spring.cloud.vault:
  authentication: GCP_IAM
  gcp-iam:
    role: my-dev-role
```

Example 32. bootstrap.yml with all GCP-IAM Authentication properties

```
spring.cloud.vault:
  authentication: GCP_IAM
  gcp-iam:
    credentials:
      location: classpath:credentials.json
      encoded-key: e+KApn0=
    gcp-path: gcp
    jwt-validity: 15m
    project-id: my-project-id
    role: my-dev-role
    service-account-id: my-service@projectid.iam.gserviceaccount.com
```

- `role` sets the name of the role against which the login is being attempted.
- `credentials.location` path to the credentials resource that contains Google credentials in JSON format.
- `credentials.encoded-key` the base64 encoded contents of an OAuth2 account private key in the JSON format.
- `gcp-path` sets the path of the GCP mount to use
- `jwt-validity` configures the JWT token validity. Defaults to 15 minutes.
- `project-id` allows overriding the project Id to a specific value. Defaults to the project Id from the

obtained credential.

- `service-account` allows overriding the service account Id to a specific value. Defaults to the service account from the obtained credential.

GCP IAM authentication requires the Google Cloud Java SDK dependency (`com.google.apis:google-api-services-iam` and `com.google.auth:google-auth-library-oauth2-http`) as the authentication implementation uses Google APIs for credentials and JWT signing.



Google credentials require an OAuth 2 token maintaining the token lifecycle. All API is synchronous therefore, `GcpIamAuthentication` does not support `AuthenticationSteps` which is required for reactive usage.

See also:

- [Vault Documentation: Using the GCP auth backend](#)
- [GCP Documentation: projects.serviceAccounts.signJwt](#)

15.3.12. Kubernetes authentication

Kubernetes authentication mechanism (since Vault 0.8.3) allows to authenticate with Vault using a Kubernetes Service Account Token. The authentication is role based and the role is bound to a service account name and a namespace.

A file containing a JWT token for a pod's service account is automatically mounted at `/var/run/secrets/kubernetes.io/serviceaccount/token`.

Example 33. bootstrap.yml with all Kubernetes authentication properties

```
spring.cloud.vault:  
  authentication: KUBERNETES  
  kubernetes:  
    role: my-dev-role  
    kubernetes-path: kubernetes  
    service-account-token-file:  
    /var/run/secrets/kubernetes.io/serviceaccount/token
```

- `role` sets the Role.
- `kubernetes-path` sets the path of the Kubernetes mount to use.
- `service-account-token-file` sets the location of the file containing the Kubernetes Service Account Token. Defaults to `/var/run/secrets/kubernetes.io/serviceaccount/token`.

See also:

- [Vault Documentation: Kubernetes](#)
- [Kubernetes Documentation: Configure Service Accounts for Pods](#)

15.3.13. Pivotal CloudFoundry authentication

The `pcf` auth backend provides a secure introduction mechanism for applications running within Pivotal's CloudFoundry instances allowing automated retrieval of a Vault token. Unlike most Vault authentication backends, this backend does not require first-deploying, or provisioning security-sensitive credentials (tokens, username/password, client certificates, etc.) as identity provisioning is handled by PCF itself. Instead, it treats PCF as a Trusted Third Party and uses the managed instance identity.

Example 34. bootstrap.yml with required PCF Authentication properties

```
spring.cloud.vault:
  authentication: PCF
  pcf:
    role: my-dev-role
```

Example 35. bootstrap.yml with all PCF Authentication properties

```
spring.cloud.vault:
  authentication: PCF
  pcf:
    role: my-dev-role
    pcf-path: path
    instance-certificate: /etc/cf-instance-credentials/instance.crt
    instance-key: /etc/cf-instance-credentials/instance.key
```

- `role` sets the name of the role against which the login is being attempted.
- `pcf-path` sets the path of the PCF mount to use.
- `instance-certificate` sets the path to the PCF instance identity certificate. Defaults to `${CF_INSTANCE_CERT}` env variable.
- `instance-key` sets the path to the PCF instance identity key. Defaults to `${CF_INSTANCE_KEY}` env variable.



PCF authentication requires BouncyCastle (bcpkix-jdk15on) to be on the classpath for RSA PSS signing.

See also: [Vault Documentation: Using the pcf auth backend](#)

15.4. Secret Backends

15.4.1. Key-Value Backend

Spring Cloud Vault supports both Key-Value secret backends, the versioned (v2) and unversioned

(v1). The key-value backend allows storage of arbitrary values as key-value store. A single context can store one or many key-value tuples. Contexts can be organized hierarchically. Spring Cloud Vault determines itself whether a secret is using versioning and maps the path to its appropriate URL. Spring Cloud Vault allows using the Application name and a default context name (`application`) in combination with active profiles.

```
/secret/{application}/{profile}
/secret/{application}
/secret/{default-context}/{profile}
/secret/{default-context}
```

The application name is determined by the properties:

- `spring.cloud.vault.kv.application-name`
- `spring.cloud.vault.application-name`
- `spring.application.name`

The profiles are determined by the properties:

- `spring.cloud.vault.kv.profiles`
- `spring.profiles.active`

Secrets can be obtained from other contexts within the key-value backend by adding their paths to the application name, separated by commas. For example, given the application name `usefulapp,mysql1,projectx/aws`, each of these folders will be used:

- `/secret/usefulapp`
- `/secret/mysql1`
- `/secret/projectx/aws`

Spring Cloud Vault adds all active profiles to the list of possible context paths. No active profiles will skip accessing contexts with a profile name.

Properties are exposed like they are stored (i.e. without additional prefixes).



Spring Cloud Vault adds the `data/` context between the mount path and the actual context path depending on whether the mount uses the versioned key-value backend.

```
spring.cloud.vault:
  kv:
    enabled: true
    backend: secret
    profile-separator: '/'
    default-context: application
    application-name: my-app
    profiles: local, cloud
```

- **enabled** setting this value to **false** disables the secret backend config usage
- **backend** sets the path of the secret mount to use
- **default-context** sets the context name used by all applications
- **application-name** overrides the application name for use in the key-value backend
- **profiles** overrides the active profiles for use in the key-value backend
- **profile-separator** separates the profile name from the context in property sources with profiles



The key-value secret backend can be operated in versioned (v2) and non-versioned (v1) modes.

See also:

- [Vault Documentation: Using the KV Secrets Engine - Version 1 \(generic secret backend\)](#)
- [Vault Documentation: Using the KV Secrets Engine - Version 2 \(versioned key-value backend\)](#)

15.4.2. Consul

Spring Cloud Vault can obtain credentials for HashiCorp Consul. The Consul integration requires the `spring-cloud-vault-config-consul` dependency.

Example 36. pom.xml

```
<dependencies>
  <dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-vault-config-consul</artifactId>
    <version>3.0.0-M2</version>
  </dependency>
</dependencies>
```

The integration can be enabled by setting `spring.cloud.vault.consul.enabled=true` (default **false**) and providing the role name with `spring.cloud.vault.consul.role=...`.

The obtained token is stored in `spring.cloud.consul.token` so using Spring Cloud Consul can pick up

the generated credentials without further configuration. You can configure the property name by setting `spring.cloud.vault.consul.token-property`.

```
spring.cloud.vault:
  consul:
    enabled: true
    role: readonly
    backend: consul
    token-property: spring.cloud.consul.token
```

- `enabled` setting this value to `true` enables the Consul backend config usage
- `role` sets the role name of the Consul role definition
- `backend` sets the path of the Consul mount to use
- `token-property` sets the property name in which the Consul ACL token is stored

See also: [Vault Documentation: Setting up Consul with Vault](#)

15.4.3. RabbitMQ

Spring Cloud Vault can obtain credentials for RabbitMQ.

The RabbitMQ integration requires the `spring-cloud-vault-config-rabbitmq` dependency.

Example 37. pom.xml

```
<dependencies>
  <dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-vault-config-rabbitmq</artifactId>
    <version>3.0.0-M2</version>
  </dependency>
</dependencies>
```

The integration can be enabled by setting `spring.cloud.vault.rabbitmq.enabled=true` (default `false`) and providing the role name with `spring.cloud.vault.rabbitmq.role=...`.

Username and password are stored in `spring.rabbitmq.username` and `spring.rabbitmq.password` so using Spring Boot will pick up the generated credentials without further configuration. You can configure the property names by setting `spring.cloud.vault.rabbitmq.username-property` and `spring.cloud.vault.rabbitmq.password-property`.


```
spring.cloud.vault:
  rabbitmq:
    enabled: true
    role: readonly
    backend: rabbitmq
    username-property: spring.rabbitmq.username
    password-property: spring.rabbitmq.password
```

- **enabled** setting this value to **true** enables the RabbitMQ backend config usage
- **role** sets the role name of the RabbitMQ role definition
- **backend** sets the path of the RabbitMQ mount to use
- **username-property** sets the property name in which the RabbitMQ username is stored
- **password-property** sets the property name in which the RabbitMQ password is stored

See also: [Vault Documentation: Setting up RabbitMQ with Vault](#)

15.4.4. AWS

Spring Cloud Vault can obtain credentials for AWS.

The AWS integration requires the `spring-cloud-vault-config-aws` dependency.

Example 38. pom.xml

```
<dependencies>
  <dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-vault-config-aws</artifactId>
    <version>3.0.0-M2</version>
  </dependency>
</dependencies>
```

The integration can be enabled by setting `spring.cloud.vault.aws=true` (default `false`) and providing the role name with `spring.cloud.vault.aws.role=...`.

The access key and secret key are stored in `cloud.aws.credentials.accessKey` and `cloud.aws.credentials.secretKey` so using Spring Cloud AWS will pick up the generated credentials without further configuration. You can configure the property names by setting `spring.cloud.vault.aws.access-key-property` and `spring.cloud.vault.aws.secret-key-property`.

```
spring.cloud.vault:
  aws:
    enabled: true
    role: readonly
    backend: aws
    access-key-property: cloud.aws.credentials.accessKey
    secret-key-property: cloud.aws.credentials.secretKey
```

- **enabled** setting this value to **true** enables the AWS backend config usage
- **role** sets the role name of the AWS role definition
- **backend** sets the path of the AWS mount to use
- **access-key-property** sets the property name in which the AWS access key is stored
- **secret-key-property** sets the property name in which the AWS secret key is stored

See also: [Vault Documentation: Setting up AWS with Vault](#)

15.5. Database backends

Vault supports several database secret backends to generate database credentials dynamically based on configured roles. This means services that need to access a database no longer need to configure credentials: they can request them from Vault, and use Vault's leasing mechanism to more easily roll keys.

Spring Cloud Vault integrates with these backends:

- [Database](#)
- [Apache Cassandra](#)
- [Elasticsearch](#)
- [MongoDB](#)
- [MySQL](#)
- [PostgreSQL](#)

Using a database secret backend requires to enable the backend in the configuration and the `spring-cloud-vault-config-databases` dependency.

Vault ships since 0.7.1 with a dedicated **database** secret backend that allows database integration via plugins. You can use that specific backend by using the generic database backend. Make sure to specify the appropriate backend path, e.g. `spring.cloud.vault.mysql.role.backend=database`.

Example 39. pom.xml

```
<dependencies>
  <dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-vault-config-databases</artifactId>
    <version>3.0.0-M2</version>
  </dependency>
</dependencies>
```



Enabling multiple JDBC-compliant databases will generate credentials and store them by default in the same property keys hence property names for JDBC secrets need to be configured separately.

15.5.1. Database

Spring Cloud Vault can obtain credentials for any database listed at www.vaultproject.io/api/secret/databases/index.html. The integration can be enabled by setting `spring.cloud.vault.database.enabled=true` (default `false`) and providing the role name with `spring.cloud.vault.database.role=...`.

While the database backend is a generic one, `spring.cloud.vault.database` specifically targets JDBC databases. Username and password are available from `spring.datasource.username` and `spring.datasource.password` properties so using Spring Boot will pick up the generated credentials for your `DataSource` without further configuration. You can configure the property names by setting `spring.cloud.vault.database.username-property` and `spring.cloud.vault.database.password-property`.

```
spring.cloud.vault:
  database:
    enabled: true
    role: readonly
    backend: database
    username-property: spring.datasource.username
    password-property: spring.datasource.password
```

- `enabled` setting this value to `true` enables the Database backend config usage
- `role` sets the role name of the Database role definition
- `backend` sets the path of the Database mount to use
- `username-property` sets the property name in which the Database username is stored
- `password-property` sets the property name in which the Database password is stored

See also: [Vault Documentation: Database Secrets backend](#)



Spring Cloud Vault does not support getting new credentials and configuring your `DataSource` with them when the maximum lease time has been reached. That is, if `max_ttl` of the Database role in Vault is set to `24h` that means that 24 hours after your application has started it can no longer authenticate with the database.

15.5.2. Apache Cassandra



The `cassandra` backend has been deprecated in Vault 0.7.1 and it is recommended to use the `database` backend and mount it as `cassandra`.

Spring Cloud Vault can obtain credentials for Apache Cassandra. The integration can be enabled by setting `spring.cloud.vault.cassandra.enabled=true` (default `false`) and providing the role name with `spring.cloud.vault.cassandra.role=...`.

Username and password are available from `spring.data.cassandra.username` and `spring.data.cassandra.password` properties so using Spring Boot will pick up the generated credentials without further configuration. You can configure the property names by setting `spring.cloud.vault.cassandra.username-property` and `spring.cloud.vault.cassandra.password-property`.

```
spring.cloud.vault:
  cassandra:
    enabled: true
    role: readonly
    backend: cassandra
    username-property: spring.data.cassandra.username
    password-property: spring.data.cassandra.password
```

- `enabled` setting this value to `true` enables the Cassandra backend config usage
- `role` sets the role name of the Cassandra role definition
- `backend` sets the path of the Cassandra mount to use
- `username-property` sets the property name in which the Cassandra username is stored
- `password-property` sets the property name in which the Cassandra password is stored

See also: [Vault Documentation: Setting up Apache Cassandra with Vault](#)

15.5.3. Elasticsearch

Spring Cloud Vault can obtain since version 3.0 credentials for Elasticsearch. The integration can be enabled by setting `spring.cloud.vault.elasticsearch.enabled=true` (default `false`) and providing the role name with `spring.cloud.vault.elasticsearch.role=...`.

Username and password are available from `spring.elasticsearch.rest.username` and `spring.elasticsearch.rest.password` properties so using Spring Boot will pick up the generated

credentials without further configuration. You can configure the property names by setting `spring.cloud.vault.elasticsearch.username-property` and `spring.cloud.vault.elasticsearch.password-property`.

```
spring.cloud.vault:
  elasticsearch:
    enabled: true
    role: readonly
    backend: mongodb
    username-property: spring.elasticsearch.rest.username
    password-property: spring.elasticsearch.rest.password
```

- `enabled` setting this value to `true` enables the Elasticsearch database backend config usage
- `role` sets the role name of the Elasticsearch role definition
- `backend` sets the path of the Elasticsearch mount to use
- `username-property` sets the property name in which the Elasticsearch username is stored
- `password-property` sets the property name in which the Elasticsearch password is stored

See also: [Vault Documentation: Setting up Elasticsearch with Vault](#)

15.5.4. MongoDB



The `mongodb` backend has been deprecated in Vault 0.7.1 and it is recommended to use the `database` backend and mount it as `mongodb`.

Spring Cloud Vault can obtain credentials for MongoDB. The integration can be enabled by setting `spring.cloud.vault.mongodb.enabled=true` (default `false`) and providing the role name with `spring.cloud.vault.mongodb.role=...`.

Username and password are stored in `spring.data.mongodb.username` and `spring.data.mongodb.password` so using Spring Boot will pick up the generated credentials without further configuration. You can configure the property names by setting `spring.cloud.vault.mongodb.username-property` and `spring.cloud.vault.mongodb.password-property`.

```
spring.cloud.vault:
  mongodb:
    enabled: true
    role: readonly
    backend: mongodb
    username-property: spring.data.mongodb.username
    password-property: spring.data.mongodb.password
```

- `enabled` setting this value to `true` enables the MongoDB backend config usage
- `role` sets the role name of the MongoDB role definition
- `backend` sets the path of the MongoDB mount to use
- `username-property` sets the property name in which the MongoDB username is stored
- `password-property` sets the property name in which the MongoDB password is stored

See also: [Vault Documentation: Setting up MongoDB with Vault](#)

15.5.5. MySQL



The `mysql` backend has been deprecated in Vault 0.7.1 and it is recommended to use the `database` backend and mount it as `mysql`. Configuration for `spring.cloud.vault.mysql` will be removed in a future version.

Spring Cloud Vault can obtain credentials for MySQL. The integration can be enabled by setting `spring.cloud.vault.mysql.enabled=true` (default `false`) and providing the role name with `spring.cloud.vault.mysql.role=...`.

Username and password are available from `spring.datasource.username` and `spring.datasource.password` properties so using Spring Boot will pick up the generated credentials without further configuration. You can configure the property names by setting `spring.cloud.vault.mysql.username-property` and `spring.cloud.vault.mysql.password-property`.

```
spring.cloud.vault:
  mysql:
    enabled: true
    role: readonly
    backend: mysql
    username-property: spring.datasource.username
    password-property: spring.datasource.password
```

- `enabled` setting this value to `true` enables the MySQL backend config usage
- `role` sets the role name of the MySQL role definition
- `backend` sets the path of the MySQL mount to use
- `username-property` sets the property name in which the MySQL username is stored
- `password-property` sets the property name in which the MySQL password is stored

See also: [Vault Documentation: Setting up MySQL with Vault](#)

15.5.6. PostgreSQL



The `postgresql` backend has been deprecated in Vault 0.7.1 and it is recommended to use the `database` backend and mount it as `postgresql`. Configuration for `spring.cloud.vault.postgresql` will be removed in a future version.

Spring Cloud Vault can obtain credentials for PostgreSQL. The integration can be enabled by setting `spring.cloud.vault.postgresql.enabled=true` (default `false`) and providing the role name with `spring.cloud.vault.postgresql.role=...`.

Username and password are available from `spring.datasource.username` and `spring.datasource.password` properties so using Spring Boot will pick up the generated credentials without further configuration. You can configure the property names by setting `spring.cloud.vault.postgresql.username-property` and `spring.cloud.vault.postgresql.password-property`.

```
spring.cloud.vault:
  postgresql:
    enabled: true
    role: readonly
    backend: postgresql
    username-property: spring.datasource.username
    password-property: spring.datasource.password
```

- `enabled` setting this value to `true` enables the PostgreSQL backend config usage
- `role` sets the role name of the PostgreSQL role definition
- `backend` sets the path of the PostgreSQL mount to use
- `username-property` sets the property name in which the PostgreSQL username is stored
- `password-property` sets the property name in which the PostgreSQL password is stored

See also: [Vault Documentation: Setting up PostgreSQL with Vault](#)

15.6. Configure `PropertySourceLocator` behavior

Spring Cloud Vault uses property-based configuration to create `PropertySources` for key-value and discovered secret backends.

Discovered backends provide `VaultSecretBackendDescriptor` beans to describe the configuration state to use secret backend as `PropertySource`. A `SecretBackendMetadataFactory` is required to create a `SecretBackendMetadata` object which contains path, name and property transformation configuration.

`SecretBackendMetadata` is used to back a particular `PropertySource`.

You can register an arbitrary number of beans implementing `VaultConfigurer` for customization. Default key-value and discovered backend registration is disabled if Spring Cloud Vault discovers at least one `VaultConfigurer` bean. You can however enable default registration with

`SecretBackendConfigurer.registerDefaultKeyValueSecretBackends()`
`SecretBackendConfigurer.registerDefaultDiscoveredSecretBackends()`.

and

```
public class CustomizationBean implements VaultConfigurer {

    @Override
    public void addSecretBackends(SecretBackendConfigurer configurer) {

        configurer.add("secret/my-application");

        configurer.registerDefaultKeyValueSecretBackends(false);
        configurer.registerDefaultDiscoveredSecretBackends(true);
    }
}
```



All customization is required to happen in the bootstrap context. Add your configuration classes to `META-INF/spring.factories` at `org.springframework.cloud.bootstrap.BootstrapConfiguration` in your application.

15.7. Service Registry Configuration

You can use a `DiscoveryClient` (such as from Spring Cloud Consul) to locate a Vault server by setting `spring.cloud.vault.discovery.enabled=true` (default `false`). The net result of that is that your apps need a `bootstrap.yml` (or an environment variable) with the appropriate discovery configuration. The benefit is that the Vault can change its co-ordinates, as long as the discovery service is a fixed point. The default service id is `vault` but you can change that on the client with `spring.cloud.vault.discovery.serviceId`.

The discovery client implementations all support some kind of metadata map (e.g. for Eureka we have `eureka.instance.metadataMap`). Some additional properties of the service may need to be configured in its service registration metadata so that clients can connect correctly. Service registries that do not provide details about transport layer security need to provide a `scheme` metadata entry to be set either to `https` or `http`. If no scheme is configured and the service is not exposed as secure service, then configuration defaults to `spring.cloud.vault.scheme` which is `https` when it's not set.

```
spring.cloud.vault.discovery:
  enabled: true
  service-id: my-vault-service
```


15.8. Vault Client Fail Fast

In some cases, it may be desirable to fail startup of a service if it cannot connect to the Vault Server. If this is the desired behavior, set the bootstrap configuration property `spring.cloud.vault.fail-fast=true` and the client will halt with an Exception.

```
spring.cloud.vault:  
  fail-fast: true
```

15.9. Vault Enterprise Namespace Support

Vault Enterprise allows using namespaces to isolate multiple Vaults on a single Vault server. Configuring a namespace by setting `spring.cloud.vault.namespace=...` enables the namespace header `X-Vault-Namespace` on every outgoing HTTP request when using the Vault `RestTemplate` or `WebClient`.

Please note that this feature is not supported by Vault Community edition and has no effect on Vault operations.

```
spring.cloud.vault:  
  namespace: my-namespace
```

See also: [Vault Enterprise: Namespaces](#)

15.10. Vault Client SSL configuration

SSL can be configured declaratively by setting various properties. You can set either `javax.net.ssl.trustStore` to configure JVM-wide SSL settings or `spring.cloud.vault.ssl.trust-store` to set SSL settings only for Spring Cloud Vault Config.

```
spring.cloud.vault:  
  ssl:  
    trust-store: classpath:keystore.jks  
    trust-store-password: changeit
```

- `trust-store` sets the resource for the trust-store. SSL-secured Vault communication will validate the Vault SSL certificate with the specified trust-store.
- `trust-store-password` sets the trust-store password

Please note that configuring `spring.cloud.vault.ssl.*` can be only applied when either Apache Http

Components or the OkHttp client is on your class-path.

15.11. Lease lifecycle management (renewal and revocation)

With every secret, Vault creates a lease: metadata containing information such as a time duration, renewability, and more.

Vault promises that the data will be valid for the given duration, or Time To Live (TTL). Once the lease is expired, Vault can revoke the data, and the consumer of the secret can no longer be certain that it is valid.

Spring Cloud Vault maintains a lease lifecycle beyond the creation of login tokens and secrets. That said, login tokens and secrets associated with a lease are scheduled for renewal just before the lease expires until terminal expiry. Application shutdown revokes obtained login tokens and renewable leases.

Secret service and database backends (such as MongoDB or MySQL) usually generate a renewable lease so generated credentials will be disabled on application shutdown.



Static tokens are not renewed or revoked.

Lease renewal and revocation is enabled by default and can be disabled by setting `spring.cloud.vault.config.lifecycle.enabled` to `false`. This is not recommended as leases can expire and Spring Cloud Vault cannot longer access Vault or services using generated credentials and valid credentials remain active after application shutdown.

```
spring.cloud.vault:
  config.lifecycle:
    enabled: true
    min-renewal: 10s
    expiry-threshold: 1m
    lease-endpoints: Legacy
```

- `enabled` controls whether leases associated with secrets are considered to be renewed and expired secrets are rotated. Enabled by default.
- `min-renewal` sets the duration that is at least required before renewing a lease. This setting prevents renewals from happening too often.
- `expiry-threshold` sets the expiry threshold. A lease is renewed the configured period of time before it expires.
- `lease-endpoints` sets the endpoints for renew and revoke. Legacy for vault versions before 0.8 and SysLeases for later.

See also: [Vault Documentation: Lease, Renew, and Revoke](#)

Chapter 16. Spring Cloud Gateway

2020.0.0-M2

This project provides an API Gateway built on top of the Spring Ecosystem, including: Spring 5, Spring Boot 2 and Project Reactor. Spring Cloud Gateway aims to provide a simple, yet effective way to route to APIs and provide cross cutting concerns to them such as: security, monitoring/metrics, and resiliency.

16.1. How to Include Spring Cloud Gateway

To include Spring Cloud Gateway in your project, use the starter with a group ID of `org.springframework.cloud` and an artifact ID of `spring-cloud-starter-gateway`. See the [Spring Cloud Project page](#) for details on setting up your build system with the current Spring Cloud Release Train.

If you include the starter, but you do not want the gateway to be enabled, set `spring.cloud.gateway.enabled=false`.



Spring Cloud Gateway is built on [Spring Boot 2.x](#), [Spring WebFlux](#), and [Project Reactor](#). As a consequence, many of the familiar synchronous libraries (Spring Data and Spring Security, for example) and patterns you know may not apply when you use Spring Cloud Gateway. If you are unfamiliar with these projects, we suggest you begin by reading their documentation to familiarize yourself with some of the new concepts before working with Spring Cloud Gateway.



Spring Cloud Gateway requires the Netty runtime provided by Spring Boot and Spring Webflux. It does not work in a traditional Servlet Container or when built as a WAR.

16.2. Glossary

- **Route:** The basic building block of the gateway. It is defined by an ID, a destination URI, a collection of predicates, and a collection of filters. A route is matched if the aggregate predicate is true.
- **Predicate:** This is a [Java 8 Function Predicate](#). The input type is a [Spring Framework ServerWebExchange](#). This lets you match on anything from the HTTP request, such as headers or parameters.
- **Filter:** These are instances of [Spring Framework GatewayFilter](#) that have been constructed with a specific factory. Here, you can modify requests and responses before or after sending the downstream request.

16.3. How It Works

The following diagram provides a high-level overview of how Spring Cloud Gateway works:

Clients make requests to Spring Cloud Gateway. If the Gateway Handler Mapping determines that a request matches a route, it is sent to the Gateway Web Handler. This handler runs the request through a filter chain that is specific to the request. The reason the filters are divided by the dotted line is that filters can run logic both before and after the proxy request is sent. All “pre” filter logic is executed. Then the proxy request is made. After the proxy request is made, the “post” filter logic is run.



URIs defined in routes without a port get default port values of 80 and 443 for the HTTP and HTTPS URIs, respectively.

16.4. Configuring Route Predicate Factories and Gateway Filter Factories

There are two ways to configure predicates and filters: shortcuts and fully expanded arguments. Most examples below use the shortcut way.

The name and argument names will be listed as `code` in the first sentence or two of the each section. The arguments are typically listed in the order that would be needed for the shortcut configuration.

16.4.1. Shortcut Configuration

Shortcut configuration is recognized by the filter name, followed by an equals sign (=), followed by argument values separated by commas (,).

application.yml

```
spring:
  cloud:
    gateway:
      routes:
        - id: after_route
          uri: https://example.org
          predicates:
            - Cookie=mycookie,mycookievalue
```

The previous sample defines the `Cookie` Route Predicate Factory with two arguments, the cookie name, `mycookie` and the value to match `mycookievalue`.

16.4.2. Fully Expanded Arguments

Fully expanded arguments appear more like standard yaml configuration with name/value pairs. Typically, there will be a `name` key and an `args` key. The `args` key is a map of key value pairs to configure the predicate or filter.

application.yml

```
spring:
  cloud:
    gateway:
      routes:
        - id: after_route
          uri: https://example.org
          predicates:
            - name: Cookie
              args:
                name: mycookie
                regexp: mycookievalue
```

This is the full configuration of the shortcut configuration of the `Cookie` predicate shown above.

16.5. Route Predicate Factories

Spring Cloud Gateway matches routes as part of the Spring WebFlux `HandlerMapping` infrastructure. Spring Cloud Gateway includes many built-in route predicate factories. All of these predicates match on different attributes of the HTTP request. You can combine multiple route predicate factories with logical `and` statements.

16.5.1. The After Route Predicate Factory

The `After` route predicate factory takes one parameter, a `datetime` (which is a java `ZonedDateTime`). This predicate matches requests that happen after the specified `datetime`. The following example configures an after route predicate:

Example 40. application.yml

```
spring:
  cloud:
    gateway:
      routes:
        - id: after_route
          uri: https://example.org
          predicates:
            - After=2017-01-20T17:42:47.789-07:00[America/Denver]
```

This route matches any request made after Jan 20, 2017 17:42 Mountain Time (Denver).

16.5.2. The Before Route Predicate Factory

The `Before` route predicate factory takes one parameter, a `datetime` (which is a java `ZonedDateTime`). This predicate matches requests that happen before the specified `datetime`. The following example configures a before route predicate:

Example 41. application.yml

```
spring:
  cloud:
    gateway:
      routes:
        - id: before_route
          uri: https://example.org
          predicates:
            - Before=2017-01-20T17:42:47.789-07:00[America/Denver]
```

This route matches any request made before Jan 20, 2017 17:42 Mountain Time (Denver).

16.5.3. The Between Route Predicate Factory

The `Between` route predicate factory takes two parameters, `datetime1` and `datetime2` which are java `ZonedDateTime` objects. This predicate matches requests that happen after `datetime1` and before `datetime2`. The `datetime2` parameter must be after `datetime1`. The following example configures a between route predicate:

Example 42. application.yml

```
spring:
  cloud:
    gateway:
      routes:
        - id: between_route
          uri: https://example.org
          predicates:
            - Between=2017-01-20T17:42:47.789-07:00[America/Denver], 2017-01-21T17:42:47.789-07:00[America/Denver]
```

This route matches any request made after Jan 20, 2017 17:42 Mountain Time (Denver) and before Jan 21, 2017 17:42 Mountain Time (Denver). This could be useful for maintenance windows.

16.5.4. The Cookie Route Predicate Factory

The `Cookie` route predicate factory takes two parameters, the cookie `name` and a `regexp` (which is a Java regular expression). This predicate matches cookies that have the given name and whose values match the regular expression. The following example configures a cookie route predicate factory:

Example 43. *application.yml*

```
spring:
  cloud:
    gateway:
      routes:
        - id: cookie_route
          uri: https://example.org
          predicates:
            - Cookie=chocolate, ch.p
```

This route matches requests that have a cookie named `chocolate` whose value matches the `ch.p` regular expression.

16.5.5. The Header Route Predicate Factory

The `Header` route predicate factory takes two parameters, the header `name` and a `regexp` (which is a Java regular expression). This predicate matches with a header that has the given name whose value matches the regular expression. The following example configures a header route predicate:

Example 44. *application.yml*

```
spring:
  cloud:
    gateway:
      routes:
        - id: header_route
          uri: https://example.org
          predicates:
            - Header=X-Request-Id, \d+
```

This route matches if the request has a header named `X-Request-Id` whose value matches the `\d+` regular expression (that is, it has a value of one or more digits).

16.5.6. The Host Route Predicate Factory

The `Host` route predicate factory takes one parameter: a list of host name `patterns`. The pattern is an Ant-style pattern with `.` as the separator. This predicates matches the `Host` header that matches the pattern. The following example configures a host route predicate:

Example 45. application.yml

```
spring:
  cloud:
    gateway:
      routes:
        - id: host_route
          uri: https://example.org
          predicates:
            - Host=**.somehost.org,**.anotherhost.org
```

URI template variables (such as `{sub}.myhost.org`) are supported as well.

This route matches if the request has a `Host` header with a value of `www.somehost.org` or `beta.somehost.org` or `www.anotherhost.org`.

This predicate extracts the URI template variables (such as `sub`, defined in the preceding example) as a map of names and values and places it in the `ServerWebExchange.getAttributes()` with a key defined in `ServerWebExchangeUtils.URI_TEMPLATE_VARIABLES_ATTRIBUTE`. Those values are then available for use by `GatewayFilter factories`

16.5.7. The Method Route Predicate Factory

The `Method` Route Predicate Factory takes a `methods` argument which is one or more parameters: the HTTP methods to match. The following example configures a method route predicate:

Example 46. application.yml

```
spring:
  cloud:
    gateway:
      routes:
        - id: method_route
          uri: https://example.org
          predicates:
            - Method=GET,POST
```

This route matches if the request method was a `GET` or a `POST`.

16.5.8. The Path Route Predicate Factory

The `Path` Route Predicate Factory takes two parameters: a list of Spring `PathMatcher patterns` and an optional flag called `matchOptionalTrailingSeparator`. The following example configures a path route predicate:

Example 47. application.yml

```
spring:
  cloud:
    gateway:
      routes:
        - id: path_route
          uri: https://example.org
          predicates:
            - Path=/red/{segment},/blue/{segment}
```

This route matches if the request path was, for example: `/red/1` or `/red/blue` or `/blue/green`.

This predicate extracts the URI template variables (such as `segment`, defined in the preceding example) as a map of names and values and places it in the `ServerWebExchange.getAttributes()` with a key defined in `ServerWebExchangeUtils.URI_TEMPLATE_VARIABLES_ATTRIBUTE`. Those values are then available for use by `GatewayFilter factories`

A utility method (called `get`) is available to make access to these variables easier. The following example shows how to use the `get` method:

```
Map<String, String> uriVariables =
ServerWebExchangeUtils.getPathPredicateVariables(exchange);

String segment = uriVariables.get("segment");
```

16.5.9. The Query Route Predicate Factory

The `Query` route predicate factory takes two parameters: a required `param` and an optional `regexp` (which is a Java regular expression). The following example configures a query route predicate:

Example 48. application.yml

```
spring:
  cloud:
    gateway:
      routes:
        - id: query_route
          uri: https://example.org
          predicates:
            - Query=green
```

The preceding route matches if the request contained a `green` query parameter.

application.yml

```
spring:
  cloud:
    gateway:
      routes:
        - id: query_route
          uri: https://example.org
          predicates:
            - Query=red, gree.
```

The preceding route matches if the request contained a **red** query parameter whose value matched the **gree.** regexp, so **green** and **greet** would match.

16.5.10. The RemoteAddr Route Predicate Factory

The **RemoteAddr** route predicate factory takes a list (min size 1) of **sources**, which are CIDR-notation (IPv4 or IPv6) strings, such as **192.168.0.1/16** (where **192.168.0.1** is an IP address and **16** is a subnet mask). The following example configures a RemoteAddr route predicate:

Example 49. application.yml

```
spring:
  cloud:
    gateway:
      routes:
        - id: remoteaddr_route
          uri: https://example.org
          predicates:
            - RemoteAddr=192.168.1.1/24
```

This route matches if the remote address of the request was, for example, **192.168.1.10**.

16.5.11. The Weight Route Predicate Factory

The **Weight** route predicate factory takes two arguments: **group** and **weight** (an int). The weights are calculated per group. The following example configures a weight route predicate:

Example 50. application.yml

```
spring:
  cloud:
    gateway:
      routes:
        - id: weight_high
          uri: https://weighthigh.org
          predicates:
            - Weight=group1, 8
        - id: weight_low
          uri: https://weightlow.org
          predicates:
            - Weight=group1, 2
```

This route would forward ~80% of traffic to weighthigh.org and ~20% of traffic to weightlow.org

Modifying the Way Remote Addresses Are Resolved

By default, the RemoteAddr route predicate factory uses the remote address from the incoming request. This may not match the actual client IP address if Spring Cloud Gateway sits behind a proxy layer.

You can customize the way that the remote address is resolved by setting a custom `RemoteAddressResolver`. Spring Cloud Gateway comes with one non-default remote address resolver that is based off of the `X-Forwarded-For` header, `XForwardedRemoteAddressResolver`.

`XForwardedRemoteAddressResolver` has two static constructor methods, which take different approaches to security:

- `XForwardedRemoteAddressResolver::trustAll` returns a `RemoteAddressResolver` that always takes the first IP address found in the `X-Forwarded-For` header. This approach is vulnerable to spoofing, as a malicious client could set an initial value for the `X-Forwarded-For`, which would be accepted by the resolver.
- `XForwardedRemoteAddressResolver::maxTrustedIndex` takes an index that correlates to the number of trusted infrastructure running in front of Spring Cloud Gateway. If Spring Cloud Gateway is, for example only accessible through HAProxy, then a value of 1 should be used. If two hops of trusted infrastructure are required before Spring Cloud Gateway is accessible, then a value of 2 should be used.

Consider the following header value:

```
X-Forwarded-For: 0.0.0.1, 0.0.0.2, 0.0.0.3
```

The following `maxTrustedIndex` values yield the following remote addresses:

<code>maxTrustedIndex</code>	result
<code>[Integer.MIN_VALUE,0]</code>	(invalid, <code>IllegalArgumentException</code> during initialization)
1	0.0.0.3
2	0.0.0.2
3	0.0.0.1
<code>[4, Integer.MAX_VALUE]</code>	0.0.0.1

The following example shows how to achieve the same configuration with Java:

Example 51. GatewayConfig.java

```

RemoteAddressResolver resolver = XForwardedRemoteAddressResolver
    .maxTrustedIndex(1);

...

.route("direct-route",
    r -> r.remoteAddr("10.1.1.1", "10.10.1.1/24")
        .uri("https://downstream1")
.route("proxied-route",
    r -> r.remoteAddr(resolver, "10.10.1.1", "10.10.1.1/24")
        .uri("https://downstream2")
)

```

16.6. GatewayFilter Factories

Route filters allow the modification of the incoming HTTP request or outgoing HTTP response in some manner. Route filters are scoped to a particular route. Spring Cloud Gateway includes many built-in GatewayFilter Factories.



For more detailed examples of how to use any of the following filters, take a look at the [unit tests](#).

16.6.1. The AddRequestHeader GatewayFilter Factory

The `AddRequestHeader GatewayFilter` factory takes a `name` and `value` parameter. The following example configures an `AddRequestHeader GatewayFilter`:

Example 52. application.yml

```
spring:
  cloud:
    gateway:
      routes:
        - id: add_request_header_route
          uri: https://example.org
          filters:
            - AddRequestHeader=X-Request-red, blue
```

This listing adds `X-Request-red:blue` header to the downstream request's headers for all matching requests.

`AddRequestHeader` is aware of the URI variables used to match a path or host. URI variables may be used in the value and are expanded at runtime. The following example configures an `AddRequestHeader GatewayFilter` that uses a variable:

Example 53. application.yml

```
spring:
  cloud:
    gateway:
      routes:
        - id: add_request_header_route
          uri: https://example.org
          predicates:
            - Path=/red/{segment}
          filters:
            - AddRequestHeader=X-Request-Red, Blue-{segment}
```

16.6.2. The `AddRequestParameter GatewayFilter Factory`

The `AddRequestParameter GatewayFilter Factory` takes a `name` and `value` parameter. The following example configures an `AddRequestParameter GatewayFilter`:

Example 54. application.yml

```
spring:
  cloud:
    gateway:
      routes:
        - id: add_request_parameter_route
          uri: https://example.org
          filters:
            - AddRequestParameter=red, blue
```

This will add `red=blue` to the downstream request's query string for all matching requests.

`AddRequestParameter` is aware of the URI variables used to match a path or host. URI variables may be used in the value and are expanded at runtime. The following example configures an `AddRequestParameter GatewayFilter` that uses a variable:

Example 55. application.yml

```
spring:
  cloud:
    gateway:
      routes:
        - id: add_request_parameter_route
          uri: https://example.org
          predicates:
            - Host: {segment}.myhost.org
          filters:
            - AddRequestParameter=foo, bar-{segment}
```

16.6.3. The `AddResponseHeader GatewayFilter Factory`

The `AddResponseHeader GatewayFilter Factory` takes a `name` and `value` parameter. The following example configures an `AddResponseHeader GatewayFilter`:

Example 56. application.yml

```
spring:
  cloud:
    gateway:
      routes:
        - id: add_response_header_route
          uri: https://example.org
          filters:
            - AddResponseHeader=X-Response-Red, Blue
```

This adds `X-Response-Foo:Bar` header to the downstream response's headers for all matching requests.

`AddResponseHeader` is aware of URI variables used to match a path or host. URI variables may be used in the value and are expanded at runtime. The following example configures an `AddResponseHeader GatewayFilter` that uses a variable:

Example 57. application.yml

```
spring:
  cloud:
    gateway:
      routes:
        - id: add_response_header_route
          uri: https://example.org
          predicates:
            - Host: {segment}.myhost.org
          filters:
            - AddResponseHeader=foo, bar-{segment}
```

16.6.4. The `DedupeResponseHeader GatewayFilter` Factory

The `DedupeResponseHeader GatewayFilter` factory takes a `name` parameter and an optional `strategy` parameter. `name` can contain a space-separated list of header names. The following example configures a `DedupeResponseHeader GatewayFilter`:

Example 58. application.yml

```
spring:
  cloud:
    gateway:
      routes:
        - id: dedupe_response_header_route
          uri: https://example.org
          filters:
            - DedupeResponseHeader=Access-Control-Allow-Credentials Access-Control-Allow-Origin
```

This removes duplicate values of `Access-Control-Allow-Credentials` and `Access-Control-Allow-Origin` response headers in cases when both the gateway CORS logic and the downstream logic add them.

The `DedupeResponseHeader` filter also accepts an optional `strategy` parameter. The accepted values are `RETAIN_FIRST` (default), `RETAIN_LAST`, and `RETAIN_UNIQUE`.

16.6.5. Spring Cloud CircuitBreaker GatewayFilter Factory

The Spring Cloud CircuitBreaker GatewayFilter factory uses the Spring Cloud CircuitBreaker APIs to wrap Gateway routes in a circuit breaker. Spring Cloud CircuitBreaker supports multiple libraries that can be used with Spring Cloud Gateway. Spring Cloud supports Resilience4J out of the box.

To enable the Spring Cloud CircuitBreaker filter, you need to place `spring-cloud-starter-circuitbreaker-reactor-resilience4j` on the classpath. The following example configures a Spring Cloud CircuitBreaker `GatewayFilter`:

Example 59. application.yml

```
spring:
  cloud:
    gateway:
      routes:
        - id: circuitbreaker_route
          uri: https://example.org
          filters:
            - CircuitBreaker=myCircuitBreaker
```

To configure the circuit breaker, see the configuration for the underlying circuit breaker implementation you are using.

- [Resilience4J Documentation](#)

The Spring Cloud CircuitBreaker filter can also accept an optional `fallbackUri` parameter. Currently,

only **forward**: schemed URIs are supported. If the fallback is called, the request is forwarded to the controller matched by the URI. The following example configures such a fallback:

Example 60. application.yml

```
spring:
  cloud:
    gateway:
      routes:
        - id: circuitbreaker_route
          uri: lb://backing-service:8088
          predicates:
            - Path=/consumingServiceEndpoint
          filters:
            - name: CircuitBreaker
              args:
                name: myCircuitBreaker
                fallbackUri: forward:/inCaseOfFailureUseThis
            - RewritePath=/consumingServiceEndpoint, /backingServiceEndpoint
```

The following listing does the same thing in Java:

Example 61. Application.java

```
@Bean
public RouteLocator routes(RouteLocatorBuilder builder) {
    return builder.routes()
        .route("circuitbreaker_route", r -> r.path("/consumingServiceEndpoint")
            .filters(f -> f.circuitBreaker(c ->
                c.name("myCircuitBreaker").fallbackUri("forward:/inCaseOfFailureUseThis")))
            .rewritePath("/consumingServiceEndpoint",
                "/backingServiceEndpoint")).uri("lb://backing-service:8088")
        .build();
}
```

This example forwards to the **/inCaseOfFailureUseThis** URI when the circuit breaker fallback is called. Note that this example also demonstrates the (optional) Spring Cloud Netflix Ribbon load-balancing (defined by the **lb** prefix on the destination URI).

The primary scenario is to use the **fallbackUri** to define an internal controller or handler within the gateway application. However, you can also reroute the request to a controller or handler in an external application, as follows:

Example 62. application.yml

```
spring:
  cloud:
    gateway:
      routes:
        - id: ingredients
          uri: lb://ingredients
          predicates:
            - Path=//ingredients/**
          filters:
            - name: CircuitBreaker
              args:
                name: fetchIngredients
                fallbackUri: forward:/fallback
        - id: ingredients-fallback
          uri: http://localhost:9994
          predicates:
            - Path=/fallback
```

In this example, there is no `fallback` endpoint or handler in the gateway application. However, there is one in another application, registered under `localhost:9994`.

In case of the request being forwarded to fallback, the Spring Cloud CircuitBreaker Gateway filter also provides the `Throwable` that has caused it. It is added to the `ServerWebExchange` as the `ServerWebExchangeUtils.CIRCUITBREAKER_EXECUTION_EXCEPTION_ATTR` attribute that can be used when handling the fallback within the gateway application.

For the external controller/handler scenario, headers can be added with exception details. You can find more information on doing so in the [FallbackHeaders GatewayFilter Factory section](#).

16.6.6. The `FallbackHeaders GatewayFilter Factory`

The `FallbackHeaders` factory lets you add Spring Cloud CircuitBreaker execution exception details in the headers of a request forwarded to a `fallbackUri` in an external application, as in the following scenario:

Example 63. application.yml

```
spring:
  cloud:
    gateway:
      routes:
        - id: ingredients
          uri: lb://ingredients
          predicates:
            - Path=//ingredients/**
          filters:
            - name: CircuitBreaker
              args:
                name: fetchIngredients
                fallbackUri: forward:/fallback
        - id: ingredients-fallback
          uri: http://localhost:9994
          predicates:
            - Path=/fallback
          filters:
            - name: FallbackHeaders
              args:
                executionExceptionTypeHeaderName: Test-Header
```

In this example, after an execution exception occurs while running the circuit breaker, the request is forwarded to the `fallback` endpoint or handler in an application running on `localhost:9994`. The headers with the exception type, message and (if available) root cause exception type and message are added to that request by the `FallbackHeaders` filter.

You can overwrite the names of the headers in the configuration by setting the values of the following arguments (shown with their default values):

- `executionExceptionTypeHeaderName` ("Execution-Exception-Type")
- `executionExceptionMessageHeaderName` ("Execution-Exception-Message")
- `rootCauseExceptionTypeHeaderName` ("Root-Cause-Exception-Type")
- `rootCauseExceptionMessageHeaderName` ("Root-Cause-Exception-Message")

For more information on circuit breakers and the gateway see the [Spring Cloud CircuitBreaker Factory](#) section.

16.6.7. The `MapRequestHeader GatewayFilter` Factory

The `MapRequestHeader GatewayFilter` factory takes `fromHeader` and `toHeader` parameters. It creates a new named header (`toHeader`), and the value is extracted out of an existing named header (`fromHeader`) from the incoming http request. If the input header does not exist, the filter has no impact. If the new named header already exists, its values are augmented with the new values. The following example configures a `MapRequestHeader`:

Example 64. *application.yml*

```
spring:
  cloud:
    gateway:
      routes:
        - id: map_request_header_route
          uri: https://example.org
          filters:
            - MapRequestHeader=Blue, X-Request-Red
```

This adds `X-Request-Red:<values>` header to the downstream request with updated values from the incoming HTTP request's `Blue` header.

16.6.8. The `PrefixPath GatewayFilter` Factory

The `PrefixPath GatewayFilter` factory takes a single `prefix` parameter. The following example configures a `PrefixPath GatewayFilter`:

Example 65. *application.yml*

```
spring:
  cloud:
    gateway:
      routes:
        - id: prefixpath_route
          uri: https://example.org
          filters:
            - PrefixPath=/mypath
```

This will prefix `/mypath` to the path of all matching requests. So a request to `/hello` would be sent to `/mypath/hello`.

16.6.9. The `PreserveHostHeader GatewayFilter` Factory

The `PreserveHostHeader GatewayFilter` factory has no parameters. This filter sets a request attribute that the routing filter inspects to determine if the original host header should be sent, rather than the host header determined by the HTTP client. The following example configures a `PreserveHostHeader GatewayFilter`:

Example 66. *application.yml*

```
spring:
  cloud:
    gateway:
      routes:
        - id: preserve_host_route
          uri: https://example.org
          filters:
            - PreserveHostHeader
```

16.6.10. The `RequestRateLimiter GatewayFilter Factory`

The `RequestRateLimiter GatewayFilter` factory uses a `RateLimiter` implementation to determine if the current request is allowed to proceed. If it is not, a status of `HTTP 429 - Too Many Requests` (by default) is returned.

This filter takes an optional `keyResolver` parameter and parameters specific to the rate limiter (described later in this section).

`keyResolver` is a bean that implements the `KeyResolver` interface. In configuration, reference the bean by name using SpEL. `#{@myKeyResolver}` is a SpEL expression that references a bean named `myKeyResolver`. The following listing shows the `KeyResolver` interface:

Example 67. *KeyResolver.java*

```
public interface KeyResolver {
    Mono<String> resolve(ServerWebExchange exchange);
}
```

The `KeyResolver` interface lets pluggable strategies derive the key for limiting requests. In future milestone releases, there will be some `KeyResolver` implementations.

The default implementation of `KeyResolver` is the `PrincipalNameKeyResolver`, which retrieves the `Principal` from the `ServerWebExchange` and calls `Principal.getName()`.

By default, if the `KeyResolver` does not find a key, requests are denied. You can adjust this behavior by setting the `spring.cloud.gateway.filter.request-rate-limiter.deny-empty-key` (true or false) and `spring.cloud.gateway.filter.request-rate-limiter.empty-key-status-code` properties.

The `RequestRateLimiter` is not configurable with the "shortcut" notation. The following example below is *invalid*:

Example 68. `application.properties`



```
# INVALID SHORTCUT CONFIGURATION
spring.cloud.gateway.routes[0].filters[0]=RequestRateLimiter=2, 2,
#{@userkeyresolver}
```

The Redis RateLimiter

The Redis implementation is based off of work done at [Stripe](#). It requires the use of the `spring-boot-starter-data-redis-reactive` Spring Boot starter.

The algorithm used is the [Token Bucket Algorithm](#).

The `redis-rate-limiter.replenishRate` property is how many requests per second you want a user to be allowed to do, without any dropped requests. This is the rate at which the token bucket is filled.

The `redis-rate-limiter.burstCapacity` property is the maximum number of requests a user is allowed to do in a single second. This is the number of tokens the token bucket can hold. Setting this value to zero blocks all requests.

The `redis-rate-limiter.requestedTokens` property is how many tokens a request costs. This is the number of tokens taken from the bucket for each request and defaults to 1.

A steady rate is accomplished by setting the same value in `replenishRate` and `burstCapacity`. Temporary bursts can be allowed by setting `burstCapacity` higher than `replenishRate`. In this case, the rate limiter needs to be allowed some time between bursts (according to `replenishRate`), as two consecutive bursts will result in dropped requests ([HTTP 429 - Too Many Requests](#)). The following listing configures a `redis-rate-limiter`:

Rate limits below 1 request/s are accomplished by setting `replenishRate` to the wanted number of requests, `requestedTokens` to the timespan in seconds and `burstCapacity` to the product of `replenishRate` and `requestedTokens`, e.g. setting `replenishRate=1`, `requestedTokens=60` and `burstCapacity=60` will result in a limit of 1 request/min.

Example 69. application.yml

```
spring:
  cloud:
    gateway:
      routes:
        - id: requestratelimiter_route
          uri: https://example.org
          filters:
            - name: RequestRateLimiter
              args:
                redis-rate-limiter.replenishRate: 10
                redis-rate-limiter.burstCapacity: 20
                redis-rate-limiter.requestedTokens: 1
```

The following example configures a `KeyResolver` in Java:

Example 70. Config.java

```
@Bean
KeyResolver userKeyResolver() {
    return exchange ->
        Mono.just(exchange.getRequest().getQueryParams().getFirst("user"));
}
```

This defines a request rate limit of 10 per user. A burst of 20 is allowed, but, in the next second, only 10 requests are available. The `KeyResolver` is a simple one that gets the `user` request parameter (note that this is not recommended for production).

You can also define a rate limiter as a bean that implements the `RateLimiter` interface. In configuration, you can reference the bean by name using SpEL. `#{@myRateLimiter}` is a SpEL expression that references a bean with named `myRateLimiter`. The following listing defines a rate limiter that uses the `KeyResolver` defined in the previous listing:

Example 71. *application.yml*

```
spring:
  cloud:
    gateway:
      routes:
        - id: requestratelimiter_route
          uri: https://example.org
          filters:
            - name: RequestRateLimiter
              args:
                rate-limiter: "#{@myRateLimiter}"
                key-resolver: "#{@userKeyResolver}"
```

16.6.11. The `RedirectTo GatewayFilter` Factory

The `RedirectTo GatewayFilter` factory takes two parameters, `status` and `url`. The `status` parameter should be a 300 series redirect HTTP code, such as 301. The `url` parameter should be a valid URL. This is the value of the `Location` header. For relative redirects, you should use `uri: no://op` as the uri of your route definition. The following listing configures a `RedirectTo GatewayFilter`:

Example 72. *application.yml*

```
spring:
  cloud:
    gateway:
      routes:
        - id: prefixpath_route
          uri: https://example.org
          filters:
            - RedirectTo=302, https://acme.org
```

This will send a status 302 with a `Location:https://acme.org` header to perform a redirect.

16.6.12. The `RemoveRequestHeader GatewayFilter` Factory

The `RemoveRequestHeader GatewayFilter` factory takes a `name` parameter. It is the name of the header to be removed. The following listing configures a `RemoveRequestHeader GatewayFilter`:

Example 73. *application.yml*

```
spring:
  cloud:
    gateway:
      routes:
        - id: removerequestheader_route
          uri: https://example.org
          filters:
            - RemoveRequestHeader=X-Request-Foo
```

This removes the `X-Request-Foo` header before it is sent downstream.

16.6.13. `RemoveResponseHeader GatewayFilter` Factory

The `RemoveResponseHeader GatewayFilter` factory takes a `name` parameter. It is the name of the header to be removed. The following listing configures a `RemoveResponseHeader GatewayFilter`:

Example 74. *application.yml*

```
spring:
  cloud:
    gateway:
      routes:
        - id: removeresponseheader_route
          uri: https://example.org
          filters:
            - RemoveResponseHeader=X-Response-Foo
```

This will remove the `X-Response-Foo` header from the response before it is returned to the gateway client.

To remove any kind of sensitive header, you should configure this filter for any routes for which you may want to do so. In addition, you can configure this filter once by using `spring.cloud.gateway.default-filters` and have it applied to all routes.

16.6.14. The `RemoveRequestParam GatewayFilter` Factory

The `RemoveRequestParam GatewayFilter` factory takes a `name` parameter. It is the name of the query parameter to be removed. The following example configures a `RemoveRequestParam GatewayFilter`:

Example 75. *application.yml*

```
spring:
  cloud:
    gateway:
      routes:
        - id: removerequestparameter_route
          uri: https://example.org
          filters:
            - RemoveRequestParameter=red
```

This will remove the `red` parameter before it is sent downstream.

16.6.15. The `RewritePath GatewayFilter` Factory

The `RewritePath GatewayFilter` factory takes a path `regex` parameter and a `replacement` parameter. This uses Java regular expressions for a flexible way to rewrite the request path. The following listing configures a `RewritePath GatewayFilter`:

Example 76. *application.yml*

```
spring:
  cloud:
    gateway:
      routes:
        - id: rewritepath_route
          uri: https://example.org
          predicates:
            - Path=/red/**
          filters:
            - RewritePath=/red(?<segment>/?.*), ${segment}
```

For a request path of `/red/blue`, this sets the path to `/blue` before making the downstream request. Note that the `$` should be replaced with `$$` because of the YAML specification.

16.6.16. `RewriteLocationResponseHeader GatewayFilter` Factory

The `RewriteLocationResponseHeader GatewayFilter` factory modifies the value of the `Location` response header, usually to get rid of backend-specific details. It takes `stripVersionMode`, `locationHeaderName`, `hostValue`, and `protocolsRegex` parameters. The following listing configures a `RewriteLocationResponseHeader GatewayFilter`:

Example 77. *application.yml*

```
spring:
  cloud:
    gateway:
      routes:
        - id: rewriterlocationresponseheader_route
          uri: http://example.org
          filters:
            - RewriteLocationResponseHeader=AS_IN_REQUEST, Location, ,
```

For example, for a request of `POST api.example.com/some/object/name`, the `Location` response header value of `object-service.prod.example.net/v2/some/object/id` is rewritten as `api.example.com/some/object/id`.

The `stripVersionMode` parameter has the following possible values: `NEVER_STRIP`, `AS_IN_REQUEST` (default), and `ALWAYS_STRIP`.

- `NEVER_STRIP`: The version is not stripped, even if the original request path contains no version.
- `AS_IN_REQUEST`: The version is stripped only if the original request path contains no version.
- `ALWAYS_STRIP`: The version is always stripped, even if the original request path contains version.

The `hostValue` parameter, if provided, is used to replace the `host:port` portion of the response `Location` header. If it is not provided, the value of the `Host` request header is used.

The `protocolsRegex` parameter must be a valid regex `String`, against which the protocol name is matched. If it is not matched, the filter does nothing. The default is `http|https|ftp|ftps`.

16.6.17. The `RewriteResponseHeader GatewayFilter Factory`

The `RewriteResponseHeader GatewayFilter` factory takes `name`, `regex`, and `replacement` parameters. It uses Java regular expressions for a flexible way to rewrite the response header value. The following example configures a `RewriteResponseHeader GatewayFilter`:

Example 78. *application.yml*

```
spring:
  cloud:
    gateway:
      routes:
        - id: rewriterresponseheader_route
          uri: https://example.org
          filters:
            - RewriteResponseHeader=X-Response-Red, , password=[^&]+, password=***
```

For a header value of `/42?user=ford&password=omg!what&flag=true`, it is set to `/42?user=ford&password=***&flag=true` after making the downstream request. You must use `\$` to mean `$` because of the YAML specification.

16.6.18. The `SaveSession GatewayFilter` Factory

The `SaveSession GatewayFilter` factory forces a `WebSession::save` operation *before* forwarding the call downstream. This is of particular use when using something like [Spring Session](#) with a lazy data store and you need to ensure the session state has been saved before making the forwarded call. The following example configures a `SaveSession GatewayFilter`:

Example 79. application.yml

```
spring:
  cloud:
    gateway:
      routes:
        - id: save_session
          uri: https://example.org
          predicates:
            - Path=/foo/**
          filters:
            - SaveSession
```

If you integrate [Spring Security](#) with Spring Session and want to ensure security details have been forwarded to the remote process, this is critical.

16.6.19. The `SecureHeaders GatewayFilter` Factory

The `SecureHeaders GatewayFilter` factory adds a number of headers to the response, per the recommendation made in [this blog post](#).

The following headers (shown with their default values) are added:

- `X-Xss-Protection:1 (mode=block)`
- `Strict-Transport-Security (max-age=631138519)`
- `X-Frame-Options (DENY)`
- `X-Content-Type-Options (nosniff)`
- `Referrer-Policy (no-referrer)`
- `Content-Security-Policy (default-src 'self' https;; font-src 'self' https: data;; img-src 'self' https: data;; object-src 'none'; script-src https;; style-src 'self' https: 'unsafe-inline)'`
- `X-Download-Options (noopen)`
- `X-Permitted-Cross-Domain-Policies (none)`

To change the default values, set the appropriate property in the `spring.cloud.gateway.filter.secure-headers` namespace. The following properties are available:

- `xss-protection-header`
- `strict-transport-security`
- `x-frame-options`
- `x-content-type-options`
- `referrer-policy`
- `content-security-policy`
- `x-download-options`
- `x-permitted-cross-domain-policies`

To disable the default values set the `spring.cloud.gateway.filter.secure-headers.disable` property with comma-separated values. The following example shows how to do so:

```
spring.cloud.gateway.filter.secure-headers.disable=x-frame-options,strict-transport-security
```



The lowercase full name of the secure header needs to be used to disable it..

16.6.20. The `SetPath GatewayFilter` Factory

The `SetPath GatewayFilter` factory takes a path `template` parameter. It offers a simple way to manipulate the request path by allowing templated segments of the path. This uses the URI templates from Spring Framework. Multiple matching segments are allowed. The following example configures a `SetPath GatewayFilter`:

Example 80. application.yml

```
spring:
  cloud:
    gateway:
      routes:
        - id: setpath_route
          uri: https://example.org
          predicates:
            - Path=/red/{segment}
          filters:
            - SetPath=/{segment}
```

For a request path of `/red/blue`, this sets the path to `/blue` before making the downstream request.

16.6.21. The `SetRequestHeader GatewayFilter` Factory

The `SetRequestHeader GatewayFilter` factory takes `name` and `value` parameters. The following listing configures a `SetRequestHeader GatewayFilter`:

Example 81. application.yml

```
spring:
  cloud:
    gateway:
      routes:
        - id: setrequestheader_route
          uri: https://example.org
          filters:
            - SetRequestHeader=X-Request-Red, Blue
```

This `GatewayFilter` replaces (rather than adding) all headers with the given name. So, if the downstream server responded with a `X-Request-Red:1234`, this would be replaced with `X-Request-Red:Blue`, which is what the downstream service would receive.

`SetRequestHeader` is aware of URI variables used to match a path or host. URI variables may be used in the value and are expanded at runtime. The following example configures an `SetRequestHeader GatewayFilter` that uses a variable:

Example 82. application.yml

```
spring:
  cloud:
    gateway:
      routes:
        - id: setrequestheader_route
          uri: https://example.org
          predicates:
            - Host: {segment}.myhost.org
          filters:
            - SetRequestHeader=foo, bar-{segment}
```

16.6.22. The `SetResponseHeader GatewayFilter` Factory

The `SetResponseHeader GatewayFilter` factory takes `name` and `value` parameters. The following listing configures a `SetResponseHeader GatewayFilter`:

Example 83. application.yml

```
spring:
  cloud:
    gateway:
      routes:
        - id: setresponseheader_route
          uri: https://example.org
          filters:
            - SetResponseHeader=X-Response-Red, Blue
```

This `GatewayFilter` replaces (rather than adding) all headers with the given name. So, if the downstream server responded with a `X-Response-Red:1234`, this is replaced with `X-Response-Red:Blue`, which is what the gateway client would receive.

`SetResponseHeader` is aware of URI variables used to match a path or host. URI variables may be used in the value and will be expanded at runtime. The following example configures an `SetResponseHeader GatewayFilter` that uses a variable:

Example 84. application.yml

```
spring:
  cloud:
    gateway:
      routes:
        - id: setresponseheader_route
          uri: https://example.org
          predicates:
            - Host: {segment}.myhost.org
          filters:
            - SetResponseHeader=foo, bar-{segment}
```

16.6.23. The `SetStatus GatewayFilter` Factory

The `SetStatus GatewayFilter` factory takes a single parameter, `status`. It must be a valid Spring `HttpStatus`. It may be the integer value `404` or the string representation of the enumeration: `NOT_FOUND`. The following listing configures a `SetStatus GatewayFilter`:

Example 85. application.yml

```
spring:
  cloud:
    gateway:
      routes:
        - id: setstatusstring_route
          uri: https://example.org
          filters:
            - SetStatus=BAD_REQUEST
        - id: setstatusint_route
          uri: https://example.org
          filters:
            - SetStatus=401
```

In either case, the HTTP status of the response is set to 401.

You can configure the `SetStatus GatewayFilter` to return the original HTTP status code from the proxied request in a header in the response. The header is added to the response if configured with the following property:

Example 86. application.yml

```
spring:
  cloud:
    gateway:
      set-status:
        original-status-header-name: original-http-status
```

16.6.24. The `StripPrefix GatewayFilter` Factory

The `StripPrefix GatewayFilter` factory takes one parameter, `parts`. The `parts` parameter indicates the number of parts in the path to strip from the request before sending it downstream. The following listing configures a `StripPrefix GatewayFilter`:

Example 87. application.yml

```
spring:
  cloud:
    gateway:
      routes:
        - id: nameRoot
          uri: https://nameservice
          predicates:
            - Path=/name/**
          filters:
            - StripPrefix=2
```

When a request is made through the gateway to `/name/blue/red`, the request made to `nameservice` looks like `nameservice/red`.

16.6.25. The Retry GatewayFilter Factory

The `Retry GatewayFilter` factory supports the following parameters:

- `retries`: The number of retries that should be attempted.
- `statuses`: The HTTP status codes that should be retried, represented by using `org.springframework.http.HttpStatus`.
- `methods`: The HTTP methods that should be retried, represented by using `org.springframework.http.HttpMethod`.
- `series`: The series of status codes to be retried, represented by using `org.springframework.http.HttpStatus.Series`.
- `exceptions`: A list of thrown exceptions that should be retried.
- `backoff`: The configured exponential backoff for the retries. Retries are performed after a backoff interval of $\text{firstBackoff} * (\text{factor} ^ n)$, where n is the iteration. If `maxBackoff` is configured, the maximum backoff applied is limited to `maxBackoff`. If `basedOnPreviousValue` is true, the backoff is calculated by using $\text{prevBackoff} * \text{factor}$.

The following defaults are configured for `Retry` filter, if enabled:

- `retries`: Three times
- `series`: 5XX series
- `methods`: GET method
- `exceptions`: `IOException` and `TimeoutException`
- `backoff`: disabled

The following listing configures a `Retry GatewayFilter`:

```
spring:
  cloud:
    gateway:
      routes:
        - id: retry_test
          uri: http://localhost:8080/flakey
          predicates:
            - Host=*.retry.com
          filters:
            - name: Retry
              args:
                retries: 3
                statuses: BAD_GATEWAY
                methods: GET,POST
                backoff:
                  firstBackoff: 10ms
                  maxBackoff: 50ms
                  factor: 2
                  basedOnPreviousValue: false
```



When using the retry filter with a `forward:` prefixed URL, the target endpoint should be written carefully so that, in case of an error, it does not do anything that could result in a response being sent to the client and committed. For example, if the target endpoint is an annotated controller, the target controller method should not return `ResponseEntity` with an error status code. Instead, it should throw an `Exception` or signal an error (for example, through a `Mono.error(ex)` return value), which the retry filter can be configured to handle by retrying.



When using the retry filter with any HTTP method with a body, the body will be cached and the gateway will become memory constrained. The body is cached in a request attribute defined by `ServerWebExchangeUtils.CACHED_REQUEST_BODY_ATTR`. The type of the object is a `org.springframework.core.io.buffer.DataBuffer`.

16.6.26. The `RequestSize GatewayFilter` Factory

When the request size is greater than the permissible limit, the `RequestSize GatewayFilter` factory can restrict a request from reaching the downstream service. The filter takes a `maxSize` parameter. The `maxSize` is a `DataSize` type, so values can be defined as a number followed by an optional `DataUnit` suffix such as 'KB' or 'MB'. The default is 'B' for bytes. It is the permissible size limit of the request defined in bytes. The following listing configures a `RequestSize GatewayFilter`:

Example 89. application.yml

```
spring:
  cloud:
    gateway:
      routes:
        - id: request_size_route
          uri: http://localhost:8080/upload
          predicates:
            - Path=/upload
          filters:
            - name: RequestSize
              args:
                maxSize: 5000000
```

The `RequestSize GatewayFilter` factory sets the response status as `413 Payload Too Large` with an additional header `errorMessage` when the request is rejected due to size. The following example shows such an `errorMessage`:

```
errorMessage` : `Request size is larger than permissible limit. Request size is
6.0 MB where permissible limit is 5.0 MB
```



The default request size is set to five MB if not provided as a filter argument in the route definition.

16.6.27. The `SetRequestHost GatewayFilter` Factory

There are certain situation when the host header may need to be overridden. In this situation, the `SetRequestHost GatewayFilter` factory can replace the existing host header with a specified vaue. The filter takes a `host` parameter. The following listing configures a `SetRequestHost GatewayFilter`:

Example 90. *application.yml*

```
spring:
  cloud:
    gateway:
      routes:
        - id: set_request_host_header_route
          uri: http://localhost:8080/headers
          predicates:
            - Path=/headers
          filters:
            - name: SetRequestHost
              args:
                host: example.org
```

The `SetRequestHost GatewayFilter` factory replaces the value of the host header with `example.org`.

16.6.28. Modify a Request Body `GatewayFilter` Factory

You can use the `ModifyRequestBody` filter filter to modify the request body before it is sent downstream by the gateway.



This filter can be configured only by using the Java DSL.

The following listing shows how to modify a request body `GatewayFilter`:

```

@Bean
public RouteLocator routes(RouteLocatorBuilder builder) {
    return builder.routes()
        .route("rewrite_request_obj", r -> r.host("*.rewriterequestobj.org")
            .filters(f -> f.prefixPath("/httpbin")
                .modifyRequestBody(String.class, Hello.class,
                    MediaType.APPLICATION_JSON_VALUE,
                    (exchange, s) -> return Mono.just(new
                    Hello(s.toUpperCase()))).uri(uri))
            .build();
}

static class Hello {
    String message;

    public Hello() { }

    public Hello(String message) {
        this.message = message;
    }

    public String getMessage() {
        return message;
    }

    public void setMessage(String message) {
        this.message = message;
    }
}
}

```

16.6.29. Modify a Response Body `GatewayFilter` Factory

You can use the `ModifyResponseBody` filter to modify the response body before it is sent back to the client.



This filter can be configured only by using the Java DSL.

The following listing shows how to modify a response body `GatewayFilter`:

```

@Bean
public RouteLocator routes(RouteLocatorBuilder builder) {
    return builder.routes()
        .route("rewrite_response_upper", r -> r.host("*.rewriteresponseupper.org")
            .filters(f -> f.prefixPath("/httpbin")
                .modifyResponseBody(String.class, String.class,
                    (exchange, s) -> Mono.just(s.toUpperCase()))).uri(uri)
            .build());
}

```

16.6.30. Default Filters

To add a filter and apply it to all routes, you can use `spring.cloud.gateway.default-filters`. This property takes a list of filters. The following listing defines a set of default filters:

Example 91. application.yml

```

spring:
  cloud:
    gateway:
      default-filters:
        - AddResponseHeader=X-Response-Default-Red, Default-Blue
        - PrefixPath=/httpbin

```

16.7. Global Filters

The `GlobalFilter` interface has the same signature as `GatewayFilter`. These are special filters that are conditionally applied to all routes.



This interface and its usage are subject to change in future milestone releases.

16.7.1. Combined Global Filter and `GatewayFilter` Ordering

When a request matches a route, the filtering web handler adds all instances of `GlobalFilter` and all route-specific instances of `GatewayFilter` to a filter chain. This combined filter chain is sorted by the `org.springframework.core.Ordered` interface, which you can set by implementing the `getOrder()` method.

As Spring Cloud Gateway distinguishes between “pre” and “post” phases for filter logic execution (see [How it Works](#)), the filter with the highest precedence is the first in the “pre”-phase and the last in the “post”-phase.

The following listing configures a filter chain:

```
@Bean
public GlobalFilter customFilter() {
    return new CustomGlobalFilter();
}

public class CustomGlobalFilter implements GlobalFilter, Ordered {

    @Override
    public Mono<Void> filter(ServerWebExchange exchange, GatewayFilterChain chain)
    {
        log.info("custom global filter");
        return chain.filter(exchange);
    }

    @Override
    public int getOrder() {
        return -1;
    }
}
```

16.7.2. Forward Routing Filter

The `ForwardRoutingFilter` looks for a URI in the exchange attribute `ServerWebExchangeUtils.GATEWAY_REQUEST_URL_ATTR`. If the URL has a `forward` scheme (such as `forward:///localendpoint`), it uses the Spring `DispatcherHandler` to handle the request. The path part of the request URL is overridden with the path in the forward URL. The unmodified original URL is appended to the list in the `ServerWebExchangeUtils.GATEWAY_ORIGINAL_REQUEST_URL_ATTR` attribute.

16.7.3. The `LoadBalancerClient` Filter

The `LoadBalancerClientFilter` looks for a URI in the exchange attribute named `ServerWebExchangeUtils.GATEWAY_REQUEST_URL_ATTR`. If the URL has a scheme of `lb` (such as `lb://myservice`), it uses the Spring Cloud `LoadBalancerClient` to resolve the name (`myservice` in this case) to an actual host and port and replaces the URI in the same attribute. The unmodified original URL is appended to the list in the `ServerWebExchangeUtils.GATEWAY_ORIGINAL_REQUEST_URL_ATTR` attribute. The filter also looks in the `ServerWebExchangeUtils.GATEWAY_SCHEME_PREFIX_ATTR` attribute to see if it equals `lb`. If so, the same rules apply. The following listing configures a `LoadBalancerClientFilter`:

```
spring:
  cloud:
    gateway:
      routes:
        - id: myRoute
          uri: lb://service
          predicates:
            - Path=/service/**
```



By default, when a service instance cannot be found in the `LoadBalancer`, a `503` is returned. You can configure the Gateway to return a `404` by setting `spring.cloud.gateway.loadbalancer.use404=true`.



The `isSecure` value of the `ServiceInstance` returned from the `LoadBalancer` overrides the scheme specified in the request made to the Gateway. For example, if the request comes into the Gateway over `HTTPS` but the `ServiceInstance` indicates it is not secure, the downstream request is made over `HTTP`. The opposite situation can also apply. However, if `GATEWAY_SCHEME_PREFIX_ATTR` is specified for the route in the Gateway configuration, the prefix is stripped and the resulting scheme from the route URL overrides the `ServiceInstance` configuration.



`LoadBalancerClientFilter` uses a blocking ribbon `LoadBalancerClient` under the hood. We suggest you use `ReactiveLoadBalancerClientFilter` instead. You can switch to it by setting the value of the `spring.cloud.loadbalancer.ribbon.enabled` to `false`.

16.7.4. The `ReactiveLoadBalancerClientFilter`

The `ReactiveLoadBalancerClientFilter` looks for a URI in the exchange attribute named `ServerWebExchangeUtils.GATEWAY_REQUEST_URL_ATTR`. If the URL has a `lb` scheme (such as `lb://myservice`), it uses the Spring Cloud `ReactorLoadBalancer` to resolve the name (`myservice` in this example) to an actual host and port and replaces the URI in the same attribute. The unmodified original URL is appended to the list in the `ServerWebExchangeUtils.GATEWAY_ORIGINAL_REQUEST_URL_ATTR` attribute. The filter also looks in the `ServerWebExchangeUtils.GATEWAY_SCHEME_PREFIX_ATTR` attribute to see if it equals `lb`. If so, the same rules apply. The following listing configures a `ReactiveLoadBalancerClientFilter`:


```
spring:
  cloud:
    gateway:
      routes:
        - id: myRoute
          uri: lb://service
          predicates:
            - Path=/service/**
```



By default, when a service instance cannot be found by the `ReactorLoadBalancer`, a `503` is returned. You can configure the gateway to return a `404` by setting `spring.cloud.gateway.loadbalancer.use404=true`.



The `isSecure` value of the `ServiceInstance` returned from the `ReactiveLoadBalancerClientFilter` overrides the scheme specified in the request made to the Gateway. For example, if the request comes into the Gateway over `HTTPS` but the `ServiceInstance` indicates it is not secure, the downstream request is made over `HTTP`. The opposite situation can also apply. However, if `GATEWAY_SCHEME_PREFIX_ATTR` is specified for the route in the Gateway configuration, the prefix is stripped and the resulting scheme from the route URL overrides the `ServiceInstance` configuration.

16.7.5. The Netty Routing Filter

The Netty routing filter runs if the URL located in the `ServerWebExchangeUtils.GATEWAY_REQUEST_URL_ATTR` exchange attribute has a `http` or `https` scheme. It uses the Netty `HttpClient` to make the downstream proxy request. The response is put in the `ServerWebExchangeUtils.CLIENT_RESPONSE_ATTR` exchange attribute for use in a later filter. (There is also an experimental `WebClientHttpRoutingFilter` that performs the same function but does not require Netty.)

16.7.6. The Netty Write Response Filter

The `NettyWriteResponseFilter` runs if there is a Netty `HttpClientResponse` in the `ServerWebExchangeUtils.CLIENT_RESPONSE_ATTR` exchange attribute. It runs after all other filters have completed and writes the proxy response back to the gateway client response. (There is also an experimental `WebClientWriteResponseFilter` that performs the same function but does not require Netty.)

16.7.7. The `RouteToRequestUrl` Filter

If there is a `Route` object in the `ServerWebExchangeUtils.GATEWAY_ROUTE_ATTR` exchange attribute, the `RouteToRequestUrlFilter` runs. It creates a new URI, based off of the request URI but updated with the `URI` attribute of the `Route` object. The new URI is placed in the

`ServerWebExchangeUtils.GATEWAY_REQUEST_URL_ATTR` exchange attribute`.

If the URI has a scheme prefix, such as `lb:ws://serviceid`, the `lb` scheme is stripped from the URI and placed in the `ServerWebExchangeUtils.GATEWAY_SCHEME_PREFIX_ATTR` for use later in the filter chain.

16.7.8. The WebSocket Routing Filter

If the URL located in the `ServerWebExchangeUtils.GATEWAY_REQUEST_URL_ATTR` exchange attribute has a `ws` or `wss` scheme, the websocket routing filter runs. It uses the Spring WebSocket infrastructure to forward the websocket request downstream.

You can load-balance websockets by prefixing the URI with `lb`, such as `lb:ws://serviceid`.



If you use [SockJS](#) as a fallback over normal HTTP, you should configure a normal HTTP route as well as the websocket Route.

The following listing configures a websocket routing filter:

Example 95. application.yml

```
spring:
  cloud:
    gateway:
      routes:
        # SockJS route
        - id: websocket_sockjs_route
          uri: http://localhost:3001
          predicates:
            - Path=/websocket/info/**
        # Normal WebSocket route
        - id: websocket_route
          uri: ws://localhost:3001
          predicates:
            - Path=/websocket/**
```

16.7.9. The Gateway Metrics Filter

To enable gateway metrics, add `spring-boot-starter-actuator` as a project dependency. Then, by default, the gateway metrics filter runs as long as the property `spring.cloud.gateway.metrics.enabled` is not set to `false`. This filter adds a timer metric named `gateway.requests` with the following tags:

- `routeId`: The route ID.
- `routeUri`: The URI to which the API is routed.
- `outcome`: The outcome, as classified by [HttpStatus.Series](#).

- `status`: The HTTP status of the request returned to the client.
- `statusCode`: The HTTP Status of the request returned to the client.
- `method`: The HTTP method used for the request.

These metrics are then available to be scraped from `/actuator/metrics/gateway.requests` and can be easily integrated with Prometheus to create a [Grafana dashboard](#).



To enable the prometheus endpoint, add `micrometer-registry-prometheus` as a project dependency.

16.7.10. Marking An Exchange As Routed

After the gateway has routed a `ServerWebExchange`, it marks that exchange as “routed” by adding `gatewayAlreadyRouted` to the exchange attributes. Once a request has been marked as routed, other routing filters will not route the request again, essentially skipping the filter. There are convenience methods that you can use to mark an exchange as routed or check if an exchange has already been routed.

- `ServerWebExchangeUtils.isAlreadyRouted` takes a `ServerWebExchange` object and checks if it has been “routed”.
- `ServerWebExchangeUtils.setAlreadyRouted` takes a `ServerWebExchange` object and marks it as “routed”.

16.8. HttpHeadersFilters

`HttpHeadersFilters` are applied to requests before sending them downstream, such as in the `NettyRoutingFilter`.

16.8.1. Forwarded Headers Filter

The `Forwarded` Headers Filter creates a `Forwarded` header to send to the downstream service. It adds the `Host` header, scheme and port of the current request to any existing `Forwarded` header.

16.8.2. RemoveHopByHop Headers Filter

The `RemoveHopByHop` Headers Filter removes headers from forwarded requests. The default list of headers that is removed comes from the [IETF](#).

The default removed headers are:

- Connection
- Keep-Alive
- Proxy-Authenticate
- Proxy-Authorization
- TE
- Trailer

- Transfer-Encoding
- Upgrade

To change this, set the `spring.cloud.gateway.filter.remove-non-proxy-headers.headers` property to the list of header names to remove.

16.8.3. XForwarded Headers Filter

The `XForwarded` Headers Filter creates various `X-Forwarded-*` headers to send to the downstream service. It uses the `Host` header, scheme, port and path of the current request to create the various headers.

Creating of individual headers can be controlled by the following boolean properties (defaults to true):

- `spring.cloud.gateway.x-forwarded.for-enabled`
- `spring.cloud.gateway.x-forwarded.host-enabled`
- `spring.cloud.gateway.x-forwarded.port-enabled`
- `spring.cloud.gateway.x-forwarded.proto-enabled`
- `spring.cloud.gateway.x-forwarded.prefix-enabled`

Appending multiple headers can be controlled by the following boolean properties (defaults to true):

- `spring.cloud.gateway.x-forwarded.for-append`
- `spring.cloud.gateway.x-forwarded.host-append`
- `spring.cloud.gateway.x-forwarded.port-append`
- `spring.cloud.gateway.x-forwarded.proto-append`
- `spring.cloud.gateway.x-forwarded.prefix-append`

16.9. TLS and SSL

The gateway can listen for requests on HTTPS by following the usual Spring server configuration. The following example shows how to do so:

Example 96. application.yml

```
server:
  ssl:
    enabled: true
    key-alias: scg
    key-store-password: scg1234
    key-store: classpath:scg-keystore.p12
    key-store-type: PKCS12
```

You can route gateway routes to both HTTP and HTTPS backends. If you are routing to an HTTPS

backend, you can configure the gateway to trust all downstream certificates with the following configuration:

Example 97. application.yml

```
spring:
  cloud:
    gateway:
      httpclient:
        ssl:
          useInsecureTrustManager: true
```

Using an insecure trust manager is not suitable for production. For a production deployment, you can configure the gateway with a set of known certificates that it can trust with the following configuration:

Example 98. application.yml

```
spring:
  cloud:
    gateway:
      httpclient:
        ssl:
          trustedX509Certificates:
            - cert1.pem
            - cert2.pem
```

If the Spring Cloud Gateway is not provisioned with trusted certificates, the default trust store is used (which you can override by setting the `javax.net.ssl.trustStore` system property).

16.9.1. TLS Handshake

The gateway maintains a client pool that it uses to route to backends. When communicating over HTTPS, the client initiates a TLS handshake. A number of timeouts are associated with this handshake. You can configure these timeouts can be configured (defaults shown) as follows:

Example 99. application.yml

```
spring:
  cloud:
    gateway:
      httpclient:
        ssl:
          handshake-timeout-millis: 10000
          close-notify-flush-timeout-millis: 3000
          close-notify-read-timeout-millis: 0
```

16.10. Configuration

Configuration for Spring Cloud Gateway is driven by a collection of `RouteDefinitionLocator` instances. The following listing shows the definition of the `RouteDefinitionLocator` interface:

Example 100. RouteDefinitionLocator.java

```
public interface RouteDefinitionLocator {
    Flux<RouteDefinition> getRouteDefinitions();
}
```

By default, a `PropertiesRouteDefinitionLocator` loads properties by using Spring Boot's `@ConfigurationProperties` mechanism.

The earlier configuration examples all use a shortcut notation that uses positional arguments rather than named ones. The following two examples are equivalent:

Example 101. application.yml

```
spring:
  cloud:
    gateway:
      routes:
        - id: setstatus_route
          uri: https://example.org
          filters:
            - name: SetStatus
              args:
                status: 401
        - id: setstatusshortcut_route
          uri: https://example.org
          filters:
            - SetStatus=401
```

For some usages of the gateway, properties are adequate, but some production use cases benefit from loading configuration from an external source, such as a database. Future milestone versions will have `RouteDefinitionLocator` implementations based off of Spring Data Repositories, such as Redis, MongoDB, and Cassandra.

16.11. Route Metadata Configuration

You can configure additional parameters for each route by using metadata, as follows:

Example 102. application.yml

```
spring:
  cloud:
    gateway:
      routes:
        - id: route_with_metadata
          uri: https://example.org
          metadata:
            optionName: "OptionValue"
            compositeObject:
              name: "value"
            iAmNumber: 1
```

You could acquire all metadata properties from an exchange, as follows:

```
Route route = exchange.getAttribute(GATEWAY_ROUTE_ATTR);
// get all metadata properties
route.getMetadata();
// get a single metadata property
route.getMetadata(someKey);
```

16.12. Http timeouts configuration

Http timeouts (response and connect) can be configured for all routes and overridden for each specific route.

16.12.1. Global timeouts

To configure Global http timeouts:

connect-timeout must be specified in milliseconds.

response-timeout must be specified as a `java.time.Duration`

global http timeouts example

```
spring:
  cloud:
    gateway:
      httpclient:
        connect-timeout: 1000
        response-timeout: 5s
```

16.12.2. Per-route timeouts

To configure per-route timeouts:

connect-timeout must be specified in milliseconds.

response-timeout must be specified in milliseconds.

per-route http timeouts configuration via configuration

```
- id: per_route_timeouts
  uri: https://example.org
  predicates:
    - name: Path
      args:
        pattern: /delay/{timeout}
  metadata:
    response-timeout: 200
    connect-timeout: 200
```



```
import static
org.springframework.cloud.gateway.support.RouteMetadataUtils.CONNECT_TIMEOUT_ATTR;
import static
org.springframework.cloud.gateway.support.RouteMetadataUtils.RESPONSE_TIMEOUT_ATTR;

@Bean
public RouteLocator customRouteLocator(RouteLocatorBuilder routeBuilder){
    return routeBuilder.routes()
        .route("test1", r -> {
            return r.host("*.somehost.org").and().path("/somepath")
                .filters(f -> f.addRequestHeader("header1", "header-value-1"))
                .uri("http://someuri")
                .metadata(RESPONSE_TIMEOUT_ATTR, 200)
                .metadata(CONNECT_TIMEOUT_ATTR, 200);
        })
        .build();
}
```

16.12.3. Fluent Java Routes API

To allow for simple configuration in Java, the `RouteLocatorBuilder` bean includes a fluent API. The following listing shows how it works:

```
// static imports from GatewayFilters and RoutePredicates
@Bean
public RouteLocator customRouteLocator(RouteLocatorBuilder builder,
ThrottleGatewayFilterFactory throttle) {
    return builder.routes()
        .route(r -> r.host("**.abc.org").and().path("/image/png")
            .filters(f ->
                f.addResponseHeader("X-TestHeader", "foobar"))
            .uri("http://httpbin.org:80")
        )
        .route(r -> r.path("/image/webp")
            .filters(f ->
                f.addResponseHeader("X-AnotherHeader", "baz"))
            .uri("http://httpbin.org:80")
            .metadata("key", "value")
        )
        .route(r -> r.order(-1)
            .host("**.throttle.org").and().path("/get")
            .filters(f -> f.filter(throttle.apply(1,
                1,
                10,
                TimeUnit.SECONDS)))
            .uri("http://httpbin.org:80")
            .metadata("key", "value")
        )
        .build();
}
```

This style also allows for more custom predicate assertions. The predicates defined by `RouteDefinitionLocator` beans are combined using logical `and`. By using the fluent Java API, you can use the `and()`, `or()`, and `negate()` operators on the `Predicate` class.

16.12.4. The `DiscoveryClient` Route Definition Locator

You can configure the gateway to create routes based on services registered with a `DiscoveryClient` compatible service registry.

To enable this, set `spring.cloud.gateway.discovery.locator.enabled=true` and make sure a `DiscoveryClient` implementation (such as Netflix Eureka, Consul, or Zookeeper) is on the classpath and enabled.

Configuring Predicates and Filters For `DiscoveryClient` Routes

By default, the gateway defines a single predicate and filter for routes created with a `DiscoveryClient`.

The default predicate is a path predicate defined with the pattern `/serviceId/**`, where `serviceId` is the ID of the service from the `DiscoveryClient`.

The default filter is a rewrite path filter with the regex `/serviceId/(?<remaining>.*)` and the replacement `/${remaining}`. This strips the service ID from the path before the request is sent downstream.

If you want to customize the predicates or filters used by the `DiscoveryClient` routes, set `spring.cloud.gateway.discovery.locator.predicates[x]` and `spring.cloud.gateway.discovery.locator.filters[y]`. When doing so, you need to make sure to include the default predicate and filter shown earlier, if you want to retain that functionality. The following example shows what this looks like:

Example 104. application.properties

```
spring.cloud.gateway.discovery.locator.predicates[0].name: Path
spring.cloud.gateway.discovery.locator.predicates[0].args[pattern]:
"/'+serviceId+'/**'"
spring.cloud.gateway.discovery.locator.predicates[1].name: Host
spring.cloud.gateway.discovery.locator.predicates[1].args[pattern]: "'**.foo.com'"
spring.cloud.gateway.discovery.locator.filters[0].name: CircuitBreaker
spring.cloud.gateway.discovery.locator.filters[0].args[name]: serviceId
spring.cloud.gateway.discovery.locator.filters[1].name: RewritePath
spring.cloud.gateway.discovery.locator.filters[1].args[regexp]: "'/' + serviceId +
'/(?<remaining>.*)'"
spring.cloud.gateway.discovery.locator.filters[1].args[replacement]:
"'/${remaining}'"
```

16.13. Reactor Netty Access Logs

To enable Reactor Netty access logs, set `-Dreactor.netty.http.server.accessLogEnabled=true`.



It must be a Java System Property, not a Spring Boot property.

You can configure the logging system to have a separate access log file. The following example creates a Logback configuration:

Example 105. logback.xml

```
<appender name="accessLog" class="ch.qos.logback.core.FileAppender">
  <file>access_log.log</file>
  <encoder>
    <pattern>%msg%n</pattern>
  </encoder>
</appender>
<appender name="async" class="ch.qos.logback.classic.AsyncAppender">
  <appender-ref ref="accessLog" />
</appender>

<logger name="reactor.netty.http.server.AccessLog" level="INFO"
additivity="false">
  <appender-ref ref="async"/>
</logger>
```

16.14. CORS Configuration

You can configure the gateway to control CORS behavior. The “global” CORS configuration is a map of URL patterns to [Spring Framework CorsConfiguration](#). The following example configures CORS:

Example 106. application.yml

```
spring:
  cloud:
    gateway:
      globalcors:
        cors-configurations:
          '["/**"]':
            allowedOrigins: "https://docs.spring.io"
            allowedMethods:
              - GET
```

In the preceding example, CORS requests are allowed from requests that originate from [docs.spring.io](#) for all GET requested paths.

To provide the same CORS configuration to requests that are not handled by some gateway route predicate, set the `spring.cloud.gateway.globalcors.add-to-simple-url-handler-mapping` property to `true`. This is useful when you try to support CORS preflight requests and your route predicate does not evaluate to `true` because the HTTP method is `options`.

16.15. Actuator API

The `/gateway` actuator endpoint lets you monitor and interact with a Spring Cloud Gateway

application. To be remotely accessible, the endpoint has to be [enabled](#) and [exposed over HTTP or JMX](#) in the application properties. The following listing shows how to do so:

Example 107. application.properties

```
management.endpoint.gateway.enabled=true # default value
management.endpoints.web.exposure.include=gateway
```

16.15.1. Verbose Actuator Format

A new, more verbose format has been added to Spring Cloud Gateway. It adds more detail to each route, letting you view the predicates and filters associated with each route along with any configuration that is available. The following example configures `/actuator/gateway/routes`:

```
[
  {
    "predicate": "(Hosts: [**.addrequestheader.org] && Paths: [/headers], match
trailing slash: true)",
    "route_id": "add_request_header_test",
    "filters": [
      "[[AddResponseHeader X-Response-Default-Foo = 'Default-Bar'], order = 1]",
      "[[AddRequestHeader X-Request-Foo = 'Bar'], order = 1]",
      "[[PrefixPath prefix = '/httpbin'], order = 2]"
    ],
    "uri": "lb://testservice",
    "order": 0
  }
]
```

This feature is enabled by default. To disable it, set the following property:

Example 108. application.properties

```
spring.cloud.gateway.actuator.verbose.enabled=false
```

This will default to `true` in a future release.

16.15.2. Retrieving Route Filters

This section details how to retrieve route filters, including:

- [Global Filters](#)
- [\[gateway-route-filters\]](#)

Global Filters

To retrieve the [global filters](#) applied to all routes, make a **GET** request to [/actuator/gateway/globalfilters](#). The resulting response is similar to the following:

```
{
  "org.springframework.cloud.gateway.filter.LoadBalancerClientFilter@77856cc5":
  10100,
  "org.springframework.cloud.gateway.filter.RouteToRequestUrlFilter@4f6fd101":
  10000,
  "org.springframework.cloud.gateway.filter.NettyWriteResponseFilter@32d22650":
  -1,
  "org.springframework.cloud.gateway.filter.ForwardRoutingFilter@106459d9":
  2147483647,
  "org.springframework.cloud.gateway.filter.NettyRoutingFilter@1fbd5e0":
  2147483647,
  "org.springframework.cloud.gateway.filter.ForwardPathFilter@33a71d23": 0,
  "org.springframework.cloud.gateway.filter.AdaptCachedBodyGlobalFilter@135064ea":
  2147483637,
  "org.springframework.cloud.gateway.filter.WebsocketRoutingFilter@23c05889":
  2147483646
}
```

The response contains the details of the global filters that are in place. For each global filter, there is a string representation of the filter object (for example, [org.springframework.cloud.gateway.filter.LoadBalancerClientFilter@77856cc5](#)) and the corresponding [order](#) in the filter chain.}

Route Filters

To retrieve the [GatewayFilter factories](#) applied to routes, make a **GET** request to [/actuator/gateway/routefilters](#). The resulting response is similar to the following:

```
{
  "[AddRequestHeaderGatewayFilterFactory@570ed9c configClass =
  AbstractNameValueGatewayFilterFactory.NameValueConfig]": null,
  "[SecureHeadersGatewayFilterFactory@fceb5d configClass = Object]": null,
  "[SaveSessionGatewayFilterFactory@4449b273 configClass = Object]": null
}
```

The response contains the details of the [GatewayFilter](#) factories applied to any particular route. For each factory there is a string representation of the corresponding object (for example, [\[SecureHeadersGatewayFilterFactory@fceb5d configClass = Object\]](#)). Note that the **null** value is due to an incomplete implementation of the endpoint controller, because it tries to set the order of the object in the filter chain, which does not apply to a [GatewayFilter](#) factory object.

16.15.3. Refreshing the Route Cache

To clear the routes cache, make a **POST** request to `/actuator/gateway/refresh`. The request returns a 200 without a response body.

16.15.4. Retrieving the Routes Defined in the Gateway

To retrieve the routes defined in the gateway, make a **GET** request to `/actuator/gateway/routes`. The resulting response is similar to the following:

```
[{
  "route_id": "first_route",
  "route_object": {
    "predicate":
"org.springframework.cloud.gateway.handler.predicate.PathRoutePredicateFactory$$La
mbda$432/1736826640@1e9d7e7d",
    "filters": [

"OrderedGatewayFilter{delegate=org.springframework.cloud.gateway.filter.factory.Pr
eserveHostHeaderGatewayFilterFactory$$Lambda$436/674480275@6631ef72, order=0}"
    ]
  },
  "order": 0
},
{
  "route_id": "second_route",
  "route_object": {
    "predicate":
"org.springframework.cloud.gateway.handler.predicate.PathRoutePredicateFactory$$La
mbda$432/1736826640@cd8d298",
    "filters": []
  },
  "order": 0
}]
```

The response contains the details of all the routes defined in the gateway. The following table describes the structure of each element (each is a route) of the response:

Path	Type	Description
<code>route_id</code>	String	The route ID.
<code>route_object.predicate</code>	Object	The route predicate.
<code>route_object.filters</code>	Array	The <code>GatewayFilter factories</code> applied to the route.
<code>order</code>	Number	The route order.

16.15.5. Retrieving Information about a Particular Route

To retrieve information about a single route, make a **GET** request to `/actuator/gateway/routes/{id}` (for example, `/actuator/gateway/routes/first_route`). The resulting response is similar to the following:

```
{
  "id": "first_route",
  "predicates": [{
    "name": "Path",
    "args": {"_genkey_0": "/first"}
  }],
  "filters": [],
  "uri": "https://www.uri-destination.org",
  "order": 0
}]
```

The following table describes the structure of the response:

Path	Type	Description
<code>id</code>	String	The route ID.
<code>predicates</code>	Array	The collection of route predicates. Each item defines the name and the arguments of a given predicate.
<code>filters</code>	Array	The collection of filters applied to the route.
<code>uri</code>	String	The destination URI of the route.
<code>order</code>	Number	The route order.

16.15.6. Creating and Deleting a Particular Route

To create a route, make a **POST** request to `/gateway/routes/{id_route_to_create}` with a JSON body that specifies the fields of the route (see [Retrieving Information about a Particular Route](#)).

To delete a route, make a **DELETE** request to `/gateway/routes/{id_route_to_delete}`.

16.15.7. Recap: The List of All endpoints

The following table below summarizes the Spring Cloud Gateway actuator endpoints (note that each endpoint has `/actuator/gateway` as the base-path):

ID	HTTP Method	Description
<code>globalfilters</code>	GET	Displays the list of global filters applied to the routes.

ID	HTTP Method	Description
<code>routefilters</code>	GET	Displays the list of <code>GatewayFilter</code> factories applied to a particular route.
<code>refresh</code>	POST	Clears the routes cache.
<code>routes</code>	GET	Displays the list of routes defined in the gateway.
<code>routes/{id}</code>	GET	Displays information about a particular route.
<code>routes/{id}</code>	POST	Adds a new route to the gateway.
<code>routes/{id}</code>	DELETE	Removes an existing route from the gateway.

16.16. Troubleshooting

This section covers common problems that may arise when you use Spring Cloud Gateway.

16.16.1. Log Levels

The following loggers may contain valuable troubleshooting information at the `DEBUG` and `TRACE` levels:

- `org.springframework.cloud.gateway`
- `org.springframework.http.server.reactive`
- `org.springframework.web.reactive`
- `org.springframework.boot.autoconfigure.web`
- `reactor.netty`
- `redisratelimiter`

16.16.2. Wiretap

The Reactor Netty `HttpClient` and `HttpServer` can have wiretap enabled. When combined with setting the `reactor.netty` log level to `DEBUG` or `TRACE`, it enables the logging of information, such as headers and bodies sent and received across the wire. To enable wiretap, set `spring.cloud.gateway.httpserver.wiretap=true` or `spring.cloud.gateway.httpclient.wiretap=true` for the `HttpServer` and `HttpClient`, respectively.

16.17. Developer Guide

These are basic guides to writing some custom components of the gateway.

16.17.1. Writing Custom Route Predicate Factories

In order to write a Route Predicate you will need to implement `RoutePredicateFactory`. There is an abstract class called `AbstractRoutePredicateFactory` which you can extend.

MyRoutePredicateFactory.java

```
public class MyRoutePredicateFactory extends
AbstractRoutePredicateFactory<HeaderRoutePredicateFactory.Config> {

    public MyRoutePredicateFactory() {
        super(Config.class);
    }

    @Override
    public Predicate<ServerWebExchange> apply(Config config) {
        // grab configuration from Config object
        return exchange -> {
            //grab the request
            ServerHttpRequest request = exchange.getRequest();
            //take information from the request to see if it
            //matches configuration.
            return matches(config, request);
        };
    }

    public static class Config {
        //Put the configuration properties for your filter here
    }
}
```

16.17.2. Writing Custom GatewayFilter Factories

To write a `GatewayFilter`, you must implement `GatewayFilterFactory`. You can extend an abstract class called `AbstractGatewayFilterFactory`. The following examples show how to do so:

Example 109. PreGatewayFilterFactory.java

```

public class PreGatewayFilterFactory extends
AbstractGatewayFilterFactory<PreGatewayFilterFactory.Config> {

    public PreGatewayFilterFactory() {
        super(Config.class);
    }

    @Override
    public GatewayFilter apply(Config config) {
        // grab configuration from Config object
        return (exchange, chain) -> {
            //If you want to build a "pre" filter you need to manipulate the
            //request before calling chain.filter
            ServerHttpRequest.Builder builder = exchange.getRequest().mutate();
            //use builder to manipulate the request
            return
chain.filter(exchange.mutate().request(builder.build()).build());
        };
    }

    public static class Config {
        //Put the configuration properties for your filter here
    }
}

```

```
public class PostGatewayFilterFactory extends
AbstractGatewayFilterFactory<PostGatewayFilterFactory.Config> {

    public PostGatewayFilterFactory() {
        super(Config.class);
    }

    @Override
    public GatewayFilter apply(Config config) {
        // grab configuration from Config object
        return (exchange, chain) -> {
            return chain.filter(exchange).then(Mono.fromRunnable(() -> {
                ServerHttpResponse response = exchange.getResponse();
                //Manipulate the response in some way
            }));
        };
    }

    public static class Config {
        //Put the configuration properties for your filter here
    }
}
```

Naming Custom Filters And References In Configuration

Custom filters class names should end in `GatewayFilterFactory`.

For example, to reference a filter named `Something` in configuration files, the filter must be in a class named `SomethingGatewayFilterFactory`.



It is possible to create a gateway filter named without the `GatewayFilterFactory` suffix, such as `class AnotherThing`. This filter could be referenced as `AnotherThing` in configuration files. This is **not** a supported naming convention and this syntax may be removed in future releases. Please update the filter name to be compliant.

16.17.3. Writing Custom Global Filters

To write a custom global filter, you must implement `GlobalFilter` interface. This applies the filter to all requests.

The following examples show how to set up global pre and post filters, respectively:

```

@Bean
public GlobalFilter customGlobalFilter() {
    return (exchange, chain) -> exchange.getPrincipal()
        .map(Principal::getName)
        .defaultIfEmpty("Default User")
        .map(userName -> {
            //adds header to proxied request
            exchange.getRequest().mutate().header("CUSTOM-REQUEST-HEADER",
userName).build();
            return exchange;
        })
        .flatMap(chain::filter);
}

@Bean
public GlobalFilter customGlobalPostFilter() {
    return (exchange, chain) -> chain.filter(exchange)
        .then(Mono.just(exchange))
        .map(serverWebExchange -> {
            //adds header to response
            serverWebExchange.getResponse().getHeaders().set("CUSTOM-RESPONSE-
HEADER",

HttpStatus.OK.equals(serverWebExchange.getResponse().getStatusCode()) ? "It
worked": "It did not work");
            return serverWebExchange;
        })
        .then();
}

```

16.18. Building a Simple Gateway by Using Spring MVC or Webflux



The following describes an alternative style gateway. None of the prior documentation applies to what follows.

Spring Cloud Gateway provides a utility object called `ProxyExchange`. You can use it inside a regular Spring web handler as a method parameter. It supports basic downstream HTTP exchanges through methods that mirror the HTTP verbs. With MVC, it also supports forwarding to a local handler through the `forward()` method. To use the `ProxyExchange`, include the right module in your classpath (either `spring-cloud-gateway-mvc` or `spring-cloud-gateway-webflux`).

The following MVC example proxies a request to `/test` downstream to a remote server:

```

@RestController
@SpringBootApplication
public class GatewaySampleApplication {

    @Value("${remote.home}")
    private URI home;

    @GetMapping("/test")
    public ResponseEntity<?> proxy(ProxyExchange<byte[]> proxy) throws Exception {
        return proxy.uri(home.toString() + "/image/png").get();
    }
}

```

The following example does the same thing with Webflux:

```

@RestController
@SpringBootApplication
public class GatewaySampleApplication {

    @Value("${remote.home}")
    private URI home;

    @GetMapping("/test")
    public Mono<ResponseEntity<?>> proxy(ProxyExchange<byte[]> proxy) throws
Exception {
        return proxy.uri(home.toString() + "/image/png").get();
    }
}

```

Convenience methods on the `ProxyExchange` enable the handler method to discover and enhance the URI path of the incoming request. For example, you might want to extract the trailing elements of a path to pass them downstream:

```

@GetMapping("/proxy/path/**")
public ResponseEntity<?> proxyPath(ProxyExchange<byte[]> proxy) throws Exception {
    String path = proxy.path("/proxy/path/");
    return proxy.uri(home.toString() + "/foos/" + path).get();
}

```

All the features of Spring MVC and Webflux are available to gateway handler methods. As a result,

you can inject request headers and query parameters, for instance, and you can constrain the incoming requests with declarations in the mapping annotation. See the documentation for [@RequestMapping](#) in Spring MVC for more details of those features.

You can add headers to the downstream response by using the `header()` methods on [ProxyExchange](#).

You can also manipulate response headers (and anything else you like in the response) by adding a mapper to the `get()` method (and other methods). The mapper is a [Function](#) that takes the incoming [ResponseEntity](#) and converts it to an outgoing one.

First-class support is provided for “sensitive” headers (by default, [cookie](#) and [authorization](#)), which are not passed downstream, and for “proxy” ([x-forwarded-*](#)) headers.

16.19. Configuration properties

To see the list of all Spring Cloud Gateway related configuration properties, see [the appendix](#).

Chapter 17. Spring Cloud Function

Mark Fisher, Dave Syer, Oleg Zhurakousky, Anshul Mehra

3.1.0.M1

17.1. Introduction

Spring Cloud Function is a project with the following high-level goals:

- Promote the implementation of business logic via functions.
- Decouple the development lifecycle of business logic from any specific runtime target so that the same code can run as a web endpoint, a stream processor, or a task.
- Support a uniform programming model across serverless providers, as well as the ability to run standalone (locally or in a PaaS).
- Enable Spring Boot features (auto-configuration, dependency injection, metrics) on serverless providers.

It abstracts away all of the transport details and infrastructure, allowing the developer to keep all the familiar tools and processes, and focus firmly on business logic.

Here's a complete, executable, testable Spring Boot application (implementing a simple string manipulation):

```
@SpringBootApplication
public class Application {

    @Bean
    public Function<Flux<String>, Flux<String>> uppercase() {
        return flux -> flux.map(value -> value.toUpperCase());
    }

    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }
}
```

It's just a Spring Boot application, so it can be built, run and tested, locally and in a CI build, the same way as any other Spring Boot application. The `Function` is from `java.util` and `Flux` is a [Reactive Streams Publisher](#) from [Project Reactor](#). The function can be accessed over HTTP or messaging.

Spring Cloud Function has 4 main features:

In the nutshell Spring Cloud Function provides the following features: 1. Wrappers for `@Beans` of

type `Function`, `Consumer` and `Supplier`, exposing them to the outside world as either HTTP endpoints and/or message stream listeners/publishers with RabbitMQ, Kafka etc.

- *Choice of programming styles - reactive, imperative or hybrid.*
- *Function composition and adaptation (e.g., composing imperative functions with reactive).*
- *Support for reactive function with multiple inputs and outputs allowing merging, joining and other complex streaming operation to be handled by functions.*
- *Transparent type conversion of inputs and outputs.*
- *Packaging functions for deployments, specific to the target platform (e.g., Project Riff, AWS Lambda and more)*
- *Adapters to expose function to the outside world as HTTP endpoints etc.*
- *Deploying a JAR file containing such an application context with an isolated classloader, so that you can pack them together in a single JVM.*
- *Compiling strings which are Java function bodies into bytecode, and then turning them into `@Beans` that can be wrapped as above.*
- *Adapters for [AWS Lambda](#), [Azure](#), [Apache OpenWhisk](#) and possibly other "serverless" service providers.*



Spring Cloud is released under the non-restrictive Apache 2.0 license. If you would like to contribute to this section of the documentation or if you find an error, please find the source code and issue trackers in the project at [github](#).

17.2. Getting Started

Build from the command line (and "install" the samples):

```
$ ./mvnw clean install
```

(If you like to YOLO add `-DskipTests`.)

Run one of the samples, e.g.

```
$ java -jar spring-cloud-function-samples/function-sample/target/*.jar
```

This runs the app and exposes its functions over HTTP, so you can convert a string to uppercase, like this:

```
$ curl -H "Content-Type: text/plain" localhost:8080/uppercase -d Hello  
HELLO
```

You can convert multiple strings (a `Flux<String>`) by separating them with new lines

```
$ curl -H "Content-Type: text/plain" localhost:8080/uppercase -d 'Hello  
> World'  
HELLOWORLD
```

(You can use `␣` in a terminal to insert a new line in a literal string like that.)

17.3. Programming model

17.3.1. Function Catalog and Flexible Function Signatures

One of the main features of Spring Cloud Function is to adapt and support a range of type signatures for user-defined functions, while providing a consistent execution model. That's why all user defined functions are transformed into a canonical representation by `FunctionCatalog`.

While users don't normally have to care about the `FunctionCatalog` at all, it is useful to know what kind of functions are supported in user code.

It is also important to understand that Spring Cloud Function provides first class support for reactive API provided by `Project Reactor` allowing reactive primitives such as `Mono` and `Flux` to be used as types in user defined functions providing greater flexibility when choosing programming model for your function implementation. Reactive programming model also enables functional support for features that would be otherwise difficult to impossible to implement using imperative programming style. For more on this please read `Function Arity` section.

17.3.2. Java 8 function support

Spring Cloud Function embraces and builds on top of the 3 core functional interfaces defined by Java and available to us since Java 8.

- `Supplier<O>`
- `Function<I, O>`
- `Consumer<I>`

Supplier

`Supplier` can be *reactive* - `Supplier<Flux<T>>` or *imperative* - `Supplier<T>`. From the invocation standpoint this should make no difference to the implementor of such `Supplier`. However, when used within frameworks (e.g., `Spring Cloud Stream`), `Suppliers`, especially reactive, often used to represent the source of the stream, therefore they are invoked once to get the stream (e.g., `Flux`) to which consumers can subscribe to. In other words such suppliers represent an equivalent of an *infinite stream*. However, the same reactive suppliers can also represent *finite* stream(s) (e.g., result set on the polled JDBC data). In those cases such reactive suppliers must be hooked up to some polling mechanism of the underlying framework.

To assist with that Spring Cloud Function provides a marker annotation `org.springframework.cloud.function.context.PollableSupplier` to signal that such supplier produces a finite stream and may need to be polled again. That said, it is important to understand that Spring

Cloud Function itself provides no behavior for this annotation.

In addition `PollableSupplier` annotation exposes a `splittable` attribute to signal that produced stream needs to be split (see [Splitter EIP](#))

Here is the example:

```
@PollableSupplier(splittable = true)
public Supplier<Flux<String>> someSupplier() {
    return () -> {
        String v1 = String.valueOf(System.nanoTime());
        String v2 = String.valueOf(System.nanoTime());
        String v3 = String.valueOf(System.nanoTime());
        return Flux.just(v1, v2, v3);
    };
}
```

Function

Function can also be written in imperative or reactive way, yet unlike `Supplier` and `Consumer` there are no special considerations for the implementor other than understanding that when used within frameworks such as [Spring Cloud Stream](#) and others, reactive function is invoked only once to pass a reference to the stream (`Flux` or `Mono`) and imperative is invoked once per event.

Consumer

`Consumer` is a little bit special because it has a `void` return type, which implies blocking, at least potentially. Most likely you will not need to write `Consumer<Flux<?>>`, but if you do need to do that, remember to subscribe to the input flux.

17.3.3. Function Composition

Function Composition is a feature that allows one to compose several functions into one. The core support is based on function composition feature available with `Function.andThen(..)` support available since Java 8. However on top of it, we provide few additional features.

Declarative Function Composition

This feature allows you to provide composition instruction in a declarative way using `|` (pipe) or `,` (comma) delimiter when providing `spring.cloud.function.definition` property.

For example

```
--spring.cloud.function.definition=uppercase|reverse
```

Here we effectively provided a definition of a single function which itself is a composition of function `uppercase` and function `reverse`. In fact that is one of the reasons why the property name is `definition` and not `name`, since the definition of a function can be a composition of several named

functions. And as mentioned you can use `and`, instead of pipe (such as `... definition=uppercase,reverse`).

Composing non-Functions

Spring Cloud Function also supports composing Supplier with `Consumer` or `Function` as well as `Function` with `Consumer`. What's important here is to understand the end product of such definitions. Composing Supplier with Function still results in Supplier while composing Supplier with Consumer will effectively render Runnable. Following the same logic composing Function with Consumer will result in Consumer.

And of course you can't compose uncomposable such as Consumer and Function, Consumer and Supplier etc.

17.3.4. Function Routing

Since version 2.2 Spring Cloud Function provides routing feature allowing you to invoke a single function which acts as a router to an actual function you wish to invoke This feature is very useful in certain FAAS environments where maintaining configurations for several functions could be cumbersome or exposing more than one function is not possible.

The `RoutingFunction` is registered in `FunctionCatalog` under the name `functionRouter`. For simplicity and consistency you can also refer to `RoutingFunction.FUNCTION_NAME` constant.

This function has the following signature:

```
public class RoutingFunction implements Function<Object, Object> {  
    . . .  
}
```

The routing instructions could be communicated in several ways;

Message Headers

If the input argument is of type `Message<?>`, you can communicate routing instruction by setting one of `spring.cloud.function.definition` or `spring.cloud.function.routing-expression` Message headers. For more static cases you can use `spring.cloud.function.definition` header which allows you to provide the name of a single function (e.g., `...definition=foo`) or a composition instruction (e.g., `...definition=foo|bar|baz`). For more dynamic cases you can use `spring.cloud.function.routing-expression` header which allows you to use Spring Expression Language (SpEL) and provide SpEL expression that should resolve into definition of a function (as described above).



SpEL evaluation context's root object is the actual input argument, so in the case of `Message<?>` you can construct expression that has access to both `payload` and `headers` (e.g., `spring.cloud.function.routing-expression=headers.function_name`).

In specific execution environments/models the adapters are responsible to translate and communicate `spring.cloud.function.definition` and/or `spring.cloud.function.routing-expression` via Message header. For example, when using `spring-cloud-function-web` you can provide

`spring.cloud.function.definition` as an HTTP header and the framework will propagate it as well as other HTTP headers as Message headers.

Application Properties

Routing instruction can also be communicated via `spring.cloud.function.definition` or `spring.cloud.function.routing-expression` as application properties. The rules described in the previous section apply here as well. The only difference is you provide these instructions as application properties (e.g., `--spring.cloud.function.definition=foo`).



When dealing with reactive inputs (e.g., Publisher), routing instructions must only be provided via Function properties. This is due to the nature of the reactive functions which are invoked only once to pass a Publisher and the rest is handled by the reactor, hence we can not access and/or rely on the routing instructions communicated via individual values (e.g., Message).

17.3.5. Function Arity

There are times when a stream of data needs to be categorized and organized. For example, consider a classic big-data use case of dealing with unorganized data containing, let's say, 'orders' and 'invoices', and you want each to go into a separate data store. This is where function arity (functions with multiple inputs and outputs) support comes to play.

Let's look at an example of such a function (full implementation details are available [here](#)),

```
@Bean
public Function<Flux<Integer>, Tuple2<Flux<String>, Flux<String>>> organise() {
    return flux -> ...;
}
```

Given that Project Reactor is a core dependency of SCF, we are using its Tuple library. Tuples give us a unique advantage by communicating to us both *cardinality* and *type* information. Both are extremely important in the context of SCSt. Cardinality lets us know how many input and output bindings need to be created and bound to the corresponding inputs and outputs of a function. Awareness of the type information ensures proper type conversion.

Also, this is where the 'index' part of the naming convention for binding names comes into play, since, in this function, the two output binding names are `organise-out-0` and `organise-out-1`.



IMPORTANT: At the moment, function arity is **only** supported for reactive functions (`Function<TupleN<Flux<?>...>, TupleN<Flux<?>...>>`) centered on Complex event processing where evaluation and computation on confluence of events typically requires view into a stream of events rather than single event.

17.3.6. Kotlin Lambda support

We also provide support for Kotlin lambdas (since v2.0). Consider the following:

```

@Bean
open fun kotlinSupplier(): () -> String {
    return { "Hello from Kotlin" }
}

@Bean
open fun kotlinFunction(): (String) -> String {
    return { it.toUpperCase() }
}

@Bean
open fun kotlinConsumer(): (String) -> Unit {
    return { println(it) }
}

```

The above represents Kotlin lambdas configured as Spring beans. The signature of each maps to a Java equivalent of `Supplier`, `Function` and `Consumer`, and thus supported/recognized signatures by the framework. While mechanics of Kotlin-to-Java mapping are outside of the scope of this documentation, it is important to understand that the same rules for signature transformation outlined in "Java 8 function support" section are applied here as well.

To enable Kotlin support all you need is to add `spring-cloud-function-kotlin` module to your classpath which contains the appropriate autoconfiguration and supporting classes.

17.3.7. Function Component Scan

Spring Cloud Function will scan for implementations of `Function`, `Consumer` and `Supplier` in a package called `functions` if it exists. Using this feature you can write functions that have no dependencies on Spring - not even the `@Component` annotation is needed. If you want to use a different package, you can set `spring.cloud.function.scan.packages`. You can also use `spring.cloud.function.scan.enabled=false` to switch off the scan completely.

17.4. Standalone Web Applications

Functions could be automatically exported as HTTP endpoints.

The `spring-cloud-function-web` module has autoconfiguration that activates when it is included in a Spring Boot web application (with MVC support). There is also a `spring-cloud-starter-function-web` to collect all the optional dependencies in case you just want a simple getting started experience.

With the web configurations activated your app will have an MVC endpoint (on "/" by default, but configurable with `spring.cloud.function.web.path`) that can be used to access the functions in the application context where function name becomes part of the URL path. The supported content types are plain text and JSON.

Method	Path	Request	Response	Status
GET	/supplier	-	Items from the named supplier	200 OK
POST	/consumer	JSON object or text	Mirrors input and pushes request body into consumer	202 Accepted
POST	/consumer	JSON array or text with new lines	Mirrors input and pushes body into consumer one by one	202 Accepted
POST	/function	JSON object or text	The result of applying the named function	200 OK
POST	/function	JSON array or text with new lines	The result of applying the named function	200 OK
GET	/function/item	-	Convert the item into an object and return the result of applying the function	200 OK

As the table above shows the behaviour of the endpoint depends on the method and also the type of incoming request data. When the incoming data is single valued, and the target function is declared as obviously single valued (i.e. not returning a collection or `Flux`), then the response will also contain a single value. For multi-valued responses the client can ask for a server-sent event stream by sending `Accept: text/event-stream`.

Functions and consumers that are declared with input and output in `Message<?>` will see the request headers on the input messages, and the output message headers will be converted to HTTP headers.

When POSTing text the response format might be different with Spring Boot 2.0 and older versions, depending on the content negotiation (provide content type and accept headers for the best results).

See [Testing Functional Applications](#) to see the details and example on how to test such application.

17.4.1. Function Mapping rules

If there is only a single function (consumer etc.) in the catalog, the name in the path is optional. In other words, providing you only have `uppercase` function in catalog `curl -H "Content-Type: text/plain" localhost:8080/uppercase -d hello` and `curl -H "Content-Type: text/plain" localhost:8080/ -d hello` calls are identical.

Composite functions can be addressed using pipes or commas to separate function names (pipes

are legal in URL paths, but a bit awkward to type on the command line). For example, `curl -H "Content-Type: text/plain" localhost:8080/uppercase,reverse -d hello`.

For cases where there is more than a single function in catalog, each function will be exported and mapped with function name being part of the path (e.g., `localhost:8080/uppercase`). In this scenario you can still map specific function or function composition to the root path by providing `spring.cloud.function.definition` property

For example,

```
--spring.cloud.function.definition=foo|bar
```

The above property will compose 'foo' and 'bar' function and map the composed function to the "/" path.

17.4.2. Function Filtering rules

In situations where there are more than one function in catalog there may be a need to only export certain functions or function compositions. In that case you can use the same `spring.cloud.function.definition` property listing functions you intend to export delimited by `;`. Note that in this case nothing will be mapped to the root path and functions that are not listed (including compositions) are not going to be exported

For example,

```
--spring.cloud.function.definition=foo;bar
```

This will only export function `foo` and function `bar` regardless how many functions are available in catalog (e.g., `localhost:8080/foo`).

```
--spring.cloud.function.definition=foo|bar;baz
```

This will only export function composition `foo|bar` and function `baz` regardless how many functions are available in catalog (e.g., `localhost:8080/foo,bar`).

17.5. Standalone Streaming Applications

To send or receive messages from a broker (such as RabbitMQ or Kafka) you can leverage `spring-cloud-stream` project and its integration with Spring Cloud Function. Please refer to [Spring Cloud Function](#) section of the Spring Cloud Stream reference manual for more details and examples.

17.6. Deploying a Packaged Function

Spring Cloud Function provides a "deployer" library that allows you to launch a jar file (or exploded archive, or set of jar files) with an isolated class loader and expose the functions defined in it. This

is quite a powerful tool that would allow you to, for instance, adapt a function to a range of different input-output adapters without changing the target jar file. Serverless platforms often have this kind of feature built in, so you could see it as a building block for a function invoker in such a platform (indeed the [Riff](#) Java function invoker uses this library).

The standard entry point is to add `spring-cloud-function-deployer` to the classpath, the deployer kicks in and looks for some configuration to tell it where to find the function jar.

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-function-deployer</artifactId>
  <version>${spring.cloud.function.version}</version>
</dependency>
```

At a minimum the user has to provide a `spring.cloud.function.location` which is a URL or resource location for the archive containing the functions. It can optionally use a `maven:` prefix to locate the artifact via a dependency lookup (see [FunctionProperties](#) for complete details). A Spring Boot application is bootstrapped from the jar file, using the `MANIFEST.MF` to locate a start class, so that a standard Spring Boot fat jar works well, for example. If the target jar can be launched successfully then the result is a function registered in the main application's `FunctionCatalog`. The registered function can be applied by code in the main application, even though it was created in an isolated class loader (by default).

Here is the example of deploying a JAR which contains an 'uppercase' function and invoking it .

```
@SpringBootApplication
public class DeployFunctionDemo {

    public static void main(String[] args) {
        ApplicationContext context = SpringApplication.run(DeployFunctionDemo.class,
            "--spring.cloud.function.location=.../target/uppercase-0.0.1-
SNAPSHOT.jar",
            "--spring.cloud.function.definition=uppercase");

        FunctionCatalog catalog = context.getBean(FunctionCatalog.class);
        Function<String, String> function = catalog.lookup("uppercase");
        System.out.println(function.apply("hello"));
    }
}
```

17.6.1. Supported Packaging Scenarios

Currently Spring Cloud Function supports several packaging scenarios to give you the most flexibility when it comes to deploying functions.

Simple JAR

This packaging option implies no dependency on anything related to Spring. For example; Consider

that such JAR contains the following class:

```
package function.example;
...
public class UpperCaseFunction implements Function<String, String> {
    @Override
    public String apply(String value) {
        return value.toUpperCase();
    }
}
```

All you need to do is specify `location` and `function-class` properties when deploying such package:

```
--spring.cloud.function.location=target/it/simplestjar/target/simplestjar
-1.0.0.RELEASE.jar
--spring.cloud.function.function-class=function.example.UpperCaseFunction
```

It's conceivable in some cases that you might want to package multiple functions together. For such scenarios you can use `spring.cloud.function.function-class` property to list several classes delimiting them by `;`.

For example,

```
--spring.cloud.function.function
-class=function.example.UpperCaseFunction;function.example.ReverseFunction
```

Here we are identifying two functions to deploy, which we can now access in function catalog by name (e.g., `catalog.lookup("reverseFunction");`).

For more details please reference the complete sample available [here](#). You can also find a corresponding test in [FunctionDeployerTests](#).

Spring Boot JAR

This packaging option implies there is a dependency on Spring Boot and that the JAR was generated as Spring Boot JAR. That said, given that the deployed JAR runs in the isolated class loader, there will not be any version conflict with the Spring Boot version used by the actual deployer. For example; Consider that such JAR contains the following class (which could have some additional Spring dependencies providing Spring/Spring Boot is on the classpath):

```

package function.example;
. . .
public class UpperCaseFunction implements Function<String, String> {
    @Override
    public String apply(String value) {
        return value.toUpperCase();
    }
}

```

As before all you need to do is specify `location` and `function-class` properties when deploying such package:

```

--spring.cloud.function.location=target/it/simplestjar/target/simplestjar
-1.0.0.RELEASE.jar
--spring.cloud.function.function-class=function.example.UpperCaseFunction

```

For more details please reference the complete sample available [here](#). You can also find a corresponding test in [FunctionDeployerTests](#).

Spring Boot Application

This packaging option implies your JAR is complete stand alone Spring Boot application with functions as managed Spring beans. As before there is an obvious assumption that there is a dependency on Spring Boot and that the JAR was generated as Spring Boot JAR. That said, given that the deployed JAR runs in the isolated class loader, there will not be any version conflict with the Spring Boot version used by the actual deployer. For example; Consider that such JAR contains the following class:

```

package function.example;
. . .
@SpringBootApplication
public class SimpleFunctionAppApplication {

    public static void main(String[] args) {
        SpringApplication.run(SimpleFunctionAppApplication.class, args);
    }

    @Bean
    public Function<String, String> uppercase() {
        return value -> value.toUpperCase();
    }
}

```

Given that we're effectively dealing with another Spring Application context and that functions are spring managed beans, in addition to the `location` property we also specify `definition` property instead of `function-class`.

```
--spring.cloud.function.location=target/it/bootapp/target/bootapp-1.0.0.RELEASE
-exec.jar
--spring.cloud.function.definition=uppercase
```

For more details please reference the complete sample available [here](#). You can also find a corresponding test in [FunctionDeployerTests](#).



This particular deployment option may or may not have Spring Cloud Function on its classpath. From the deployer perspective this doesn't matter.

17.7. Functional Bean Definitions

Spring Cloud Function supports a "functional" style of bean declarations for small apps where you need fast startup. The functional style of bean declaration was a feature of Spring Framework 5.0 with significant enhancements in 5.1.

17.7.1. Comparing Functional with Traditional Bean Definitions

Here's a vanilla Spring Cloud Function application from with the familiar `@Configuration` and `@Bean` declaration style:

```
@SpringBootApplication
public class DemoApplication {

    @Bean
    public Function<String, String> uppercase() {
        return value -> value.toUpperCase();
    }

    public static void main(String[] args) {
        SpringApplication.run(DemoApplication.class, args);
    }
}
```

Now for the functional beans: the user application code can be recast into "functional" form, like this:

```

@SpringBootConfiguration
public class DemoApplication implements
ApplicationContextInitializer<GenericApplicationContext> {

    public static void main(String[] args) {
        FunctionalSpringApplication.run(DemoApplication.class, args);
    }

    public Function<String, String> uppercase() {
        return value -> value.toUpperCase();
    }

    @Override
    public void initialize(GenericApplicationContext context) {
        context.registerBean("demo", FunctionRegistration.class,
            () -> new FunctionRegistration<>(uppercase())
                .type(FunctionType.from(String.class).to(String.class)));
    }
}

```

The main differences are:

- The main class is an `ApplicationContextInitializer`.
- The `@Bean` methods have been converted to calls to `context.registerBean()`
- The `@SpringBootApplication` has been replaced with `@SpringBootConfiguration` to signify that we are not enabling Spring Boot autoconfiguration, and yet still marking the class as an "entry point".
- The `SpringApplication` from Spring Boot has been replaced with a `FunctionalSpringApplication` from Spring Cloud Function (it's a subclass).

The business logic beans that you register in a Spring Cloud Function app are of type `FunctionRegistration`. This is a wrapper that contains both the function and information about the input and output types. In the `@Bean` form of the application that information can be derived reflectively, but in a functional bean registration some of it is lost unless we use a `FunctionRegistration`.

An alternative to using an `ApplicationContextInitializer` and `FunctionRegistration` is to make the application itself implement `Function` (or `Consumer` or `Supplier`). Example (equivalent to the above):

```

@SpringBootConfiguration
public class DemoApplication implements Function<String, String> {

    public static void main(String[] args) {
        FunctionalSpringApplication.run(DemoApplication.class, args);
    }

    @Override
    public String apply(String value) {
        return value.toUpperCase();
    }

}

```

It would also work if you add a separate, standalone class of type `Function` and register it with the `SpringApplication` using an alternative form of the `run()` method. The main thing is that the generic type information is available at runtime through the class declaration.

Suppose you have

```

@Component
public class CustomFunction implements Function<Flux<Foo>, Flux<Bar>> {
    @Override
    public Flux<Bar> apply(Flux<Foo> flux) {
        return flux.map(foo -> new Bar("This is a Bar object from Foo value: " +
foo.getValue()));
    }

}

```

You register it as such:

```

@Override
public void initialize(GenericApplicationContext context) {
    context.registerBean("function", FunctionRegistration.class,
        () -> new FunctionRegistration<>(new
CustomFunction()).type(CustomFunction.class));
}

```

17.7.2. Limitations of Functional Bean Declaration

Most Spring Cloud Function apps have a relatively small scope compared to the whole of Spring Boot, so we are able to adapt it to these functional bean definitions easily. If you step outside that limited scope, you can extend your Spring Cloud Function app by switching back to `@Bean` style configuration, or by using a hybrid approach. If you want to take advantage of Spring Boot autoconfiguration for integrations with external datastores, for example, you will need to use `@EnableAutoConfiguration`. Your functions can still be defined using the functional declarations if

you want (i.e. the "hybrid" style), but in that case you will need to explicitly switch off the "full functional mode" using `spring.functional.enabled=false` so that Spring Boot can take back control.

17.8. Testing Functional Applications

Spring Cloud Function also has some utilities for integration testing that will be very familiar to Spring Boot users.

Suppose this is your application:

```
@SpringBootApplication
public class SampleFunctionApplication {

    public static void main(String[] args) {
        SpringApplication.run(SampleFunctionApplication.class, args);
    }

    @Bean
    public Function<String, String> uppercase() {
        return v -> v.toUpperCase();
    }
}
```

Here is an integration test for the HTTP server wrapping this application:

```
@SpringBootTest(classes = SampleFunctionApplication.class,
                webEnvironment = WebEnvironment.RANDOM_PORT)
public class WebFunctionTests {

    @Autowired
    private TestRestTemplate rest;

    @Test
    public void test() throws Exception {
        ResponseEntity<String> result = this.rest.exchange(
            RequestEntity.post(new URI("/uppercase")).body("hello"), String.class);
        System.out.println(result.getBody());
    }
}
```

or when function bean definition style is used:

```
@FunctionalSpringBootTest
public class WebFunctionTests {

    @Autowired
    private TestRestTemplate rest;

    @Test
    public void test() throws Exception {
        ResponseEntity<String> result = this.rest.exchange(
            RequestEntity.post(new URI("/uppercase")).body("hello"), String.class);
        System.out.println(result.getBody());
    }
}
```

This test is almost identical to the one you would write for the `@Bean` version of the same app - the only difference is the `@FunctionalSpringBootTest` annotation, instead of the regular `@SpringBootTest`. All the other pieces, like the `@Autowired TestRestTemplate`, are standard Spring Boot features.

And to help with correct dependencies here is the excerpt from POM


```

<parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-parent</artifactId>
  <version>2.2.2.RELEASE</version>
  <relativePath/> <!-- lookup parent from repository -->
</parent>
. . . .
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-function-web</artifactId>
  <version>3.0.1.BUILD-SNAPSHOT</version>
</dependency>
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter</artifactId>
</dependency>
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-web</artifactId>
  <scope>test</scope>
</dependency>
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-test</artifactId>
  <scope>test</scope>
  <exclusions>
    <exclusion>
      <groupId>org.junit.vintage</groupId>
      <artifactId>junit-vintage-engine</artifactId>
    </exclusion>
  </exclusions>
</dependency>

```

Or you could write a test for a non-HTTP app using just the `FunctionCatalog`. For example:

```

@RunWith(SpringRunner.class)
@FunctionalSpringBootTest
public class FunctionalTests {

    @Autowired
    private FunctionCatalog catalog;

    @Test
    public void words() throws Exception {
        Function<String, String> function = catalog.lookup(Function.class,
            "uppercase");
        assertThat(function.apply("hello")).isEqualTo("HELLO");
    }
}

```

17.9. Dynamic Compilation

There is a sample app that uses the function compiler to create a function from a configuration property. The vanilla "function-sample" also has that feature. And there are some scripts that you can run to see the compilation happening at run time. To run these examples, change into the `scripts` directory:

```
cd scripts
```

Also, start a RabbitMQ server locally (e.g. execute `rabbitmq-server`).

Start the Function Registry Service:

```
./function-registry.sh
```

Register a Function:

```
./registerFunction.sh -n uppercase -f "f->f.map(s->s.toString().toUpperCase())"
```

Run a REST Microservice using that Function:

```
./web.sh -f uppercase -p 9000
curl -H "Content-Type: text/plain" -H "Accept: text/plain" localhost:9000/uppercase -d
foo
```

Register a Supplier:

```
./registerSupplier.sh -n words -f "()->Flux.just(\"foo\", \"bar\")"
```

Run a REST Microservice using that Supplier:

```
./web.sh -s words -p 9001  
curl -H "Accept: application/json" localhost:9001/words
```

Register a Consumer:

```
./registerConsumer.sh -n print -t String -f "System.out::println"
```

Run a REST Microservice using that Consumer:

```
./web.sh -c print -p 9002  
curl -X POST -H "Content-Type: text/plain" -d foo localhost:9002/print
```

Run Stream Processing Microservices:

First register a streaming words supplier:

```
./registerSupplier.sh -n wordstream -f "()->  
>Flux.interval(Duration.ofMillis(1000)).map(i->\"message-\"+i)"
```

Then start the source (supplier), processor (function), and sink (consumer) apps (in reverse order):

```
./stream.sh -p 9103 -i uppercaseWords -c print  
./stream.sh -p 9102 -i words -f uppercase -o uppercaseWords  
./stream.sh -p 9101 -s wordstream -o words
```

The output will appear in the console of the sink app (one message per second, converted to uppercase):

```
MESSAGE-0
MESSAGE-1
MESSAGE-2
MESSAGE-3
MESSAGE-4
MESSAGE-5
MESSAGE-6
MESSAGE-7
MESSAGE-8
MESSAGE-9
...
```

17.10. Serverless Platform Adapters

As well as being able to run as a standalone process, a Spring Cloud Function application can be adapted to run one of the existing serverless platforms. In the project there are adapters for [AWS Lambda](#), [Azure](#), and [Apache OpenWhisk](#). The [Oracle Fn platform](#) has its own Spring Cloud Function adapter. And [Riff](#) supports Java functions and its [Java Function Invoker](#) acts natively as an adapter for Spring Cloud Function jars.

17.10.1. AWS Lambda

The [AWS](#) adapter takes a Spring Cloud Function app and converts it to a form that can run in AWS Lambda.

The details of how to get started with AWS Lambda is out of scope of this document, so the expectation is that user has some familiarity with AWS and AWS Lambda and wants to learn what additional value spring provides.

Getting Started

One of the goals of Spring Cloud Function framework is to provide necessary infrastructure elements to enable a *simple function application* to interact in a certain way in a particular environment. A simple function application (in context of Spring) is an application that contains beans of type Supplier, Function or Consumer. So, with AWS it means that a simple function bean should somehow be recognised and executed in AWS Lambda environment.

Let's look at the example:

```
@SpringBootApplication
public class FunctionConfiguration {

    public static void main(String[] args) {
        SpringApplication.run(FunctionConfiguration.class, args);
    }

    @Bean
    public Function<String, String> uppercase() {
        return value -> value.toUpperCase();
    }
}
```

It shows a complete Spring Boot application with a function bean defined in it. What's interesting is that on the surface this is just another boot app, but in the context of AWS Adapter it is also a perfectly valid AWS Lambda application. No other code or configuration is required. All you need to do is package it and deploy it, so let's look how we can do that.

To make things simpler we've provided a sample project ready to be built and deployed and you can access it [here](#).

You simply execute `./mvnw clean package` to generate JAR file. All the necessary maven plugins have already been setup to generate appropriate AWS deployable JAR file. (You can read more details about JAR layout in [Notes on JAR Layout](#)).

Then you have to upload the JAR file (via AWS dashboard or AWS CLI) to AWS.

When `ask` about `handler` you specify `org.springframework.cloud.function.adapter.aws.FunctionInvoker::handleRequest` which is a generic request handler.

[AWS deploy] | <https://raw.githubusercontent.com/spring-cloud/spring-cloud->

That is all. Save and execute the function with some sample data which for this function is expected to be a String which function will uppercase and return back.

While `org.springframework.cloud.function.adapter.aws.FunctionInvoker` is a general purpose AWS's `RequestHandler` implementation aimed at completely isolating you from the specifics of AWS Lambda API, for some cases you may want to specify which specific AWS's `RequestHandler` you want to use. The next section will explain you how you can accomplish just that.

AWS Request Handlers

The adapter has a couple of generic request handlers that you can use. The most generic is (and the one we used in the Getting Started section) is `org.springframework.cloud.function.adapter.aws.FunctionInvoke` which is the implementation of AWS's `RequestStreamHandler`. User doesn't need to do anything other than specify it as 'handler' on AWS dashboard when deploying function. It will handle most of the case including Kinesis, streaming etc. .

The most generic is `SpringBootStreamHandler`, which uses a Jackson `ObjectMapper` provided by Spring Boot to serialize and deserialize the objects in the function. There is also a `SpringBootRequestHandler` which you can extend, and provide the input and output types as type parameters (enabling AWS to inspect the class and do the JSON conversions itself).

If your app has more than one `@Bean` of type `Function` etc. then you can choose the one to use by configuring `function.name` (e.g. as `FUNCTION_NAME` environment variable in AWS). The functions are extracted from the Spring Cloud `FunctionCatalog` (searching first for `Function` then `Consumer` and finally `Supplier`).

Notes on JAR Layout

You don't need the Spring Cloud Function Web or Stream adapter at runtime in Lambda, so you might need to exclude those before you create the JAR you send to AWS. A Lambda application has to be shaded, but a Spring Boot standalone application does not, so you can run the same app using 2 separate jars (as per the sample). The sample app creates 2 jar files, one with an `aws` classifier for deploying in Lambda, and one executable (thin) jar that includes `spring-cloud-function-web` at runtime. Spring Cloud Function will try and locate a "main class" for you from the JAR file manifest, using the `Start-Class` attribute (which will be added for you by the Spring Boot tooling if you use the starter parent). If there is no `Start-Class` in your manifest you can use an environment variable or system property `MAIN_CLASS` when you deploy the function to AWS.

If you are not using the functional bean definitions but relying on Spring Boot's auto-configuration, then additional transformers must be configured as part of the maven-shade-plugin execution.

```

<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-shade-plugin</artifactId>
  <dependencies>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-maven-plugin</artifactId>
    </dependency>
  </dependencies>
  <configuration>
    <createDependencyReducedPom>>false</createDependencyReducedPom>
    <shadedArtifactAttached>true</shadedArtifactAttached>
    <shadedClassifierName>aws</shadedClassifierName>
    <transformers>
      <transformer
implementation="org.apache.maven.plugins.shade.resource.AppendingTransformer">
        <resource>META-INF/spring.handlers</resource>
      </transformer>
      <transformer
implementation="org.springframework.boot.maven.PropertiesMergingResourceTransformer">
        <resource>META-INF/spring.factories</resource>
      </transformer>
      <transformer
implementation="org.apache.maven.plugins.shade.resource.AppendingTransformer">
        <resource>META-INF/spring.schemas</resource>
      </transformer>
    </transformers>
  </configuration>
</plugin>

```

Build file setup

In order to run Spring Cloud Function applications on AWS Lambda, you can leverage Maven or Gradle plugins offered by the cloud platform provider.

Maven

In order to use the adapter plugin for Maven, add the plugin dependency to your `pom.xml` file:

```

<dependencies>
  <dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-function-adapter-aws</artifactId>
  </dependency>
</dependencies>

```

As pointed out in the [Notes on JAR Layout](#), you will need a shaded jar in order to upload it to AWS Lambda. You can use the [Maven Shade Plugin](#) for that. The example of the [setup](#) can be found

above.

You can use the Spring Boot Maven Plugin to generate the [thin jar](#).

```
<plugin>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-maven-plugin</artifactId>
  <dependencies>
    <dependency>
      <groupId>org.springframework.boot.experimental</groupId>
      <artifactId>spring-boot-thin-layout</artifactId>
      <version>${wrapper.version}</version>
    </dependency>
  </dependencies>
</plugin>
```

You can find the entire sample [pom.xml](#) file for deploying Spring Cloud Function applications to AWS Lambda with Maven [here](#).

Gradle

In order to use the adapter plugin for Gradle, add the dependency to your [build.gradle](#) file:

```
dependencies {
  compile("org.springframework.cloud:spring-cloud-function-adapter-aws:${version}")
}
```

As pointed out in [Notes on JAR Layout](#), you will need a shaded jar in order to upload it to AWS Lambda. You can use the [Gradle Shadow Plugin](#) for that:


```

buildscript {
    dependencies {
        classpath "com.github.jengelman.gradle.plugins:shadow:${shadowPluginVersion}"
    }
}
apply plugin: 'com.github.johnrengelman.shadow'

assemble.dependsOn = [shadowJar]

import com.github.jengelman.gradle.plugins.shadow.transformers.*

shadowJar {
    classifier = 'aws'
    dependencies {
        exclude(
            dependency("org.springframework.cloud:spring-cloud-function-
web:${springCloudFunctionVersion}"))
    }
    // Required for Spring
    mergeServiceFiles()
    append 'META-INF/spring.handlers'
    append 'META-INF/spring.schemas'
    append 'META-INF/spring.tooling'
    transform(PropertiesFileTransformer) {
        paths = ['META-INF/spring.factories']
        mergeStrategy = "append"
    }
}
}

```

You can use the Spring Boot Gradle Plugin and Spring Boot Thin Gradle Plugin to generate the [thin jar](#).

```

buildscript {
    dependencies {
        classpath("org.springframework.boot.experimental:spring-boot-thin-gradle-
plugin:${wrapperVersion}")
        classpath("org.springframework.boot:spring-boot-gradle-
plugin:${springBootVersion}")
    }
}
apply plugin: 'org.springframework.boot'
apply plugin: 'org.springframework.boot.experimental.thin-launcher'
assemble.dependsOn = [thinJar]

```

You can find the entire sample `build.gradle` file for deploying Spring Cloud Function applications to AWS Lambda with Gradle [here](#).

Upload

Build the sample under `spring-cloud-function-samples/function-sample-aws` and upload the `-aws` jar file to Lambda. The handler can be `example.Handler` or `org.springframework.cloud.function.adapter.aws.SpringBootStreamHandler` (FQN of the class, *not* a method reference, although Lambda does accept method references).

```
./mvnw -U clean package
```

Using the AWS command line tools it looks like this:

```
aws lambda create-function --function-name Uppercase --role
arn:aws:iam::[USERID]:role/service-role/[ROLE] --zip-file fileb://function-sample-
aws/target/function-sample-aws-2.0.0.BUILD-SNAPSHOT-aws.jar --handler
org.springframework.cloud.function.adapter.aws.SpringBootStreamHandler --description
"Spring Cloud Function Adapter Example" --runtime java8 --region us-east-1 --timeout
30 --memory-size 1024 --publish
```

The input type for the function in the AWS sample is a `Foo` with a single property called `"value"`. So you would need this to test it:

```
{
  "value": "test"
}
```



The AWS sample app is written in the "functional" style (as an `ApplicationContextInitializer`). This is much faster on startup in Lambda than the traditional `@Bean` style, so if you don't need `@Beans` (or `@EnableAutoConfiguration`) it's a good choice. Warm starts are not affected.

Type Conversion

Spring Cloud Function will attempt to transparently handle type conversion between the raw input stream and types declared by your function.

For example, if your function signature is as such `Function<Foo, Bar>` we will attempt to convert incoming stream event to an instance of `Foo`.

In the event type is not known or can not be determined (e.g., `Function<?, ?>`) we will attempt to convert an incoming stream event to a generic `Map`.

Raw Input

There are times when you may want to have access to a raw input. In this case all you need is to declare your function signature to accept `InputStream`. For example, `Function<InputStream, ?>`. In this case we will not attempt any conversion and will pass the raw input directly to a function.

17.10.2. Microsoft Azure

The [Azure](#) adapter bootstraps a Spring Cloud Function context and channels function calls from the Azure framework into the user functions, using Spring Boot configuration where necessary. Azure Functions has quite a unique, but invasive programming model, involving annotations in user code that are specific to the platform. The easiest way to use it with Spring Cloud is to extend a base class and write a method in it with the `@FunctionName` annotation which delegates to a base class method.

This project provides an adapter layer for a Spring Cloud Function application onto Azure. You can write an app with a single `@Bean` of type `Function` and it will be deployable in Azure if you get the JAR file laid out right.

There is an `AzureSpringBootRequestHandler` which you must extend, and provide the input and output types as annotated method parameters (enabling Azure to inspect the class and create JSON bindings). The base class has two useful methods (`handleRequest` and `handleOutput`) to which you can delegate the actual function call, so mostly the function will only ever have one line.

Example:

```
public class FooHandler extends AzureSpringBootRequestHandler<Foo, Bar> {
    @FunctionName("uppercase")
    public Bar execute(@HttpTrigger(name = "req", methods = {HttpMethod.GET,
        HttpMethod.POST}, authLevel = AuthorizationLevel.ANONYMOUS)
        HttpRequestMessage<Optional<Foo>> request,
        ExecutionContext context) {
        return handleRequest(request.getBody().get(), context);
    }
}
```

This Azure handler will delegate to a `Function<Foo,Bar>` bean (or a `Function<Publisher<Foo>,Publisher<Bar>>`). Some Azure triggers (e.g. `@CosmosDBTrigger`) result in an input type of `List` and in that case you can bind to `List` in the Azure handler, or `String` (the raw JSON). The `List` input delegates to a `Function` with input type `Map<String,Object>`, or `Publisher` or `List` of the same type. The output of the `Function` can be a `List` (one-for-one) or a single value (aggregation), and the output binding in the Azure declaration should match.

If your app has more than one `@Bean` of type `Function` etc. then you can choose the one to use by configuring `function.name`. Or if you make the `@FunctionName` in the Azure handler method match the function name it should work that way (also for function apps with multiple functions). The functions are extracted from the Spring Cloud `FunctionCatalog` so the default function names are the same as the bean names.

Accessing Azure ExecutionContext

Some time there is a need to access the target execution context provided by Azure runtime in the form of `com.microsoft.azure.functions.ExecutionContext`. For example one of such needs is logging, so it can appear in the Azure console.

For that purpose Spring Cloud Function will register `ExecutionContext` as bean in the Application

context, so it could be injected into your function. For example

```
@Bean
public Function<Foo, Bar> uppercase(ExecutionContext targetContext) {
    return foo -> {
        targetContext.getLogger().info("Invoking 'uppercase' on " + foo.getValue());
        return new Bar(foo.getValue().toUpperCase());
    };
}
```

Normally type-based injection should suffice, however if need to you can also utilise the bean name under which it is registered which is `targetExecutionContext`.

Notes on JAR Layout

You don't need the Spring Cloud Function Web at runtime in Azure, so you can exclude this before you create the JAR you deploy to Azure, but it won't be used if you include it, so it doesn't hurt to leave it in. A function application on Azure is an archive generated by the Maven plugin. The function lives in the JAR file generated by this project. The sample creates it as an executable jar, using the thin layout, so that Azure can find the handler classes. If you prefer you can just use a regular flat JAR file. The dependencies should **not** be included.

Build file setup

In order to run Spring Cloud Function applications on Microsoft Azure, you can leverage the Maven plugin offered by the cloud platform provider.

In order to use the adapter plugin for Maven, add the plugin dependency to your `pom.xml` file:

```
<dependencies>
  <dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-function-adapter-azure</artifactId>
  </dependency>
</dependencies>
```

Then, configure the plugin. You will need to provide Azure-specific configuration for your application, specifying the `resourceGroup`, `appName` and other optional properties, and add the `package` goal execution so that the `function.json` file required by Azure is generated for you. Full plugin documentation can be found in the [plugin repository](#).

```
<plugin>
  <groupId>com.microsoft.azure</groupId>
  <artifactId>azure-functions-maven-plugin</artifactId>
  <configuration>
    <resourceGroup>${functionResourceGroup}</resourceGroup>
    <appName>${functionAppName}</appName>
  </configuration>
  <executions>
    <execution>
      <id>package-functions</id>
      <goals>
        <goal>package</goal>
      </goals>
    </execution>
  </executions>
</plugin>
```

You will also have to ensure that the files to be scanned by the plugin can be found in the Azure functions staging directory (see the [plugin repository](#) for more details on the staging directory and its default location).

You can find the entire sample `pom.xml` file for deploying Spring Cloud Function applications to Microsoft Azure with Maven [here](#).



As of yet, only Maven plugin is available. Gradle plugin has not been created by the cloud platform provider.

Build

```
./mvnw -U clean package
```

Running the sample

You can run the sample locally, just like the other Spring Cloud Function samples:

```
and curl -H "Content-Type: text/plain" localhost:8080/api/uppercase -d '{"value": "hello foobar"}'.
```

You will need the `az` CLI app (see docs.microsoft.com/en-us/azure/azure-functions/functions-create-first-java-maven for more detail). To deploy the function on Azure runtime:

```
$ az login
$ mvn azure-functions:deploy
```

On another terminal try this: `curl <azure-function-url-from-the-log>/api/uppercase -d '{"value": "hello foobar!"}'`. Please ensure that you use the right URL for the function above. Alternatively you can test the function in the Azure Dashboard UI (click on the function name, go to the right hand side and click "Test" and to the bottom right, "Run").

The input type for the function in the Azure sample is a Foo with a single property called "value". So you need this to test it with something like below:

```
{
  "value": "foobar"
}
```



The Azure sample app is written in the "non-functional" style (using `@Bean`). The functional style (with just `Function` or `ApplicationContextInitializer`) is much faster on startup in Azure than the traditional `@Bean` style, so if you don't need `@Beans` (or `@EnableAutoConfiguration`) it's a good choice. Warm starts are not affected. :branch: master

17.10.3. Google Cloud Functions (Alpha)

The Google Cloud Functions adapter enables Spring Cloud Function apps to run on the [Google Cloud Functions](#) serverless platform. You can either run the function locally using the open source [Google Functions Framework for Java](#) or on GCP.

Getting Started

Let's start with a simple Spring Cloud Function example:

```
@SpringBootApplication
public class CloudFunctionMain {

    public static void main(String[] args) {
        SpringApplication.run(CloudFunctionMain.class, args);
    }

    @Bean
    public Function<String, String> uppercase() {
        return value -> value.toUpperCase();
    }
}
```

Test locally

Start by adding the Maven plugin provided as part of the Google Functions Framework for Java.

```
<plugin>
  <groupId>com.google.cloud.functions</groupId>
  <artifactId>function-maven-plugin</artifactId>
  <version>0.9.1</version>
  <configuration>

  <functionTarget>org.springframework.cloud.function.adapter.gcloud.FunctionInvoker</functionTarget>
    <port>8080</port>
  </configuration>
</plugin>
```

Specify your configuration main class in `resources/META-INF/MANIFEST.MF`.

```
Main-Class: com.example.CloudFunctionMain
```

Then run the function:

```
mvn function:run
```

Invoke the HTTP function:

```
curl http://localhost:8080/ -d "hello"
```

Deploy to GCP

As of March 2020, Google Cloud Functions for Java is in Alpha. You can get on the [whitelist](#) to try it out.

First, add the Shade Plugin configuration to generate a fat jar when you run the `mvn package` command.

```

<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-shade-plugin</artifactId>
  <executions>
    <execution>
      <phase>package</phase>
      <goals>
        <goal>shade</goal>
      </goals>
      <configuration>
        <shadedArtifactAttached>>true</shadedArtifactAttached>
        <outputDirectory>deploy</outputDirectory>
        <shadedClassifierName>gcp</shadedClassifierName>
        <transformers>
          <transformer
implementation="org.apache.maven.plugins.shade.resource.AppendingTransformer">
            <resource>META-INF/spring.handlers</resource>
          </transformer>
          <transformer
implementation="org.springframework.boot.maven.PropertiesMergingResourceTransformer">
            <resource>META-INF/spring.factories</resource>
          </transformer>
          <transformer
implementation="org.apache.maven.plugins.shade.resource.AppendingTransformer">
            <resource>META-INF/spring.schemas</resource>
          </transformer>
          <transformer
implementation="org.apache.maven.plugins.shade.resource.ServicesResourceTransformer"/>
          <transformer
implementation="org.apache.maven.plugins.shade.resource.ManifestResourceTransformer">
            <mainClass>com.example.CloudFunctionMain</mainClass>
          </transformer>
        </transformers>
      </configuration>
    </execution>
  </executions>
</plugin>

```

Package the application.

```
mvn package
```

You should see the fat jar in `deploy` directory.

Make sure that you have the [Cloud SDK CLI](#) installed.

From the project base directory run the following command to deploy.


```
gcloud alpha functions deploy function-sample-gcp \  
--entry-point org.springframework.cloud.function.adapter.gcloud.FunctionInvoker \  
--runtime java11 \  
--trigger-http \  
--source deploy \  
--memory 512MB
```

Invoke the HTTP function:

```
curl https://REGION-PROJECT_ID.cloudfunctions.net/function-sample-gcp -d "hello"
```

Sample Function

Go to the [function-sample-gcp](#) to try out a sample function that you can test locally or deploy to GCP.

Chapter 18. Spring Cloud Kubernetes

This reference guide covers how to use Spring Cloud Kubernetes.

18.1. Why do you need Spring Cloud Kubernetes?

Spring Cloud Kubernetes provide Spring Cloud common interface implementations that consume Kubernetes native services. The main objective of the projects provided in this repository is to facilitate the integration of Spring Cloud and Spring Boot applications running inside Kubernetes.

18.2. Starters

Starters are convenient dependency descriptors you can include in your application. Include a starter to get the dependencies and Spring Boot auto-configuration for a feature set.

Starter	Features
<pre><dependency> <groupId>org.springframework.cloud</group pId> <artifactId>spring-cloud-starter- kubernetes</artifactId> </dependency></pre>	<p>Discovery Client implementation that resolves service names to Kubernetes Services.</p>
<pre><dependency> <groupId>org.springframework.cloud</group pId> <artifactId>spring-cloud-starter- kubernetes-config</artifactId> </dependency></pre>	<p>Load application properties from Kubernetes ConfigMaps and Secrets. Reload application properties when a ConfigMap or Secret changes.</p>
<pre><dependency> <groupId>org.springframework.cloud</group pId> <artifactId>spring-cloud-starter- kubernetes-all</artifactId> </dependency></pre>	<p>All Spring Cloud Kubernetes features.</p>

18.3. DiscoveryClient for Kubernetes

This project provides an implementation of [Discovery Client](#) for [Kubernetes](#). This client lets you query Kubernetes endpoints (see [services](#)) by name. A service is typically exposed by the Kubernetes API server as a collection of endpoints that represent [http](#) and [https](#) addresses and that a client can access from a Spring Boot application running as a pod.

This is something that you get for free by adding the following dependency inside your project:

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-kubernetes</artifactId>
</dependency>
```

To enable loading of the [DiscoveryClient](#), add [@EnableDiscoveryClient](#) to the according configuration or application class, as the following example shows:

```
@SpringBootApplication
@EnableDiscoveryClient
public class Application {
    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }
}
```

Then you can inject the client in your code simply by autowiring it, as the following example shows:

```
@Autowired
private DiscoveryClient discoveryClient;
```

You can choose to enable [DiscoveryClient](#) from all namespaces by setting the following property in [application.properties](#):

```
spring.cloud.kubernetes.discovery.all-namespaces=true
```

If, for any reason, you need to disable the [DiscoveryClient](#), you can set the following property in [application.properties](#):

```
spring.cloud.kubernetes.discovery.enabled=false
```

Some Spring Cloud components use the `DiscoveryClient` in order to obtain information about the local service instance. For this to work, you need to align the Kubernetes service name with the `spring.application.name` property.



`spring.application.name` has no effect as far as the name registered for the application within Kubernetes

Spring Cloud Kubernetes can also watch the Kubernetes service catalog for changes and update the `DiscoveryClient` implementation accordingly. In order to enable this functionality you need to add `@EnableScheduling` on a configuration class in your application.

18.4. Kubernetes native service discovery

Kubernetes itself is capable of (server side) service discovery (see: kubernetes.io/docs/concepts/services-networking/service/#discovering-services). Using native kubernetes service discovery ensures compatibility with additional tooling, such as Istio (istio.io), a service mesh that is capable of load balancing, circuit breaker, failover, and much more.

The caller service then need only refer to names resolvable in a particular Kubernetes cluster. A simple implementation might use a spring `RestTemplate` that refers to a fully qualified domain name (FQDN), such as `{service-name}.{namespace}.svc.{cluster}.local:{service-port}`.

Additionally, you can use Hystrix for:

- Circuit breaker implementation on the caller side, by annotating the spring boot application class with `@EnableCircuitBreaker`
- Fallback functionality, by annotating the respective method with `@HystrixCommand(fallbackMethod=`

18.5. Kubernetes PropertySource implementations

The most common approach to configuring your Spring Boot application is to create an `application.properties` or `application.yaml` or an `application-profile.properties` or `application-profile.yaml` file that contains key-value pairs that provide customization values to your application or Spring Boot starters. You can override these properties by specifying system properties or environment variables.

18.5.1. Using a `ConfigMap` PropertySource

Kubernetes provides a resource named `ConfigMap` to externalize the parameters to pass to your application in the form of key-value pairs or embedded `application.properties` or `application.yaml` files. The [Spring Cloud Kubernetes Config](#) project makes Kubernetes `ConfigMap` instances available during application bootstrapping and triggers hot reloading of beans or Spring context when

changes are detected on observed `ConfigMap` instances.

The default behavior is to create a `ConfigMapPropertySource` based on a Kubernetes `ConfigMap` that has a `metadata.name` value of either the name of your Spring application (as defined by its `spring.application.name` property) or a custom name defined within the `bootstrap.properties` file under the following key: `spring.cloud.kubernetes.config.name`.

However, more advanced configuration is possible where you can use multiple `ConfigMap` instances. The `spring.cloud.kubernetes.config.sources` list makes this possible. For example, you could define the following `ConfigMap` instances:

```
spring:
  application:
    name: cloud-k8s-app
  cloud:
    kubernetes:
      config:
        name: default-name
        namespace: default-namespace
        sources:
          # Spring Cloud Kubernetes looks up a ConfigMap named c1 in namespace
          default-namespace
          - name: c1
          # Spring Cloud Kubernetes looks up a ConfigMap named default-name in
          whatever namespace n2
          - namespace: n2
          # Spring Cloud Kubernetes looks up a ConfigMap named c3 in namespace n3
          - namespace: n3
            name: c3
```

In the preceding example, if `spring.cloud.kubernetes.config.namespace` had not been set, the `ConfigMap` named `c1` would be looked up in the namespace that the application runs.

Any matching `ConfigMap` that is found is processed as follows:

- Apply individual configuration properties.
- Apply as `yaml` the content of any property named `application.yaml`.
- Apply as a properties file the content of any property named `application.properties`.

The single exception to the aforementioned flow is when the `ConfigMap` contains a **single** key that indicates the file is a YAML or properties file. In that case, the name of the key does NOT have to be `application.yaml` or `application.properties` (it can be anything) and the value of the property is treated correctly. This feature facilitates the use case where the `ConfigMap` was created by using something like the following:

```
kubectl create configmap game-config --from-file=/path/to/app-config.yaml
```

Assume that we have a Spring Boot application named `demo` that uses the following properties to read its thread pool configuration.

- `pool.size.core`
- `pool.size.maximum`

This can be externalized to config map in `yaml` format as follows:

```
kind: ConfigMap
apiVersion: v1
metadata:
  name: demo
data:
  pool.size.core: 1
  pool.size.max: 16
```

Individual properties work fine for most cases. However, sometimes, embedded `yaml` is more convenient. In this case, we use a single property named `application.yaml` to embed our `yaml`, as follows:

```
kind: ConfigMap
apiVersion: v1
metadata:
  name: demo
data:
  application.yaml: |-
    pool:
      size:
        core: 1
        max:16
```

The following example also works:

```
kind: ConfigMap
apiVersion: v1
metadata:
  name: demo
data:
  custom-name.yaml: |-
    pool:
      size:
        core: 1
        max:16
```

You can also configure Spring Boot applications differently depending on active profiles that are merged together when the `ConfigMap` is read. You can provide different property values for different profiles by using an `application.properties` or `application.yaml` property, specifying profile-specific values, each in their own document (indicated by the `---` sequence), as follows:

```
kind: ConfigMap
apiVersion: v1
metadata:
  name: demo
data:
  application.yaml: |-
    greeting:
      message: Say Hello to the World
    farewell:
      message: Say Goodbye
    ---
    spring:
      profiles: development
    greeting:
      message: Say Hello to the Developers
    farewell:
      message: Say Goodbye to the Developers
    ---
    spring:
      profiles: production
    greeting:
      message: Say Hello to the Ops
```

In the preceding case, the configuration loaded into your Spring Application with the `development` profile is as follows:

```
greeting:
  message: Say Hello to the Developers
farewell:
  message: Say Goodbye to the Developers
```

However, if the `production` profile is active, the configuration becomes:

```
greeting:
  message: Say Hello to the Ops
farewell:
  message: Say Goodbye
```

If both profiles are active, the property that appears last within the `ConfigMap` overwrites any preceding values.

Another option is to create a different config map per profile and spring boot will automatically fetch it based on active profiles

```
kind: ConfigMap
apiVersion: v1
metadata:
  name: demo
data:
  application.yml: |-
    greeting:
      message: Say Hello to the World
    farewell:
      message: Say Goodbye
```



```
kind: ConfigMap
apiVersion: v1
metadata:
  name: demo-development
data:
  application.yml: |-
    spring:
      profiles: development
    greeting:
      message: Say Hello to the Developers
    farewell:
      message: Say Goodbye to the Developers
```

```
kind: ConfigMap
apiVersion: v1
metadata:
  name: demo-production
data:
  application.yml: |-
    spring:
      profiles: production
    greeting:
      message: Say Hello to the Ops
    farewell:
      message: Say Goodbye
```

To tell Spring Boot which **profile** should be enabled at bootstrap, you can pass **SPRING_PROFILES_ACTIVE** environment variable. To do so, you can launch your Spring Boot application with an environment variable that you can define it in the PodSpec at the container specification. Deployment resource file, as follows:

```

apiVersion: apps/v1
kind: Deployment
metadata:
  name: deployment-name
  labels:
    app: deployment-name
spec:
  replicas: 1
  selector:
    matchLabels:
      app: deployment-name
  template:
    metadata:
      labels:
        app: deployment-name
    spec:
      containers:
        - name: container-name
          image: your-image
          env:
            - name: SPRING_PROFILES_ACTIVE
              value: "development"

```



You should check the security configuration section. To access config maps from inside a pod you need to have the correct Kubernetes service accounts, roles and role bindings.

Another option for using `ConfigMap` instances is to mount them into the Pod by running the Spring Cloud Kubernetes application and having Spring Cloud Kubernetes read them from the file system. This behavior is controlled by the `spring.cloud.kubernetes.config.paths` property. You can use it in addition to or instead of the mechanism described earlier. You can specify multiple (exact) file paths in `spring.cloud.kubernetes.config.paths` by using the `,` delimiter.



You have to provide the full exact path to each property file, because directories are not being recursively parsed.

Table 10. Properties:

Name	Type	Default	Description
<code>spring.cloud.kubernetes.config.enabled</code>	Boolean	true	Enable ConfigMaps PropertySource
<code>spring.cloud.kubernetes.config.name</code>	String	<code>\${spring.application.name}</code>	Sets the name of ConfigMap to look up

Name	Type	Default	Description
<code>spring.cloud.kubernetes.config.namespace</code>	String	Client namespace	Sets the Kubernetes namespace where to lookup
<code>spring.cloud.kubernetes.config.paths</code>	List	null	Sets the paths where <code>ConfigMap</code> instances are mounted
<code>spring.cloud.kubernetes.config.enableApi</code>	Boolean	true	Enable or disable consuming <code>ConfigMap</code> instances through APIs

18.5.2. Secrets PropertySource

Kubernetes has the notion of [Secrets](#) for storing sensitive data such as passwords, OAuth tokens, and so on. This project provides integration with `Secrets` to make secrets accessible by Spring Boot applications. You can explicitly enable or disable This feature by setting the `spring.cloud.kubernetes.secrets.enabled` property.

When enabled, the `SecretsPropertySource` looks up Kubernetes for `Secrets` from the following sources:

1. Reading recursively from secrets mounts
2. Named after the application (as defined by `spring.application.name`)
3. Matching some labels

Note:

By default, consuming Secrets through the API (points 2 and 3 above) **is not enabled** for security reasons. The permission 'list' on secrets allows clients to inspect secrets values in the specified namespace. Further, we recommend that containers share secrets through mounted volumes.

If you enable consuming Secrets through the API, we recommend that you limit access to Secrets by using an authorization policy, such as RBAC. For more information about risks and best practices when consuming Secrets through the API refer to [this doc](#).

If the secrets are found, their data is made available to the application.

Assume that we have a spring boot application named `demo` that uses properties to read its database configuration. We can create a Kubernetes secret by using the following command:

```
kubectl create secret generic db-secret --from-literal=username=user --from-literal=password=p455w0rd
```

The preceding command would create the following secret (which you can see by using `kubectl get secrets db-secret -o yaml`):

```
apiVersion: v1
data:
  password: cDQ1NXcwcmQ=
  username: dXNlcmQ=
kind: Secret
metadata:
  creationTimestamp: 2017-07-04T09:15:57Z
  name: db-secret
  namespace: default
  resourceVersion: "357496"
  selfLink: /api/v1/namespaces/default/secrets/db-secret
  uid: 63c89263-6099-11e7-b3da-76d6186905a8
type: Opaque
```

Note that the data contains Base64-encoded versions of the literal provided by the `create` command.

Your application can then use this secret—for example, by exporting the secret's value as environment variables:

```
apiVersion: v1
kind: Deployment
metadata:
  name: ${project.artifactId}
spec:
  template:
    spec:
      containers:
      - env:
        - name: DB_USERNAME
          valueFrom:
            secretKeyRef:
              name: db-secret
              key: username
        - name: DB_PASSWORD
          valueFrom:
            secretKeyRef:
              name: db-secret
              key: password
```

You can select the Secrets to consume in a number of ways:

1. By listing the directories where secrets are mapped:

```
-Dspring.cloud.kubernetes.secrets.paths=/etc/secrets/db  
-secret,etc/secrets/postgresql
```

If you have all the secrets mapped to a common root, you can set them like:

```
-Dspring.cloud.kubernetes.secrets.paths=/etc/secrets
```

2. By setting a named secret:

```
-Dspring.cloud.kubernetes.secrets.name=db-secret
```

3. By defining a list of labels:

```
-Dspring.cloud.kubernetes.secrets.labels.broker=activemq  
-Dspring.cloud.kubernetes.secrets.labels.db=postgresql
```

As the case with `ConfigMap`, more advanced configuration is also possible where you can use multiple `Secret` instances. The `spring.cloud.kubernetes.secrets.sources` list makes this possible. For example, you could define the following `Secret` instances:

```

spring:
  application:
    name: cloud-k8s-app
  cloud:
    kubernetes:
      secrets:
        name: default-name
        namespace: default-namespace
        sources:
          # Spring Cloud Kubernetes looks up a Secret named s1 in namespace
          default-namespace
          - name: s1
          # Spring Cloud Kubernetes looks up a Secret named default-name in
          whatever namespace n2
          - namespace: n2
          # Spring Cloud Kubernetes looks up a Secret named s3 in namespace n3
          - namespace: n3
            name: s3

```

In the preceding example, if `spring.cloud.kubernetes.secrets.namespace` had not been set, the **Secret** named `s1` would be looked up in the namespace that the application runs.

Table 11. Properties:

Name	Type	Default	Description
<code>spring.cloud.kubernetes.secrets.enabled</code>	Boolean	<code>true</code>	Enable Secrets <code>PropertySource</code>
<code>spring.cloud.kubernetes.secrets.name</code>	String	<code>\${spring.application.name}</code>	Sets the name of the secret to look up
<code>spring.cloud.kubernetes.secrets.namespace</code>	String	Client namespace	Sets the Kubernetes namespace where to look up
<code>spring.cloud.kubernetes.secrets.labels</code>	Map	<code>null</code>	Sets the labels used to lookup secrets
<code>spring.cloud.kubernetes.secrets.paths</code>	List	<code>null</code>	Sets the paths where secrets are mounted (example 1)
<code>spring.cloud.kubernetes.secrets.enableApi</code>	Boolean	<code>false</code>	Enables or disables consuming secrets through APIs (examples 2 and 3)

Notes:

- The `spring.cloud.kubernetes.secrets.labels` property behaves as defined by [Map-based](#)

[binding](#).

- The `spring.cloud.kubernetes.secrets.paths` property behaves as defined by [Collection-based binding](#).
- Access to secrets through the API may be restricted for security reasons. The preferred way is to mount secrets to the Pod.

You can find an example of an application that uses secrets (though it has not been updated to use the new `spring-cloud-kubernetes` project) at [spring-boot-camel-config](#)

18.5.3. PropertySource Reload

Some applications may need to detect changes on external property sources and update their internal status to reflect the new configuration. The reload feature of Spring Cloud Kubernetes is able to trigger an application reload when a related `ConfigMap` or `Secret` changes.

By default, this feature is disabled. You can enable it by using the `spring.cloud.kubernetes.reload.enabled=true` configuration property (for example, in the `application.properties` file).

The following levels of reload are supported (by setting the `spring.cloud.kubernetes.reload.strategy` property):

- * `refresh` (default): Only configuration beans annotated with `@ConfigurationProperties` or `@RefreshScope` are reloaded. This reload level leverages the refresh feature of Spring Cloud Context.
- * `restart_context`: the whole Spring `ApplicationContext` is gracefully restarted. Beans are recreated with the new configuration.
- * `shutdown`: the Spring `ApplicationContext` is shut down to activate a restart of the container. When you use this level, make sure that the lifecycle of all non-daemon threads is bound to the `ApplicationContext` and that a replication controller or replica set is configured to restart the pod.

Assuming that the reload feature is enabled with default settings (`refresh` mode), the following bean is refreshed when the config map changes:

```
@Configuration
@ConfigurationProperties(prefix = "bean")
public class MyConfig {

    private String message = "a message that can be changed live";

    // getter and setters

}
```

To see that changes effectively happen, you can create another bean that prints the message periodically, as follows

```

@Component
public class MyBean {

    @Autowired
    private MyConfig config;

    @Scheduled(fixedDelay = 5000)
    public void hello() {
        System.out.println("The message is: " + config.getMessage());
    }
}

```

You can change the message printed by the application by using a `ConfigMap`, as follows:

```

apiVersion: v1
kind: ConfigMap
metadata:
  name: reload-example
data:
  application.properties: |-
    bean.message=Hello World!

```

Any change to the property named `bean.message` in the `ConfigMap` associated with the pod is reflected in the output. More generally speaking, changes associated to properties prefixed with the value defined by the `prefix` field of the `@ConfigurationProperties` annotation are detected and reflected in the application. [Associating a ConfigMap with a pod](#) is explained earlier in this chapter.

The full example is available in [spring-cloud-kubernetes-reload-example](#).

The reload feature supports two operating modes: * **Event (default)**: Watches for changes in config maps or secrets by using the Kubernetes API (web socket). Any event produces a re-check on the configuration and, in case of changes, a reload. The `view` role on the service account is required in order to listen for config map changes. A higher level role (such as `edit`) is required for secrets (by default, secrets are not monitored). * **Polling**: Periodically re-creates the configuration from config maps and secrets to see if it has changed. You can configure the polling period by using the `spring.cloud.kubernetes.reload.period` property and defaults to 15 seconds. It requires the same role as the monitored property source. This means, for example, that using polling on file-mounted secret sources does not require particular privileges.

Table 12. Properties:

Name	Type	Default	Description
<code>spring.cloud.kubernetes.reload.enabled</code>	Boolean	false	Enables monitoring of property sources and configuration reload
<code>spring.cloud.kubernetes.reload.monitoring-config-maps</code>	Boolean	true	Allow monitoring changes in config maps
<code>spring.cloud.kubernetes.reload.monitoring-secrets</code>	Boolean	false	Allow monitoring changes in secrets
<code>spring.cloud.kubernetes.reload.strategy</code>	Enum	refresh	The strategy to use when firing a reload (<code>refresh</code> , <code>restart_context</code> , or <code>shutdown</code>)
<code>spring.cloud.kubernetes.reload.mode</code>	Enum	event	Specifies how to listen for changes in property sources (<code>event</code> or <code>polling</code>)
<code>spring.cloud.kubernetes.reload.period</code>	Duration	15s	The period for verifying changes when using the <code>polling</code> strategy

Notes: * You should not use properties under `spring.cloud.kubernetes.reload` in config maps or secrets. Changing such properties at runtime may lead to unexpected results. * Deleting a property or the whole config map does not restore the original state of the beans when you use the `refresh` level.

18.6. Kubernetes Ecosystem Awareness

All of the features described earlier in this guide work equally well, regardless of whether your application is running inside Kubernetes. This is really helpful for development and troubleshooting. From a development point of view, this lets you start your Spring Boot application and debug one of the modules that is part of this project. You need not deploy it in Kubernetes, as the code of the project relies on the [Fabric8 Kubernetes Java client](#), which is a fluent DSL that can communicate by using `http` protocol to the REST API of the Kubernetes Server.

To disable the integration with Kubernetes you can set `spring.cloud.kubernetes.enabled` to `false`. Please be aware that when `spring-cloud-kubernetes-config` is on the classpath, `spring.cloud.kubernetes.enabled` should be set in `bootstrap.{properties|yaml}` (or the profile specific one) otherwise it should be in `application.{properties|yaml}` (or the profile specific one). Also note that these properties: `spring.cloud.kubernetes.config.enabled` and `spring.cloud.kubernetes.secrets.enabled` only take effect when set in `bootstrap.{properties|yaml}`

18.6.1. Kubernetes Profile Autoconfiguration

When the application runs as a pod inside Kubernetes, a Spring profile named `kubernetes` automatically gets activated. This lets you customize the configuration, to define beans that are applied when the Spring Boot application is deployed within the Kubernetes platform (for example, different development and production configuration).

18.6.2. Istio Awareness

When you include the `spring-cloud-kubernetes-istio` module in the application classpath, a new profile is added to the application, provided the application is running inside a Kubernetes Cluster with `Istio` installed. You can then use `spring @Profile("istio")` annotations in your Beans and `@Configuration` classes.

The Istio awareness module uses `me.snowdrop:istio-client` to interact with Istio APIs, letting us discover traffic rules, circuit breakers, and so on, making it easy for our Spring Boot applications to consume this data to dynamically configure themselves according to the environment.

18.7. Pod Health Indicator

Spring Boot uses `HealthIndicator` to expose info about the health of an application. That makes it really useful for exposing health-related information to the user and makes it a good fit for use as `readiness probes`.

The Kubernetes health indicator (which is part of the core module) exposes the following info:

- Pod name, IP address, namespace, service account, node name, and its IP address
- A flag that indicates whether the Spring Boot application is internal or external to Kubernetes

18.8. Leader Election

<TBD>

18.9. Security Configurations Inside Kubernetes

18.9.1. Namespace

Most of the components provided in this project need to know the namespace. For Kubernetes (1.3+), the namespace is made available to the pod as part of the service account secret and is automatically detected by the client. For earlier versions, it needs to be specified as an environment variable to the pod. A quick way to do this is as follows:

```
env:
  - name: "KUBERNETES_NAMESPACE"
    valueFrom:
      fieldRef:
        fieldPath: "metadata.namespace"
```

18.9.2. Service Account

For distributions of Kubernetes that support more fine-grained role-based access within the cluster, you need to make sure a pod that runs with `spring-cloud-kubernetes` has access to the Kubernetes API. For any service accounts you assign to a deployment or pod, you need to make sure they have the correct roles.

Depending on the requirements, you'll need `get`, `list` and `watch` permission on the following resources:

Table 13. Kubernetes Resource Permissions

Dependency	Resources
spring-cloud-starter-kubernetes	pods, services, endpoints
spring-cloud-starter-kubernetes-config	configmaps, secrets

For development purposes, you can add `cluster-reader` permissions to your `default` service account. On a production system you'll likely want to provide more granular permissions.

The following Role and RoleBinding are an example for namespaced permissions for the `default` account:

```

kind: Role
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  namespace: YOUR-NAME-SPACE
  name: namespace-reader
rules:
  - apiGroups: ["", "extensions", "apps"]
    resources: ["configmaps", "pods", "services", "endpoints", "secrets"]
    verbs: ["get", "list", "watch"]

---

kind: RoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: namespace-reader-binding
  namespace: YOUR-NAME-SPACE
subjects:
  - kind: ServiceAccount
    name: default
    apiGroup: ""
roleRef:
  kind: Role
  name: namespace-reader
  apiGroup: ""

```

18.10. Service Registry Implementation

In Kubernetes service registration is controlled by the platform, the application itself does not control registration as it may do in other platforms. For this reason using `spring.cloud.service-registry.auto-registration.enabled` or setting `@EnableDiscoveryClient(autoRegister=false)` will have no effect in Spring Cloud Kubernetes.

18.11. Examples

Spring Cloud Kubernetes tries to make it transparent for your applications to consume Kubernetes Native Services by following the Spring Cloud interfaces.

In your applications, you need to add the `spring-cloud-kubernetes-discovery` dependency to your classpath and remove any other dependency that contains a `DiscoveryClient` implementation (that is, a Eureka discovery client). The same applies for `PropertySourceLocator`, where you need to add to the classpath the `spring-cloud-kubernetes-config` and remove any other dependency that contains a `PropertySourceLocator` implementation (that is, a configuration server client).

The following projects highlight the usage of these dependencies and demonstrate how you can use these libraries from any Spring Boot application:

- [Spring Cloud Kubernetes Examples](#): the ones located inside this repository.
- Spring Cloud Kubernetes Full Example: Minions and Boss
 - [Minion](#)
 - [Boss](#)
- Spring Cloud Kubernetes Full Example: [SpringOne Platform Tickets Service](#)
- [Spring Cloud Gateway with Spring Cloud Kubernetes Discovery and Config](#)
- [Spring Boot Admin with Spring Cloud Kubernetes Discovery and Config](#)

18.12. Other Resources

This section lists other resources, such as presentations (slides) and videos about Spring Cloud Kubernetes.

- [S1P Spring Cloud on PKS](#)
- [Spring Cloud, Docker, Kubernetes → London Java Community July 2018](#)

Please feel free to submit other resources through pull requests to [this repository](#).

18.13. Configuration properties

To see the list of all Sleuth related configuration properties please check [the Appendix page](#).

18.14. Building

18.14.1. Basic Compile and Test

To build the source you will need to install JDK 1.7.

Spring Cloud uses Maven for most build-related activities, and you should be able to get off the ground quite quickly by cloning the project you are interested in and typing

```
$ ./mvnw install
```



You can also install Maven (>=3.3.3) yourself and run the `mvn` command in place of `./mvnw` in the examples below. If you do that you also might need to add `-P spring` if your local Maven settings do not contain repository declarations for spring pre-release artifacts.



Be aware that you might need to increase the amount of memory available to Maven by setting a `MAVEN_OPTS` environment variable with a value like `-Xmx512m -XX:MaxPermSize=128m`. We try to cover this in the `.mvn` configuration, so if you find you have to do it to make a build succeed, please raise a ticket to get the settings added to source control.

For hints on how to build the project look in `.travis.yml` if there is one. There should be a "script" and maybe "install" command. Also look at the "services" section to see if any services need to be running locally (e.g. mongo or rabbit). Ignore the git-related bits that you might find in "before_install" since they're related to setting git credentials and you already have those.

The projects that require middleware generally include a `docker-compose.yml`, so consider using [Docker Compose](#) to run the middleware servers in Docker containers. See the README in the [scripts demo repository](#) for specific instructions about the common cases of mongo, rabbit and redis.



If all else fails, build with the command from `.travis.yml` (usually `./mvnw install`).

18.14.2. Documentation

The spring-cloud-build module has a "docs" profile, and if you switch that on it will try to build asciidoc sources from `src/main/asciidoc`. As part of that process it will look for a `README.adoc` and process it by loading all the includes, but not parsing or rendering it, just copying it to `${main.basedir}` (defaults to `$/home/marcin/repo/spring-cloud-scripts`, i.e. the root of the project). If there are any changes in the README it will then show up after a Maven build as a modified file in the correct place. Just commit it and push the change.

18.14.3. Working with the code

If you don't have an IDE preference we would recommend that you use [Spring Tools Suite](#) or [Eclipse](#) when working with the code. We use the [m2eclipse](#) eclipse plugin for maven support. Other IDEs and tools should also work without issue as long as they use Maven 3.3.3 or better.

Activate the Spring Maven profile

Spring Cloud projects require the 'spring' Maven profile to be activated to resolve the spring milestone and snapshot repositories. Use your preferred IDE to set this profile to be active, or you may experience build errors.

Importing into eclipse with m2eclipse

We recommend the [m2eclipse](#) eclipse plugin when working with eclipse. If you don't already have m2eclipse installed it is available from the "eclipse marketplace".



Older versions of m2e do not support Maven 3.3, so once the projects are imported into Eclipse you will also need to tell m2eclipse to use the right profile for the projects. If you see many different errors related to the POMs in the projects, check that you have an up to date installation. If you can't upgrade m2e, add the "spring" profile to your `settings.xml`. Alternatively you can copy the repository settings from the "spring" profile of the parent pom into your `settings.xml`.

Importing into eclipse without m2eclipse

If you prefer not to use m2eclipse you can generate eclipse project metadata using the following command:

```
$ ./mvnw eclipse:eclipse
```

The generated eclipse projects can be imported by selecting **import existing projects** from the **file** menu.

18.15. Contributing

Spring Cloud is released under the non-restrictive Apache 2.0 license, and follows a very standard Github development process, using Github tracker for issues and merging pull requests into master. If you want to contribute even something trivial please do not hesitate, but follow the guidelines below.

18.15.1. Sign the Contributor License Agreement

Before we accept a non-trivial patch or pull request we will need you to sign the [Contributor License Agreement](#). Signing the contributor's agreement does not grant anyone commit rights to the main repository, but it does mean that we can accept your contributions, and you will get an author credit if we do. Active contributors might be asked to join the core team, and given the ability to merge pull requests.

18.15.2. Code of Conduct

This project adheres to the Contributor Covenant [code of conduct](#). By participating, you are expected to uphold this code. Please report unacceptable behavior to spring-code-of-conduct@pivotal.io.

18.15.3. Code Conventions and Housekeeping

None of these is essential for a pull request, but they will all help. They can also be added after the original pull request but before a merge.

- Use the Spring Framework code format conventions. If you use Eclipse you can import formatter settings using the `eclipse-code-formatter.xml` file from the [Spring Cloud Build](#) project. If using IntelliJ, you can use the [Eclipse Code Formatter Plugin](#) to import the same file.
- Make sure all new `.java` files to have a simple Javadoc class comment with at least an `@author` tag identifying you, and preferably at least a paragraph on what the class is for.
- Add the ASF license header comment to all new `.java` files (copy from existing files in the project)
- Add yourself as an `@author` to the `.java` files that you modify substantially (more than cosmetic changes).
- Add some Javadocs and, if you change the namespace, some XSD doc elements.
- A few unit tests would help a lot as well — someone has to do it.
- If no-one else is using your branch, please rebase it against the current master (or other target branch in the main project).

- When writing a commit message please follow [these conventions](#), if you are fixing an existing issue please add `Fixes gh-XXXX` at the end of the commit message (where XXXX is the issue number).

18.15.4. Checkstyle

Spring Cloud Build comes with a set of checkstyle rules. You can find them in the `spring-cloud-build-tools` module. The most notable files under the module are:

`spring-cloud-build-tools/`

```
└── src
    ├── checkstyle
    │   ├── checkstyle-suppressions.xml ③
    │   └── main
    │       └── resources
    │           ├── checkstyle-header.txt ②
    │           └── checkstyle.xml ①
```

- ① Default Checkstyle rules
- ② File header setup
- ③ Default suppression rules

Checkstyle configuration

Checkstyle rules are **disabled by default**. To add checkstyle to your project just define the following properties and plugins.


```
<properties>
<maven-checkstyle-plugin.failOnError>true</maven-checkstyle-plugin.failOnError> ①
  <maven-checkstyle-plugin.failOnViolation>true
  </maven-checkstyle-plugin.failOnViolation> ②
  <maven-checkstyle-plugin.includeTestSourceDirectory>true
  </maven-checkstyle-plugin.includeTestSourceDirectory> ③
</properties>

<build>
  <plugins>
    <plugin> ④
      <groupId>io.spring.javaformat</groupId>
      <artifactId>spring-javaformat-maven-plugin</artifactId>
    </plugin>
    <plugin> ⑤
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-checkstyle-plugin</artifactId>
    </plugin>
  </plugins>

  <reporting>
    <plugins>
      <plugin> ⑤
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-checkstyle-plugin</artifactId>
      </plugin>
    </plugins>
  </reporting>
</build>
```

- ① Fails the build upon Checkstyle errors
- ② Fails the build upon Checkstyle violations
- ③ Checkstyle analyzes also the test sources
- ④ Add the Spring Java Format plugin that will reformat your code to pass most of the Checkstyle formatting rules
- ⑤ Add checkstyle plugin to your build and reporting phases

If you need to suppress some rules (e.g. line length needs to be longer), then it's enough for you to define a file under `${project.root}/src/checkstyle/checkstyle-suppressions.xml` with your suppressions. Example:

projectRoot/src/checkstyle/checkstyle-suppressions.xml

```
<?xml version="1.0"?>
<!DOCTYPE suppressions PUBLIC
    "-//Puppy Crawl//DTD Suppressions 1.1//EN"
    "https://www.puppycrawl.com/dtds/suppressions_1_1.dtd">
<suppressions>
    <suppress files=".*ConfigServerApplication\.java"
checks="HideUtilityClassConstructor"/>
    <suppress files=".*ConfigClientWatch\.java" checks="LineLengthCheck"/>
</suppressions>
```

It's advisable to copy the `${spring-cloud-build.rootFolder}/.editorconfig` and `${spring-cloud-build.rootFolder}/.springformat` to your project. That way, some default formatting rules will be applied. You can do so by running this script:

```
$ curl https://raw.githubusercontent.com/spring-cloud/spring-cloud-
build/master/.editorconfig -o .editorconfig
$ touch .springformat
```

18.15.5. IDE setup

IntelliJ IDEA

In order to setup IntelliJ you should import our coding conventions, inspection profiles and set up the checkstyle plugin. The following files can be found in the [Spring Cloud Build](#) project.

spring-cloud-build-tools/

```
├── src
│   ├── checkstyle
│   │   ├── checkstyle-suppressions.xml ③
│   │   └── main
│   │       ├── resources
│   │       │   ├── checkstyle-header.txt ②
│   │       │   ├── checkstyle.xml ①
│   │       │   └── intellij
│   │       │       ├── IntelliJ_Project_Defaults.xml ④
│   │       │       └── IntelliJ_Spring_Boot_Java_Conventions.xml ⑤
```

- ① Default Checkstyle rules
- ② File header setup
- ③ Default suppression rules
- ④ Project defaults for IntelliJ that apply most of Checkstyle rules
- ⑤ Project style conventions for IntelliJ that apply most of Checkstyle rules

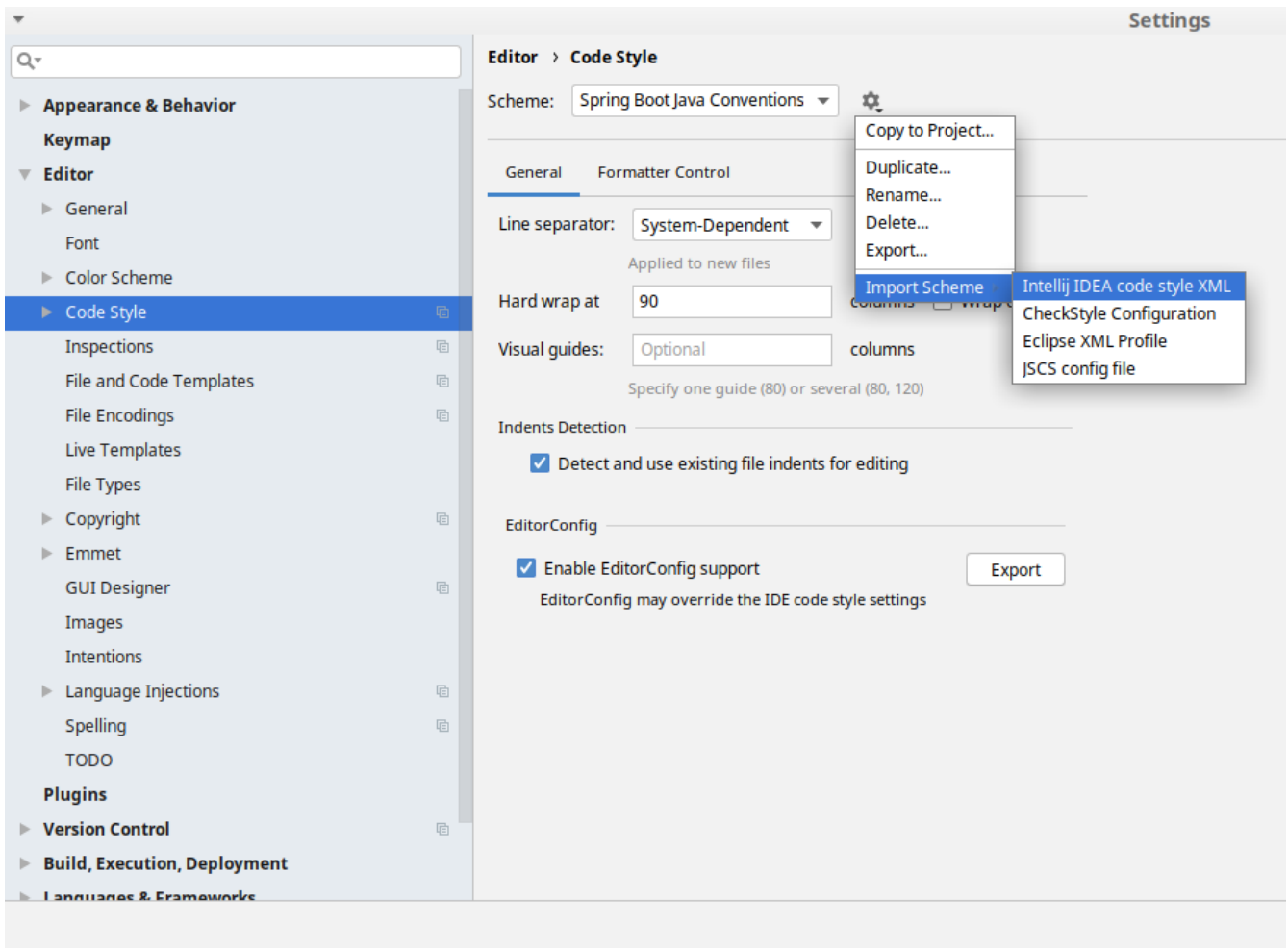


Figure 1. Code style

Go to **File** → **Settings** → **Editor** → **Code style**. There click on the icon next to the **Scheme** section. There, click on the **Import Scheme** value and pick the **IntelliJ IDEA code style XML** option. Import the `spring-cloud-build-tools/src/main/resources/intellij/IntelliJ_Spring_Boot_Java_Conventions.xml` file.

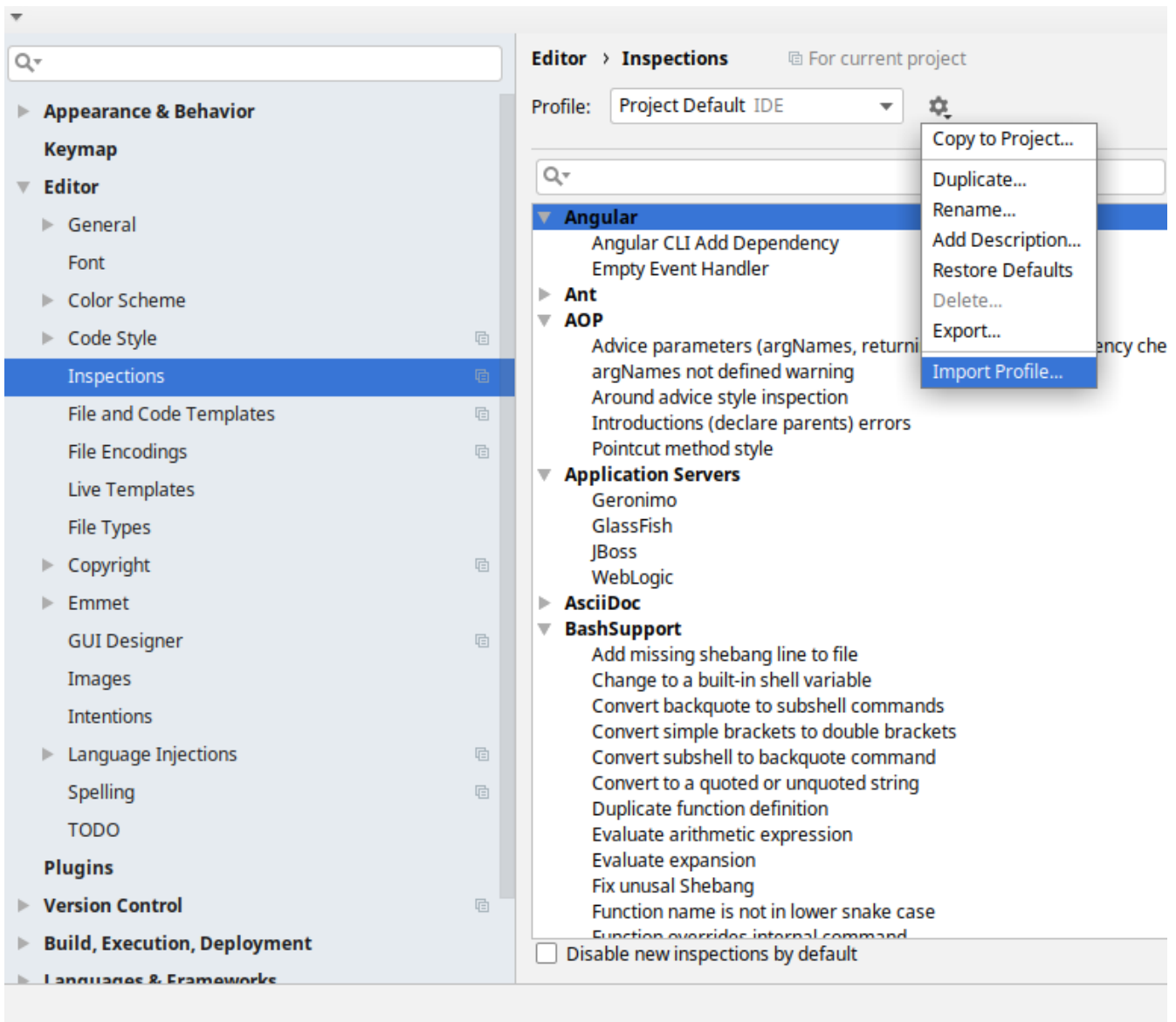
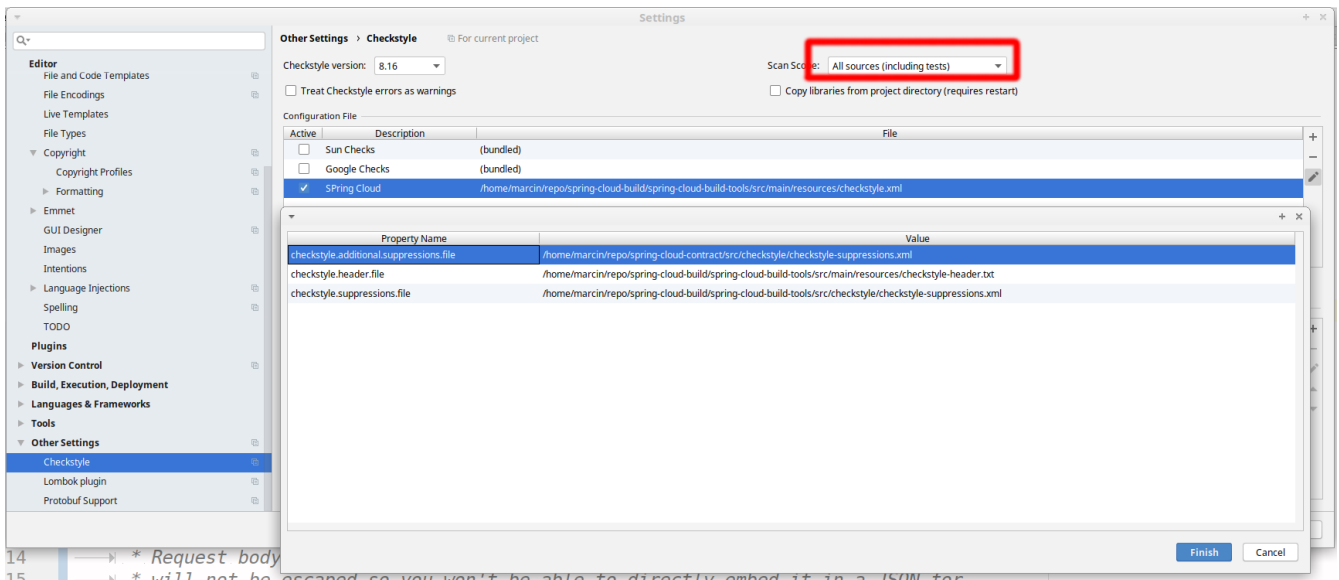


Figure 2. Inspection profiles

Go to **File** → **Settings** → **Editor** → **Inspections**. There click on the icon next to the **Profile** section. There, click on the **Import Profile** and import the `spring-cloud-build-tools/src/main/resources/intellij/Intellij_Project_Defaults.xml` file.

Checkstyle

To have IntelliJ work with Checkstyle, you have to install the **Checkstyle** plugin. It's advisable to also install the **Assertions2Assertj** to automatically convert the JUnit assertions



Go to **File** → **Settings** → **Other settings** → **Checkstyle**. There click on the **+** icon in the **Configuration file** section. There, you'll have to define where the checkstyle rules should be picked from. In the image above, we've picked the rules from the cloned Spring Cloud Build repository. However, you can point to the Spring Cloud Build's GitHub repository (e.g. for the `checkstyle.xml` : raw.githubusercontent.com/spring-cloud/spring-cloud-build/master/spring-cloud-build-tools/src/main/resources/checkstyle.xml). We need to provide the following variables:

- `checkstyle.header.file` - please point it to the Spring Cloud Build's, `spring-cloud-build-tools/src/main/resources/checkstyle-header.txt` file either in your cloned repo or via the raw.githubusercontent.com/spring-cloud/spring-cloud-build/master/spring-cloud-build-tools/src/main/resources/checkstyle-header.txt URL.
- `checkstyle.suppressions.file` - default suppressions. Please point it to the Spring Cloud Build's, `spring-cloud-build-tools/src/checkstyle/checkstyle-suppressions.xml` file either in your cloned repo or via the raw.githubusercontent.com/spring-cloud/spring-cloud-build/master/spring-cloud-build-tools/src/checkstyle/checkstyle-suppressions.xml URL.
- `checkstyle.additional.suppressions.file` - this variable corresponds to suppressions in your local project. E.g. you're working on `spring-cloud-contract`. Then point to the `project-root/src/checkstyle/checkstyle-suppressions.xml` folder. Example for `spring-cloud-contract` would be: `/home/username/spring-cloud-contract/src/checkstyle/checkstyle-suppressions.xml`.



Remember to set the **Scan Scope** to **All sources** since we apply checkstyle rules for production and test sources.

Chapter 19. Spring Cloud GCP

João André Martins; Jisha Abubaker; Ray Tsang; Mike Eltsufin; Artem Bilan; Andreas Berger; Balint Pato; Chengyuan Zhao; Dmitry Solomakha; Elena Felder; Daniel Zou, Eddú Meléndez

19.1. Introduction

The Spring Cloud GCP project makes the Spring Framework a first-class citizen of Google Cloud Platform (GCP).

Spring Cloud GCP lets you leverage the power and simplicity of the Spring Framework to:

- Publish and subscribe to Google Cloud Pub/Sub topics
- Configure Spring JDBC with a few properties to use Google Cloud SQL
- Map objects, relationships, and collections with Spring Data Cloud Spanner, Spring Data Cloud Datastore and Spring Data Reactive Repositories for Cloud Firestore
- Write and read from Spring Resources backed up by Google Cloud Storage
- Exchange messages with Spring Integration using Google Cloud Pub/Sub on the background
- Trace the execution of your app with Spring Cloud Sleuth and Google Stackdriver Trace
- Configure your app with Spring Cloud Config, backed up by the Google Runtime Configuration API
- Consume and produce Google Cloud Storage data via Spring Integration GCS Channel Adapters
- Use Spring Security via Google Cloud IAP
- Analyze your images for text, objects, and other content with Google Cloud Vision

19.2. Getting Started

This section describes how to get up to speed with Spring Cloud GCP libraries.

19.2.1. Setting up Dependencies

All Spring Cloud GCP artifacts are made available through Maven Central. The following resources are provided to help you setup the libraries for your project:

- Maven Bill of Materials for dependency management
- Starter Dependencies for depending on Spring Cloud GCP modules

You may also consult our [Github project](#) to examine the code or build directly from source.

Bill of Materials

The Spring Cloud GCP Bill of Materials (BOM) contains the versions of all the dependencies it uses.

If you're a Maven user, adding the following to your pom.xml file will allow you omit any Spring

Cloud GCP dependency version numbers from your configuration. Instead, the version of the BOM you're using determines the versions of the used dependencies.

```
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>org.springframework.cloud</groupId>
      <artifactId>spring-cloud-gcp-dependencies</artifactId>
      <version>{spring-cloud-gcp-version}</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
  </dependencies>
</dependencyManagement>
```

See the [sections](#) in the README for selecting an available version and Maven repository.

In the following sections, it will be assumed you are using the Spring Cloud GCP BOM and the dependency snippets will not contain versions.

Gradle users can achieve the same kind of BOM experience using Spring's [dependency-management-plugin](#) Gradle plugin. For simplicity, the Gradle dependency snippets in the remainder of this document will also omit their versions.

Starter Dependencies

Spring Cloud GCP offers [starter dependencies](#) through Maven to easily depend on different modules of the library. Each starter contains all the dependencies and transitive dependencies needed to begin using their corresponding Spring Cloud GCP module.

For example, if you wish to write a Spring application with Cloud Pub/Sub, you would include the [spring-cloud-gcp-starter-pubsub](#) dependency in your project. You do **not** need to include the underlying [spring-cloud-gcp-pubsub](#) dependency, because the [starter](#) dependency includes it.

A summary of these artifacts are provided below.

Spring Cloud GCP Starter	Description	Maven Artifact Name
Core	Automatically configure authentication and Google project settings	org.springframework.cloud:spring-cloud-gcp-starter
Cloud Spanner	Provides integrations with Google Cloud Spanner	org.springframework.cloud:spring-cloud-gcp-starter-data-spanner
Cloud Datastore	Provides integrations with Google Cloud Datastore	org.springframework.cloud:spring-cloud-gcp-starter-data-datastore

Spring Cloud GCP Starter	Description	Maven Artifact Name
Cloud Pub/Sub	Provides integrations with Google Cloud Pub/Sub	org.springframework.cloud:spring-cloud-gcp-starter-pubsub
Logging	Enables Stackdriver Logging	org.springframework.cloud:spring-cloud-gcp-starter-logging
SQL - MySQL	Cloud SQL integrations with MySQL	org.springframework.cloud:spring-cloud-gcp-starter-sql-mysql
SQL - PostgreSQL	Cloud SQL integrations with PostgreSQL	org.springframework.cloud:spring-cloud-gcp-starter-sql-postgresql
Storage	Provides integrations with Google Cloud Storage and Spring Resource	org.springframework.cloud:spring-cloud-gcp-starter-storage
Config	Enables usage of Google Runtime Configuration API as a Spring Cloud Config server	org.springframework.cloud:spring-cloud-gcp-starter-config
Trace	Enables instrumentation with Google Stackdriver Tracing	org.springframework.cloud:spring-cloud-gcp-starter-trace
Vision	Provides integrations with Google Cloud Vision	org.springframework.cloud:spring-cloud-gcp-starter-vision
Security - IAP	Provides a security layer over applications deployed to Google Cloud	org.springframework.cloud:spring-cloud-gcp-starter-security-iap

Spring Initializr

[Spring Initializr](#) is a tool which generates the scaffolding code for a new Spring Boot project. It handles the work of generating the Maven or Gradle build file so you do not have to manually add the dependencies yourself.

Spring Initializr offers three modules from Spring Cloud GCP that you can use to generate your project.

- **GCP Support:** The GCP Support module contains auto-configuration support for every Spring Cloud GCP integration. Most of the autoconfiguration code is only enabled if the required dependency is added to your project.
- **GCP Messaging:** Google Cloud Pub/Sub integrations work out of the box.
- **GCP Storage:** Google Cloud Storage integrations work out of the box.

19.2.2. Learning Spring Cloud GCP

There are a variety of resources to help you learn how to use Spring Cloud GCP libraries.

Sample Applications

The easiest way to learn how to use Spring Cloud GCP is to consult the [sample applications on Github](#). Spring Cloud GCP provides sample applications which demonstrate how to use every integration in the library. The table below highlights several samples of the most commonly used integrations in Spring Cloud GCP.

GCP Integration	Sample Application
Cloud Pub/Sub	spring-cloud-gcp-pubsub-sample
Cloud Spanner	spring-cloud-gcp-data-spanner-sample
Datastore	spring-cloud-gcp-data-datastore-sample
Cloud SQL (w/ MySQL)	spring-cloud-gcp-sql-mysql-sample
Cloud Storage	spring-cloud-gcp-storage-resource-sample
Stackdriver Logging	spring-cloud-gcp-logging-sample
Trace	spring-cloud-gcp-trace-sample
Cloud Vision	spring-cloud-gcp-vision-api-sample
Cloud Security - IAP	spring-cloud-gcp-security-iap-sample

Each sample application demonstrates how to use Spring Cloud GCP libraries in context and how to setup the dependencies for the project. The applications are fully functional and can be deployed to Google Cloud Platform as well. If you are interested, you may consult guides for [deploying an application to AppEngine](#) and [to Google Kubernetes Engine](#).

Codelabs

For a more hands-on approach, there are several guides and codelabs to help you get up to speed. These guides provide step-by-step instructions for building an application using Spring Cloud GCP.

Some examples include:

- [Deploy a Spring Boot app to App Engine](#)
- [Build a Kotlin Spring Boot app with Cloud SQL and Cloud Pub/Sub](#)
- [Build a Spring Boot application with Datastore](#)
- [Messaging with Spring Integration and Cloud Pub/Sub](#)

The full collection of Spring codelabs can be found on the [Google Developer Codelabs page](#).

19.3. Spring Cloud GCP Core

Each Spring Cloud GCP module uses `GcpProjectIdProvider` and `CredentialsProvider` to get the GCP project ID and access credentials.

Spring Cloud GCP provides a Spring Boot starter to auto-configure the core components.

Maven coordinates, using [Spring Cloud GCP BOM](#):

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-gcp-starter</artifactId>
</dependency>
```

Gradle coordinates:

```
dependencies {
  compile group: 'org.springframework.cloud', name: 'spring-cloud-gcp-starter'
}
```

19.3.1. Configuration

The following options may be configured with Spring Cloud core.

Name	Description	Required	Default value
<code>spring.cloud.gcp.core.enabled</code>	Enables or disables GCP core auto configuration	No	<code>true</code>

19.3.2. Project ID

`GcpProjectIdProvider` is a functional interface that returns a GCP project ID string.

```
public interface GcpProjectIdProvider {
  String getProjectId();
}
```

The Spring Cloud GCP starter auto-configures a `GcpProjectIdProvider`. If a `spring.cloud.gcp.project-id` property is specified, the provided `GcpProjectIdProvider` returns that property value.

```
spring.cloud.gcp.project-id=my-gcp-project-id
```

Otherwise, the project ID is discovered based on an [ordered list of rules](#):

1. The project ID specified by the `GOOGLE_CLOUD_PROJECT` environment variable
2. The Google App Engine project ID
3. The project ID specified in the JSON credentials file pointed by the `GOOGLE_APPLICATION_CREDENTIALS` environment variable
4. The Google Cloud SDK project ID
5. The Google Compute Engine project ID, from the Google Compute Engine Metadata Server

19.3.3. Credentials

`CredentialsProvider` is a functional interface that returns the credentials to authenticate and authorize calls to Google Cloud Client Libraries.

```
public interface CredentialsProvider {
    Credentials getCredentials() throws IOException;
}
```

The Spring Cloud GCP starter auto-configures a `CredentialsProvider`. It uses the `spring.cloud.gcp.credentials.location` property to locate the OAuth2 private key of a Google service account. Keep in mind this property is a Spring Resource, so the credentials file can be obtained from a number of [different locations](#) such as the file system, classpath, URL, etc. The next example specifies the credentials location property in the file system.

```
spring.cloud.gcp.credentials.location=file:/usr/local/key.json
```

Alternatively, you can set the credentials by directly specifying the `spring.cloud.gcp.credentials.encoded-key` property. The value should be the base64-encoded account private key in JSON format.

If that credentials aren't specified through properties, the starter tries to discover credentials from a [number of places](#):

1. Credentials file pointed to by the `GOOGLE_APPLICATION_CREDENTIALS` environment variable
2. Credentials provided by the Google Cloud SDK `gcloud auth application-default login` command
3. Google App Engine built-in credentials
4. Google Cloud Shell built-in credentials
5. Google Compute Engine built-in credentials

If your app is running on Google App Engine or Google Compute Engine, in most cases, you should omit the `spring.cloud.gcp.credentials.location` property and, instead, let the Spring Cloud GCP Starter get the correct credentials for those environments. On App Engine Standard, the [App Identity service account credentials](#) are used, on App Engine Flexible, the [Flexible service account credential](#) are used and on Google Compute Engine, the [Compute Engine Default Service Account](#) is used.

Scopes

By default, the credentials provided by the Spring Cloud GCP Starter contain scopes for every service supported by Spring Cloud GCP.

Service	Scope
Spanner	www.googleapis.com/auth/spanner.admin , www.googleapis.com/auth/spanner.data

Datastore	www.googleapis.com/auth/datastore
Pub/Sub	www.googleapis.com/auth/pubsub
Storage (Read Only)	www.googleapis.com/auth/devstorage.read_only
Storage (Write/Write)	www.googleapis.com/auth/devstorage.read_write
Runtime Config	www.googleapis.com/auth/cloudruntimeconfig
Trace (Append)	www.googleapis.com/auth/trace.append
Cloud Platform	www.googleapis.com/auth/cloud-platform
Vision	www.googleapis.com/auth/cloud-vision

The Spring Cloud GCP starter allows you to configure a custom scope list for the provided credentials. To do that, specify a comma-delimited list of [Google OAuth2 scopes](#) in the `spring.cloud.gcp.credentials.scopes` property.

`spring.cloud.gcp.credentials.scopes` is a comma-delimited list of [Google OAuth2 scopes](#) for Google Cloud Platform services that the credentials returned by the provided `CredentialsProvider` support.

```
spring.cloud.gcp.credentials.scopes=https://www.googleapis.com/auth/pubsub,https://www.googleapis.com/auth/sqlservice.admin
```

You can also use `DEFAULT_SCOPES` placeholder as a scope to represent the starters default scopes, and append the additional scopes you need to add.

```
spring.cloud.gcp.credentials.scopes=DEFAULT_SCOPES,https://www.googleapis.com/auth/cloud-vision
```

19.3.4. Environment

`GcpEnvironmentProvider` is a functional interface, auto-configured by the Spring Cloud GCP starter, that returns a `GcpEnvironment` enum. The provider can help determine programmatically in which GCP environment (App Engine Flexible, App Engine Standard, Kubernetes Engine or Compute Engine) the application is deployed.

```
public interface GcpEnvironmentProvider {
    GcpEnvironment getCurrentEnvironment();
}
```

19.3.5. Spring Initializr

This starter is available from [Spring Initializr](#) through the `GCP Support` entry.

19.4. Google Cloud Pub/Sub

Spring Cloud GCP provides an abstraction layer to publish to and subscribe from Google Cloud Pub/Sub topics and to create, list or delete Google Cloud Pub/Sub topics and subscriptions.

A Spring Boot starter is provided to auto-configure the various required Pub/Sub components.

Maven coordinates, using [Spring Cloud GCP BOM](#):

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-gcp-starter-pubsub</artifactId>
</dependency>
```

Gradle coordinates:

```
dependencies {
    compile group: 'org.springframework.cloud', name: 'spring-cloud-gcp-starter-
pubsub'
}
```

This starter is also available from [Spring Initializr](#) through the [GCP Messaging](#) entry.

19.4.1. Spring Boot Actuator Support

Cloud Pub/Sub Health Indicator

If you are using Spring Boot Actuator, you can take advantage of the Cloud Pub/Sub health indicator called `pubsub`. The health indicator will verify whether Cloud Pub/Sub is up and accessible by your application. To enable it, all you need to do is add the [Spring Boot Actuator](#) to your project. To disable the Cloud Pub/Sub indicator, set `management.health.pubsub.enabled` to `false`.

The `pubsub` indicator will then roll up to the overall application status visible at [localhost:8080/actuator/health](#) (use the `management.endpoint.health.show-details` property to view per-indicator details).

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
```

19.4.2. Sample

Sample applications for [using the template](#) and [using a subscription-backed reactive stream](#) are available.

19.4.3. Pub/Sub Operations & Template

`PubSubOperations` is an abstraction that allows Spring users to use Google Cloud Pub/Sub without depending on any Google Cloud Pub/Sub API semantics. It provides the common set of operations needed to interact with Google Cloud Pub/Sub. `PubSubTemplate` is the default implementation of `PubSubOperations` and it uses the [Google Cloud Java Client for Pub/Sub](#) to interact with Google Cloud Pub/Sub.

Publishing to a topic

`PubSubTemplate` provides asynchronous methods to publish messages to a Google Cloud Pub/Sub topic. The `publish()` method takes in a topic name to post the message to, a payload of a generic type and, optionally, a map with the message headers. The topic name could either be a canonical topic name within the current project, or the fully-qualified name referring to a topic in a different project using the `projects/<project_name>/topics/<topic_name>` format.

Here is an example of how to publish a message to a Google Cloud Pub/Sub topic:

```
Map<String, String> headers = Collections.singletonMap("key1", "val1");
pubSubTemplate.publish(topicName, "message", headers).get();
```

By default, the `SimplePubSubMessageConverter` is used to convert payloads of type `byte[]`, `ByteString`, `ByteBuffer`, and `String` to Pub/Sub messages.

Subscribing to a subscription

Google Cloud Pub/Sub allows many subscriptions to be associated to the same topic. `PubSubTemplate` allows you to listen to subscriptions via the `subscribe()` method. When listening to a subscription, messages will be pulled from Google Cloud Pub/Sub asynchronously and passed to a user provided message handler. The subscription name could either be a canonical subscription name within the current project, or the fully-qualified name referring to a subscription in a different project using the `projects/<project_name>/subscriptions/<subscription_name>` format.

Example

Subscribe to a subscription with a message handler:

```
Subscriber subscriber = pubSubTemplate.subscribe(subscriptionName, (message) -> {
    logger.info("Message received from " + subscriptionName + " subscription: "
        + message.getPubsSubMessage().getData().toStringUtf8());
    message.ack();
});
```

Subscribe methods

`PubSubTemplate` provides the following subscribe methods:

subscribe(String subscription, Consumer<Basic Acknowledgeable PubsubMessage> messageConsumer)	asynchronously pulls messages and passes them to <code>messageConsumer</code>
subscribeAndConvert(String subscription, Consumer<ConvertedBasicAcknowledgeablePubsubMessage<T>> messageConsumer, Class<T> payloadType)	same as <code>pull</code> , but converts message payload to <code>payloadType</code> using the converter configured in the template



As of version 1.2, subscribing by itself is not enough to keep an application running. For a command-line application, you may want to provide your own `ThreadPoolTaskScheduler` bean named `pubsubSubscriberThreadPool`, which by default creates non-daemon threads that will keep an application from stopping. This default behavior has been overridden in Spring Cloud GCP for consistency with Cloud Pub/Sub client library, and to avoid holding up command-line applications that would like to shut down once their work is done.

Pulling messages from a subscription

Google Cloud Pub/Sub supports synchronous pulling of messages from a subscription. This is different from subscribing to a subscription, in the sense that subscribing is an asynchronous task.

Example

Pull up to 10 messages:

```
int maxMessages = 10;
boolean returnImmediately = false;
List<AcknowledgeablePubsubMessage> messages = pubSubTemplate.pull(subscriptionName,
    maxMessages,
    returnImmediately);

//acknowledge the messages
pubSubTemplate.ack(messages);

messages.forEach(message ->
    logger.info(message.getPubsubMessage().getData().toStringUtf8()));
```

Pull methods

`PubSubTemplate` provides the following pull methods:

<code>pull(String subscription, Integer maxMessages, Boolean returnImmediately)</code>	<p>Pulls a number of messages from a subscription, allowing for the retry settings to be configured. Any messages received by <code>pull()</code> are not automatically acknowledged. See Acknowledging messages.</p> <p>The <code>maxMessages</code> parameter is the maximum limit of how many messages to pull from a subscription in a single call; this value must be greater than 0. You may omit this parameter by passing in <code>null</code>; this means there will be no limit on the number of messages pulled (<code>maxMessages</code> will be <code>Integer.MAX_INTEGER</code>).</p> <p>If <code>returnImmediately</code> is <code>true</code>, the system will respond immediately even if it there are no messages available to return in the <code>Pull</code> response. Otherwise, the system may wait (for a bounded amount of time) until at least one message is available, rather than returning no messages.</p>
<code>pullAndAck</code>	Works the same as the <code>pull</code> method and, additionally, acknowledges all received messages.
<code>pullNext</code>	Allows for a single message to be pulled and automatically acknowledged from a subscription.
<code>pullAndConvert</code>	Works the same as the <code>pull</code> method and, additionally, converts the Pub/Sub binary payload to an object of the desired type, using the converter configured in the template.

Acknowledging messages

There are two ways to acknowledge messages.

1. To acknowledge multiple messages at once, you can use the `PubSubTemplate.ack()` method. You can also use the `PubSubTemplate.nack()` for negatively acknowledging messages. Using these methods for acknowledging messages in batches is more efficient than acknowledging messages individually, but they **require** the collection of messages to be from the same project.
2. To acknowledge messages individually you can use the `ack()` or `nack()` method on each of them (to acknowledge or negatively acknowledge, correspondingly).



All `ack()`, `nack()`, and `modifyAckDeadline()` methods on messages as well as `PubSubSubscriberTemplate` are implemented asynchronously, returning a `ListenableFuture<Void>` to be able to process the asynchronous execution.

JSON support

For serialization and deserialization of POJOs using Jackson JSON, configure a `PubSubMessageConverter` bean, and the Spring Boot starter for GCP Pub/Sub will automatically wire it into the `PubSubTemplate`.


```
// Note: The ObjectMapper is used to convert Java POJOs to and from JSON.
// You will have to configure your own instance if you are unable to depend
// on the ObjectMapper provided by Spring Boot starters.
@Bean
public PubSubMessageConverter pubSubMessageConverter() {
    return new JacksonPubSubMessageConverter(new ObjectMapper());
}
```



Alternatively, you can set it directly by calling the `setMessageConverter()` method on the `PubSubTemplate`. Other implementations of the `PubSubMessageConverter` can also be configured in the same manner.

Assuming you have the following class defined:

```
static class TestUser {

    String username;

    String password;

    public String getUsername() {
        return this.username;
    }

    void setUsername(String username) {
        this.username = username;
    }

    public String getPassword() {
        return this.password;
    }

    void setPassword(String password) {
        this.password = password;
    }
}
```

You can serialize objects to JSON on publish automatically:

```
TestUser user = new TestUser();
user.setUsername("John");
user.setPassword("password");
pubSubTemplate.publish(topicName, user);
```

And that's how you convert messages to objects on pull:

```

int maxMessages = 1;
boolean returnImmediately = false;
List<ConvertedAcknowledgeablePubsubMessage<TestUser>> messages =
    pubSubTemplate.pullAndConvert(
        subscriptionName, maxMessages, returnImmediately, TestUser.class);

ConvertedAcknowledgeablePubsubMessage<TestUser> message = messages.get(0);

//acknowledge the message
message.ack();

TestUser receivedTestUser = message.getPayload();

```

Please refer to our [Pub/Sub JSON Payload Sample App](#) as a reference for using this functionality.

19.4.4. Reactive Stream Subscription

It is also possible to acquire a reactive stream backed by a subscription. To do so, a Project Reactor dependency (`io.projectreactor:reactor-core`) must be added to the project. The combination of the Pub/Sub starter and the Project Reactor dependencies will then make a `PubSubReactiveFactory` bean available, which can then be used to get a `Publisher`.

```

@Autowired
PubSubReactiveFactory reactiveFactory;

// ...

Flux<AcknowledgeablePubsubMessage> flux
    = reactiveFactory.poll("exampleSubscription", 1000);

```

The `Flux` then represents an infinite stream of GCP Pub/Sub messages coming in through the specified subscription. For unlimited demand, the Pub/Sub subscription will be polled regularly, at intervals determined by `pollingPeriodMs` parameter passed in when creating the `Flux`. For bounded demand, the `pollingPeriodMs` parameter is unused. Instead, as many messages as possible (up to the requested number) are delivered immediately, with the remaining messages delivered as they become available.

Any exceptions thrown by the underlying message retrieval logic will be passed as an error to the stream. The error handling operators (`Flux#retry()`, `Flux#onErrorResume()` etc.) can be used to recover.

The full range of Project Reactor operations can be applied to the stream. For example, if you only want to fetch 5 messages, you can use `limitRequest` operation to turn the infinite stream into a finite one:

```

Flux<AcknowledgeablePubsubMessage> fiveMessageFlux = flux.limitRequest(5);

```

Messages flowing through the **Flux** should be manually acknowledged.

```
flux.doOnNext(AcknowledgeablePubsubMessage::ack);
```

19.4.5. Pub/Sub management

PubSubAdmin is the abstraction provided by Spring Cloud GCP to manage Google Cloud Pub/Sub resources. It allows for the creation, deletion and listing of topics and subscriptions.



Generally when referring to topics and subscriptions, you can either use the short canonical name within the current project, or the fully-qualified name referring to a topic or subscription in a different project using the `projects/<project_name>/(<topics|subscriptions>/<name>` format.

PubSubAdmin depends on **GcpProjectIdProvider** and either a **CredentialsProvider** or a **TopicAdminClient** and a **SubscriptionAdminClient**. If given a **CredentialsProvider**, it creates a **TopicAdminClient** and a **SubscriptionAdminClient** with the Google Cloud Java Library for Pub/Sub default settings. The Spring Boot starter for GCP Pub/Sub auto-configures a **PubSubAdmin** object using the **GcpProjectIdProvider** and the **CredentialsProvider** auto-configured by the Spring Boot GCP Core starter.

Creating a topic

PubSubAdmin implements a method to create topics:

```
public Topic createTopic(String topicName)
```

Here is an example of how to create a Google Cloud Pub/Sub topic:

```
public void newTopic() {
    pubSubAdmin.createTopic("topicName");
}
```

Deleting a topic

PubSubAdmin implements a method to delete topics:

```
public void deleteTopic(String topicName)
```

Here is an example of how to delete a Google Cloud Pub/Sub topic:

```
public void deleteTopic() {
    pubSubAdmin.deleteTopic("topicName");
}
```

Listing topics

`PubSubAdmin` implements a method to list topics:

```
public List<Topic> listTopics
```

Here is an example of how to list every Google Cloud Pub/Sub topic name in a project:

```
List<String> topics = pubSubAdmin
    .listTopics()
    .stream()
    .map(Topic::getName)
    .collect(Collectors.toList());
```

Creating a subscription

`PubSubAdmin` implements a method to create subscriptions to existing topics:

```
public Subscription createSubscription(String subscriptionName, String topicName,
Integer ackDeadline, String pushEndpoint)
```

Here is an example of how to create a Google Cloud Pub/Sub subscription:

```
public void newSubscription() {
    pubSubAdmin.createSubscription("subscriptionName", "topicName", 10,
    "https://my.endpoint/push");
}
```

Alternative methods with default settings are provided for ease of use. The default value for `ackDeadline` is 10 seconds. If `pushEndpoint` isn't specified, the subscription uses message pulling, instead.

```
public Subscription createSubscription(String subscriptionName, String topicName)
```

```
public Subscription createSubscription(String subscriptionName, String topicName,
Integer ackDeadline)
```

```
public Subscription createSubscription(String subscriptionName, String topicName,
String pushEndpoint)
```

Deleting a subscription

`PubSubAdmin` implements a method to delete subscriptions:

```
public void deleteSubscription(String subscriptionName)
```

Here is an example of how to delete a Google Cloud Pub/Sub subscription:

```
public void deleteSubscription() {  
    pubSubAdmin.deleteSubscription("subscriptionName");  
}
```

Listing subscriptions

`PubSubAdmin` implements a method to list subscriptions:

```
public List<Subscription> listSubscriptions()
```

Here is an example of how to list every subscription name in a project:

```
List<String> subscriptions = pubSubAdmin  
    .listSubscriptions()  
    .stream()  
    .map(Subscription::getName)  
    .collect(Collectors.toList());
```

19.4.6. Configuration

The Spring Boot starter for Google Cloud Pub/Sub provides the following configuration options:

Name	Description	Required	Default value
<code>spring.cloud.gcp.pubsub.enabled</code>	Enables or disables Pub/Sub auto-configuration	No	<code>true</code>
<code>spring.cloud.gcp.pubsub.project-id</code>	GCP project ID where the Google Cloud Pub/Sub API is hosted, if different from the one in the Spring Cloud GCP Core Module	No	

<code>spring.cloud.gcp.pubsub.credentials.location</code>	OAuth2 credentials for authenticating with the Google Cloud Pub/Sub API, if different from the ones in the Spring Cloud GCP Core Module	No	
<code>spring.cloud.gcp.pubsub.emulator-host</code>	The host and port of the local running emulator. If provided, this will setup the client to connect against a running Google Cloud Pub/Sub Emulator .	No	
<code>spring.cloud.gcp.pubsub.credentials.encoded-key</code>	Base64-encoded contents of OAuth2 account private key for authenticating with the Google Cloud Pub/Sub API, if different from the ones in the Spring Cloud GCP Core Module	No	
<code>spring.cloud.gcp.pubsub.credentials.scopes</code>	OAuth2 scope for Spring Cloud GCP Pub/Sub credentials	No	www.googleapis.com/auth/pubsub
<code>spring.cloud.gcp.pubsub.keepAliveIntervalMinutes</code>	Determines frequency of keepalive gRPC ping	No	5 minutes
<code>spring.cloud.gcp.pubsub.subscriber.parallel-pull-count</code>	The number of pull workers	No	1
<code>spring.cloud.gcp.pubsub.subscriber.max-ack-extension-period</code>	The maximum period a message ack deadline will be extended, in seconds	No	0
<code>spring.cloud.gcp.pubsub.subscriber.pull-endpoint</code>	The endpoint for synchronous pulling messages	No	pubsub.googleapis.com:443
<code>spring.cloud.gcp.pubsub.[subscriber,publisher].executor-threads</code>	Number of threads used by <code>Subscriber</code> instances created by <code>SubscriberFactory</code>	No	4

<code>spring.cloud.gcp.pubsub.[subscriber,publisher].retry.total-timeout-seconds</code>	TotalTimeout has ultimate control over how long the logic should keep trying the remote call until it gives up completely. The higher the total timeout, the more retries can be attempted.	No	0
<code>spring.cloud.gcp.pubsub.[subscriber,publisher].retry.initial-retry-delay-second</code>	InitialRetryDelay controls the delay before the first retry. Subsequent retries will use this value adjusted according to the RetryDelayMultiplier.	No	0
<code>spring.cloud.gcp.pubsub.[subscriber,publisher].retry.retry-delay-multiplier</code>	RetryDelayMultiplier controls the change in retry delay. The retry delay of the previous call is multiplied by the RetryDelayMultiplier to calculate the retry delay for the next call.	No	1
<code>spring.cloud.gcp.pubsub.[subscriber,publisher].retry.max-retry-delay-seconds</code>	MaxRetryDelay puts a limit on the value of the retry delay, so that the RetryDelayMultiplier can't increase the retry delay higher than this amount.	No	0
<code>spring.cloud.gcp.pubsub.[subscriber,publisher].retry.max-attempts</code>	MaxAttempts defines the maximum number of attempts to perform. If this value is greater than 0, and the number of attempts reaches this limit, the logic will give up retrying even if the total retry time is still lower than TotalTimeout.	No	0

<code>spring.cloud.gcp.pubsub.[subscriber,publisher].retry.jittered</code>	Jitter determines if the delay time should be randomized.	No	true
<code>spring.cloud.gcp.pubsub.[subscriber,publisher].retry.initial-rpc-timeout-seconds</code>	InitialRpcTimeout controls the timeout for the initial RPC. Subsequent calls will use this value adjusted according to the RpcTimeoutMultiplier.	No	0
<code>spring.cloud.gcp.pubsub.[subscriber,publisher].retry.rpc-timeout-multiplier</code>	RpcTimeoutMultiplier controls the change in RPC timeout. The timeout of the previous call is multiplied by the RpcTimeoutMultiplier to calculate the timeout for the next call.	No	1
<code>spring.cloud.gcp.pubsub.[subscriber,publisher].retry.max-rpc-timeout-seconds</code>	MaxRpcTimeout puts a limit on the value of the RPC timeout, so that the RpcTimeoutMultiplier can't increase the RPC timeout higher than this amount.	No	0
<code>spring.cloud.gcp.pubsub.[subscriber,publisher.batching].flow-control.max-outstanding-element-count</code>	Maximum number of outstanding elements to keep in memory before enforcing flow control.	No	unlimited
<code>spring.cloud.gcp.pubsub.[subscriber,publisher.batching].flow-control.max-outstanding-request-bytes</code>	Maximum number of outstanding bytes to keep in memory before enforcing flow control.	No	unlimited
<code>spring.cloud.gcp.pubsub.[subscriber,publisher.batching].flow-control.limit-exceeded-behavior</code>	The behavior when the specified limits are exceeded.	No	Block
<code>spring.cloud.gcp.pubsub.publisher.batching.element-count-threshold</code>	The element count threshold to use for batching.	No	unset (threshold does not apply)

<code>spring.cloud.gcp.pubsub.publisher.batching.request-byte-threshold</code>	The request byte threshold to use for batching.	No	unset (threshold does not apply)
<code>spring.cloud.gcp.pubsub.publisher.batching.delay-threshold-seconds</code>	The delay threshold to use for batching. After this amount of time has elapsed (counting from the first element added), the elements will be wrapped up in a batch and sent.	No	unset (threshold does not apply)
<code>spring.cloud.gcp.pubsub.publisher.batching.enabled</code>	Enables batching.	No	false

19.5. Google Cloud Storage

[Google Cloud Storage](#) allows storing any types of files in single or multiple regions. A Spring Boot starter is provided to auto-configure the various Storage components.

Maven coordinates, using [Spring Cloud GCP BOM](#):

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-gcp-starter-storage</artifactId>
</dependency>
```

Gradle coordinates:

```
dependencies {
  compile group: 'org.springframework.cloud', name: 'spring-cloud-gcp-starter-storage'
}
```

This starter is also available from [Spring Initializr](#) through the [GCP Storage](#) entry.

19.5.1. Using Cloud Storage

The starter automatically configures and registers a [Storage](#) bean in the Spring application context. The [Storage](#) bean ([Javadoc](#)) can be used to list/create/update/delete buckets (a group of objects with similar permissions and resiliency requirements) and objects.

```

@Autowired
private Storage storage;

public void createFile() {
    Bucket bucket = storage.create(BucketInfo.of("my-app-storage-bucket"));

    storage.create(
        BlobInfo.newBuilder("my-app-storage-bucket", "subdirectory/my-file").build(),
        "file contents".getBytes()
    );
}

```

19.5.2. Cloud Storage Objects As Spring Resources

[Spring Resources](#) are an abstraction for a number of low-level resources, such as file system files, classpath files, servlet context-relative files, etc. Spring Cloud GCP adds a new resource type: a Google Cloud Storage (GCS) object.

The Spring Resource Abstraction for Google Cloud Storage allows GCS objects to be accessed by their GCS URL using the `@Value` annotation:

```

@Value("gs://[YOUR_GCS_BUCKET]/[GCS_FILE_NAME]")
private Resource gcsResource;

```

...or the Spring application context

```

SpringApplication.run(...).getResource("gs://[YOUR_GCS_BUCKET]/[GCS_FILE_NAME]");

```

This creates a `Resource` object that can be used to read the object, among [other possible operations](#).

It is also possible to write to a `Resource`, although a `WritableResource` is required.

```

@Value("gs://[YOUR_GCS_BUCKET]/[GCS_FILE_NAME]")
private Resource gcsResource;
...
try (OutputStream os = ((WritableResource) gcsResource).getOutputStream()) {
    os.write("foo".getBytes());
}

```

To work with the `Resource` as a Google Cloud Storage resource, cast it to `GoogleStorageResource`.

If the resource path refers to an object on Google Cloud Storage (as opposed to a bucket), then the `getBlob` method can be called to obtain a `Blob`. This type represents a GCS file, which has associated [metadata](#), such as content-type, that can be set. The `createSignedUrl` method can also be used to obtain [signed URLs](#) for GCS objects. However, creating signed URLs requires that the resource was created using service account credentials.

The Spring Boot Starter for Google Cloud Storage auto-configures the `Storage` bean required by the `spring-cloud-gcp-storage` module, based on the `CredentialsProvider` provided by the Spring Boot GCP starter.

Setting the Content Type

You can set the content-type of Google Cloud Storage files from their corresponding `Resource` objects:

```
((GoogleStorageResource)gcsResource).getBlob().toBuilder().setContentTypes("text/html").build().update();
```

19.5.3. Configuration

The Spring Boot Starter for Google Cloud Storage provides the following configuration options:

Name	Description	Required	Default value
<code>spring.cloud.gcp.storage.enabled</code>	Enables the GCP storage APIs.	No	<code>true</code>
<code>spring.cloud.gcp.storage.auto-create-files</code>	Creates files and buckets on Google Cloud Storage when writes are made to non-existent files	No	<code>true</code>
<code>spring.cloud.gcp.storage.credentials.location</code>	OAuth2 credentials for authenticating with the Google Cloud Storage API, if different from the ones in the Spring Cloud GCP Core Module	No	
<code>spring.cloud.gcp.storage.credentials.encoded-key</code>	Base64-encoded contents of OAuth2 account private key for authenticating with the Google Cloud Storage API, if different from the ones in the Spring Cloud GCP Core Module	No	
<code>spring.cloud.gcp.storage.credentials.scopes</code>	OAuth2 scope for Spring Cloud GCP Storage credentials	No	www.googleapis.com/auth/devstorage.read_write

19.5.4. Sample

A [sample application](#) and a [codelab](#) are available.

19.6. Spring JDBC

Spring Cloud GCP adds integrations with [Spring JDBC](#) so you can run your MySQL or PostgreSQL databases in Google Cloud SQL using Spring JDBC, or other libraries that depend on it like Spring Data JPA.

The Cloud SQL support is provided by Spring Cloud GCP in the form of two Spring Boot starters, one for MySQL and another one for PostgreSQL. The role of the starters is to read configuration from properties and assume default settings so that user experience connecting to MySQL and PostgreSQL is as simple as possible.

Maven coordinates, using [Spring Cloud GCP BOM](#):

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-gcp-starter-sql-mysql</artifactId>
</dependency>
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-gcp-starter-sql-postgresql</artifactId>
</dependency>
```

Gradle coordinates:

```
dependencies {
    compile group: 'org.springframework.cloud', name: 'spring-cloud-gcp-starter-sql-mysql'
    compile group: 'org.springframework.cloud', name: 'spring-cloud-gcp-starter-sql-postgresql'
}
```

19.6.1. Prerequisites

In order to use the Spring Boot Starters for Google Cloud SQL, the Google Cloud SQL API must be enabled in your GCP project.

To do that, go to the [API library page](#) of the Google Cloud Console, search for "Cloud SQL API", click the first result and enable the API.



There are several similar "Cloud SQL" results. You must access the "Google Cloud SQL API" one and enable the API from there.

19.6.2. Spring Boot Starter for Google Cloud SQL

The Spring Boot Starters for Google Cloud SQL provide an auto-configured `DataSource` object. Coupled with Spring JDBC, it provides a `JdbcTemplate` object bean that allows for operations such as querying and modifying a database.

```
public List<Map<String, Object>> listUsers() {
    return jdbcTemplate.queryForList("SELECT * FROM user;");
}
```

You can rely on [Spring Boot data source auto-configuration](#) to configure a `DataSource` bean. In other words, properties like the SQL username, `spring.datasource.username`, and password, `spring.datasource.password` can be used. There is also some configuration specific to Google Cloud SQL:

Property name	Description	Default value
<code>spring.cloud.gcp.sql.enabled</code>	Enables or disables Cloud SQL auto configuration	<code>true</code>
<code>spring.cloud.gcp.sql.database-name</code>	Name of the database to connect to.	
<code>spring.cloud.gcp.sql.instance-connection-name</code>	A string containing a Google Cloud SQL instance's project ID, region and name, each separated by a colon. For example, <code>my-project-id:my-region:my-instance-name</code> .	
<code>spring.cloud.gcp.sql.credentials.location</code>	File system path to the Google OAuth2 credentials private key file. Used to authenticate and authorize new connections to a Google Cloud SQL instance.	Default credentials provided by the Spring GCP Boot starter
<code>spring.cloud.gcp.sql.credentials.encoded-key</code>	Base64-encoded contents of OAuth2 account private key in JSON format. Used to authenticate and authorize new connections to a Google Cloud SQL instance.	Default credentials provided by the Spring GCP Boot starter



If you provide your own `spring.datasource.url`, it will be ignored, unless you disable Cloud SQL auto configuration with `spring.cloud.gcp.sql.enabled=false`.

DataSource creation flow

Based on the previous properties, the Spring Boot starter for Google Cloud SQL creates a `CloudSqlJdbcInfoProvider` object which is used to obtain an instance's JDBC URL and driver class name. If you provide your own `CloudSqlJdbcInfoProvider` bean, it is used instead and the properties related to building the JDBC URL or driver class are ignored.

The `DataSourceProperties` object provided by Spring Boot Autoconfigure is mutated in order to use the JDBC URL and driver class names provided by `CloudSqlJdbcInfoProvider`, unless those values were provided in the properties. It is in the `DataSourceProperties` mutation step that the credentials

factory is registered in a system property to be `SqlCredentialFactory`.

`DataSource` creation is delegated to `Spring Boot`. You can select the type of connection pool (e.g., Tomcat, HikariCP, etc.) by [adding their dependency to the classpath](#).

Using the created `DataSource` in conjunction with Spring JDBC provides you with a fully configured and operational `JdbcTemplate` object that you can use to interact with your SQL database. You can connect to your database with as little as a database and instance names.

Troubleshooting tips

Connection issues

If you're not able to connect to a database and see an endless loop of `Connecting to Cloud SQL instance [...] on IP [...]`, it's likely that exceptions are being thrown and logged at a level lower than your logger's level. This may be the case with HikariCP, if your logger is set to INFO or higher level.

To see what's going on in the background, you should add a `logback.xml` file to your application resources folder, that looks like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
  <include resource="org/springframework/boot/logging/logback/base.xml"/>
  <logger name="com.zaxxer.hikari.pool" level="DEBUG"/>
</configuration>
```

Errors like `c.g.cloud.sql.core.SslSocketFactory : Re-throwing cached exception due to attempt to refresh instance information too soon after error`

If you see a lot of errors like this in a loop and can't connect to your database, this is usually a symptom that something isn't right with the permissions of your credentials or the Google Cloud SQL API is not enabled. Verify that the Google Cloud SQL API is enabled in the Cloud Console and that your service account has the [necessary IAM roles](#).

To find out what's causing the issue, you can enable DEBUG logging level as mentioned [above](#).

PostgreSQL: `java.net.SocketException: already connected` **issue**

We found this exception to be common if your Maven project's parent is `spring-boot` version `1.5.x`, or in any other circumstance that would cause the version of the `org.postgresql:postgresql` dependency to be an older one (e.g., `9.4.1212.jre7`).

To fix this, re-declare the dependency in its correct version. For example, in Maven:

```
<dependency>
  <groupId>org.postgresql</groupId>
  <artifactId>postgresql</artifactId>
  <version>42.1.1</version>
</dependency>
```

19.6.3. Samples

Available sample applications and codelabs:

- [Spring Cloud GCP MySQL](#)
- [Spring Cloud GCP PostgreSQL](#)
- [Spring Data JPA with Spring Cloud GCP SQL](#)
- Codelab: [Spring Pet Clinic using Cloud SQL](#)

19.7. Spring Integration

Spring Cloud GCP provides Spring Integration adapters that allow your applications to use Enterprise Integration Patterns backed up by Google Cloud Platform services.

19.7.1. Channel Adapters for Cloud Pub/Sub

The channel adapters for Google Cloud Pub/Sub connect your Spring `MessageChannels` to Google Cloud Pub/Sub topics and subscriptions. This enables messaging between different processes, applications or micro-services backed up by Google Cloud Pub/Sub.

The Spring Integration Channel Adapters for Google Cloud Pub/Sub are included in the `spring-cloud-gcp-pubsub` module.

Maven coordinates, using [Spring Cloud GCP BOM](#):

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-gcp-pubsub</artifactId>
</dependency>
<dependency>
  <groupId>org.springframework.integration</groupId>
  <artifactId>spring-integration-core</artifactId>
</dependency>
```

Gradle coordinates:

```
dependencies {
    compile group: 'org.springframework.cloud', name: 'spring-cloud-gcp-pubsub'
    compile group: 'org.springframework.integration', name: 'spring-integration-core'
}
```

Inbound channel adapter (using Pub/Sub Streaming Pull)

`PubSubInboundChannelAdapter` is the inbound channel adapter for GCP Pub/Sub that listens to a GCP Pub/Sub subscription for new messages. It converts new messages to an internal Spring `Message` and then sends it to the bound output channel.

Google Pub/Sub treats message payloads as byte arrays. So, by default, the inbound channel adapter will construct the Spring `Message` with `byte[]` as the payload. However, you can change the desired payload type by setting the `payloadType` property of the `PubSubInboundChannelAdapter`. The `PubSubInboundChannelAdapter` delegates the conversion to the desired payload type to the `PubSubMessageConverter` configured in the `PubSubTemplate`.

To use the inbound channel adapter, a `PubSubInboundChannelAdapter` must be provided and configured on the user application side.

```
@Bean
public MessageChannel pubsubInputChannel() {
    return new PublishSubscribeChannel();
}

@Bean
public PubSubInboundChannelAdapter messageChannelAdapter(
    @Qualifier("pubsubInputChannel") MessageChannel inputChannel,
    PubSubSubscriberOperations subscriberOperations) {
    PubSubInboundChannelAdapter adapter =
        new PubSubInboundChannelAdapter(subscriberOperations, "subscriptionName");
    adapter.setOutputChannel(inputChannel);
    adapter.setAckMode(AckMode.MANUAL);

    return adapter;
}
```

In the example, we first specify the `MessageChannel` where the adapter is going to write incoming messages to. The `MessageChannel` implementation isn't important here. Depending on your use case, you might want to use a `MessageChannel` other than `PublishSubscribeChannel`.

Then, we declare a `PubSubInboundChannelAdapter` bean. It requires the channel we just created and a `SubscriberFactory`, which creates `Subscriber` objects from the Google Cloud Java Client for Pub/Sub. The Spring Boot starter for GCP Pub/Sub provides a configured `PubSubSubscriberOperations` object.

The `PubSubInboundChannelAdapter` supports three acknowledgement modes, with `AckMode.AUTO` being the default value;

Automatic acking (`AckMode.AUTO`)

A message is acked with GCP Pub/Sub if the adapter sent it to the channel and no exceptions were thrown.

If a `RuntimeException` is thrown while the message is processed, then the message is handled in the following way:

- If an error channel is configured for the adapter (`pubSubInboundChannelAdapter.getErrorChannel() != null`) then the message will be acked and forwarded with the error to the error channel.
- If no error channel is configured for the adapter, then the message is nacked.

Automatic acking OK (`AckMode.AUTO_ACK`)

A message is acked with GCP Pub/Sub if the adapter sent it to the channel and no exceptions were thrown. If a `RuntimeException` is thrown while the message is processed, then the message is neither acked / nor nacked.

This is useful when using the subscription's ack deadline timeout as a retry delivery backoff mechanism.

Manually acking (`AckMode.MANUAL`)

The adapter attaches a `BasicAcknowledgeablePubsubMessage` object to the `Message` headers. Users can extract the `BasicAcknowledgeablePubsubMessage` using the `GcpPubSubHeaders.ORIGINAL_MESSAGE` key and use it to (n)ack a message.

```
@Bean
@ServiceActivator(inputChannel = "pubsubInputChannel")
public MessageHandler messageReceiver() {
    return message -> {
        LOGGER.info("Message arrived! Payload: " + new String((byte[])
message.getPayload()));
        BasicAcknowledgeablePubsubMessage originalMessage =
            message.getHeaders().get(GcpPubSubHeaders.ORIGINAL_MESSAGE,
BasicAcknowledgeablePubsubMessage.class);
        originalMessage.ack();
    };
}
```

Pollable Message Source (using Pub/Sub Synchronous Pull)

While `PubSubInboundChannelAdapter`, through the underlying Asynchronous Pull Pub/Sub mechanism, provides the best performance for high-volume applications that receive a steady flow of messages, it can create load balancing anomalies due to message caching. This behavior is most obvious when publishing a large batch of small messages that take a long time to process individually. It manifests as one subscriber taking up most messages, even if multiple subscribers are available to take on the work. For a more detailed explanation of this scenario, see [GCP Pub/Sub](#)

[documentation](#).

In such a scenario, a `PubSubMessageSource` can help spread the load between different subscribers more evenly.

As with the Inbound Channel Adapter, the message source has a configurable acknowledgement mode, payload type, and header mapping.

The default behavior is to return from the synchronous pull operation immediately if no messages are present. This can be overridden by using `setBlockOnPull()` method to wait for at least one message to arrive.

By default, `PubSubMessageSource` pulls from the subscription one message at a time. To pull a batch of messages on each request, use the `setMaxFetchSize()` method to set the batch size.

```
@Bean
@InboundChannelAdapter(channel = "pubsubInputChannel", poller = @Poller(fixedDelay =
"100"))
public MessageSource<Object> pubsubAdapter(PubSubTemplate pubSubTemplate) {
    PubSubMessageSource messageSource = new PubSubMessageSource(pubSubTemplate,
"exampleSubscription");
    messageSource.setAckMode(AckMode.MANUAL);
    messageSource.setPayloadType(String.class);
    messageSource.setBlockOnPull(true);
    messageSource.setMaxFetchSize(100);
    return messageSource;
}
```

The `@InboundChannelAdapter` annotation above ensures that the configured `MessageSource` is polled for messages, which are then available for manipulation with any Spring Integration mechanism on the `pubsubInputChannel` message channel. For example, messages can be retrieved in a method annotated with `@ServiceActivator`, as seen below.

For additional flexibility, `PubSubMessageSource` attaches an `AcknowledgeablePubSubMessage` object to the `GcpPubSubHeaders.ORIGINAL_MESSAGE` message header. The object can be used for manually (n)acking the message.

```
@ServiceActivator(inputChannel = "pubsubInputChannel")
public void messageReceiver(String payload,
    @Header(GcpPubSubHeaders.ORIGINAL_MESSAGE) AcknowledgeablePubsubMessage
message)
    throws InterruptedException {
    LOGGER.info("Message arrived by Synchronous Pull! Payload: " + payload);
    message.ack();
}
```



`AcknowledgeablePubSubMessage` objects acquired by synchronous pull are aware of their own acknowledgement IDs. Streaming pull does not expose this information due to limitations of the underlying API, and returns `BasicAcknowledgeablePubsubMessage` objects that allow acking/nacking individual messages, but not extracting acknowledgement IDs for future processing.

Outbound channel adapter

`PubSubMessageHandler` is the outbound channel adapter for GCP Pub/Sub that listens for new messages on a Spring `MessageChannel`. It uses `PubSubTemplate` to post them to a GCP Pub/Sub topic.

To construct a Pub/Sub representation of the message, the outbound channel adapter needs to convert the Spring `Message` payload to a byte array representation expected by Pub/Sub. It delegates this conversion to the `PubSubTemplate`. To customize the conversion, you can specify a `PubSubMessageConverter` in the `PubSubTemplate` that should convert the `Object` payload and headers of the Spring `Message` to a `PubsubMessage`.

To use the outbound channel adapter, a `PubSubMessageHandler` bean must be provided and configured on the user application side.

```
@Bean
@ServiceActivator(inputChannel = "pubsubOutputChannel")
public MessageHandler messageSender(PubSubTemplate pubsubTemplate) {
    return new PubSubMessageHandler(pubsubTemplate, "topicName");
}
```

The provided `PubSubTemplate` contains all the necessary configuration to publish messages to a GCP Pub/Sub topic.

`PubSubMessageHandler` publishes messages asynchronously by default. A publish timeout can be configured for synchronous publishing. If none is provided, the adapter waits indefinitely for a response.

It is possible to set user-defined callbacks for the `publish()` call in `PubSubMessageHandler` through the `setPublishFutureCallback()` method. These are useful to process the message ID, in case of success, or the error if any was thrown.

To override the default destination you can use the `GcpPubSubHeaders.DESTINATION` header.

```

@Autowired
private MessageChannel pubsubOutputChannel;

public void handleMessage(Message<?> msg) throws MessagingException {
    final Message<?> message = MessageBuilder
        .withPayload(msg.getPayload())
        .setHeader(GcpPubSubHeaders.TOPIC, "customTopic").build();
    pubsubOutputChannel.send(message);
}

```

It is also possible to set an SpEL expression for the topic with the `setTopicExpression()` or `setTopicExpressionString()` methods.

Header mapping

These channel adapters contain header mappers that allow you to map, or filter out, headers from Spring to Google Cloud Pub/Sub messages, and vice-versa. By default, the inbound channel adapter maps every header on the Google Cloud Pub/Sub messages to the Spring messages produced by the adapter. The outbound channel adapter maps every header from Spring messages into Google Cloud Pub/Sub ones, except the ones added by Spring, like headers with key `"id"`, `"timestamp"` and `"gcp_pubsub_acknowledgement"`. In the process, the outbound mapper also converts the value of the headers into string.

Each adapter declares a `setHeaderMapper()` method to let you further customize which headers you want to map from Spring to Google Cloud Pub/Sub, and vice-versa.

For example, to filter out headers `"foo"`, `"bar"` and all headers starting with the prefix `"prefix_"`, you can use `setHeaderMapper()` along with the `PubSubHeaderMapper` implementation provided by this module.

```

PubSubMessageHandler adapter = ...
...
PubSubHeaderMapper headerMapper = new PubSubHeaderMapper();
headerMapper.setOutboundHeaderPatterns("!foo", "!bar", "!prefix_*", "*");
adapter.setHeaderMapper(headerMapper);

```



The order in which the patterns are declared in `PubSubHeaderMapper.setOutboundHeaderPatterns()` and `PubSubHeaderMapper.setInboundHeaderPatterns()` matters. The first patterns have precedence over the following ones.

In the previous example, the `"*"` pattern means every header is mapped. However, because it comes last in the list, [the previous patterns take precedence](#).

19.7.2. Sample

Available examples:

- [sender and receiver sample application](#)
- [JSON payloads sample application](#)
- [codelab](#)

19.7.3. Channel Adapters for Google Cloud Storage

The channel adapters for Google Cloud Storage allow you to read and write files to Google Cloud Storage through `MessageChannels`.

Spring Cloud GCP provides two inbound adapters, `GcsInboundFileSynchronizingMessageSource` and `GcsStreamingMessageSource`, and one outbound adapter, `GcsMessageHandler`.

The Spring Integration Channel Adapters for Google Cloud Storage are included in the `spring-cloud-gcp-storage` module.

To use the Storage portion of Spring Integration for Spring Cloud GCP, you must also provide the `spring-integration-file` dependency, since it isn't pulled transitively.

Maven coordinates, using [Spring Cloud GCP BOM](#):

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-gcp-storage</artifactId>
</dependency>
<dependency>
  <groupId>org.springframework.integration</groupId>
  <artifactId>spring-integration-file</artifactId>
</dependency>
```

Gradle coordinates:

```
dependencies {
  compile group: 'org.springframework.cloud', name: 'spring-cloud-gcp-starter-storage'
  compile group: 'org.springframework.integration', name: 'spring-integration-file'
}
```

Inbound channel adapter

The Google Cloud Storage inbound channel adapter polls a Google Cloud Storage bucket for new files and sends each of them in a `Message` payload to the `MessageChannel` specified in the `@InboundChannelAdapter` annotation. The files are temporarily stored in a folder in the local file system.

Here is an example of how to configure a Google Cloud Storage inbound channel adapter.

```

@Bean
@InboundChannelAdapter(channel = "new-file-channel", poller = @Poller(fixedDelay =
"5000"))
public MessageSource<File> synchronizerAdapter(Storage gcs) {
    GcsInboundFileSynchronizer synchronizer = new GcsInboundFileSynchronizer(gcs);
    synchronizer.setRemoteDirectory("your-gcs-bucket");

    GcsInboundFileSynchronizingMessageSource synchAdapter =
        new GcsInboundFileSynchronizingMessageSource(synchronizer);
    synchAdapter.setLocalDirectory(new File("local-directory"));

    return synchAdapter;
}

```

Inbound streaming channel adapter

The inbound streaming channel adapter is similar to the normal inbound channel adapter, except it does not require files to be stored in the file system.

Here is an example of how to configure a Google Cloud Storage inbound streaming channel adapter.

```

@Bean
@InboundChannelAdapter(channel = "streaming-channel", poller = @Poller(fixedDelay =
"5000"))
public MessageSource<InputStream> streamingAdapter(Storage gcs) {
    GcsStreamingMessageSource adapter =
        new GcsStreamingMessageSource(new GcsRemoteFileTemplate(new
GcsSessionFactory(gcs)));
    adapter.setRemoteDirectory("your-gcs-bucket");
    return adapter;
}

```

If you would like to process the files in your bucket in a specific order, you may pass in a `Comparator<BlobInfo>` to the constructor `GcsStreamingMessageSource` to sort the files being processed.

Outbound channel adapter

The outbound channel adapter allows files to be written to Google Cloud Storage. When it receives a `Message` containing a payload of type `File`, it writes that file to the Google Cloud Storage bucket specified in the adapter.

Here is an example of how to configure a Google Cloud Storage outbound channel adapter.

```

@Bean
@ServiceActivator(inputChannel = "writeFiles")
public MessageHandler outboundChannelAdapter(Storage gcs) {
    GcsMessageHandler outboundChannelAdapter = new GcsMessageHandler(new
GcsSessionFactory(gcs));
    outboundChannelAdapter.setRemoteDirectoryExpression(new ValueExpression<>("your-gcs-
bucket"));

    return outboundChannelAdapter;
}

```

19.7.4. Sample

A [sample application](#) is available.

19.8. Spring Cloud Stream

Spring Cloud GCP provides a [Spring Cloud Stream](#) binder to Google Cloud Pub/Sub.

The provided binder relies on the [Spring Integration Channel Adapters for Google Cloud Pub/Sub](#).

Maven coordinates, using [Spring Cloud GCP BOM](#):

```

<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-gcp-pubsub-stream-binder</artifactId>
</dependency>

```

Gradle coordinates:

```

dependencies {
    compile group: 'org.springframework.cloud', name: 'spring-cloud-gcp-pubsub-stream-
binder'
}

```

19.8.1. Overview

This binder binds producers to Google Cloud Pub/Sub topics and consumers to subscriptions.



Partitioning is currently not supported by this binder.

19.8.2. Configuration

You can configure the Spring Cloud Stream Binder for Google Cloud Pub/Sub to automatically generate the underlying resources, like the Google Cloud Pub/Sub topics and subscriptions for producers and consumers. For that, you can use the

`spring.cloud.stream.gcp.pubsub.bindings.<channelName>.<consumer|producer>.auto-create-resources` property, which is turned ON by default.

Starting with version 1.1, these and other binder properties can be configured globally for all the bindings, e.g. `spring.cloud.stream.gcp.pubsub.default.consumer.auto-create-resources`.

If you are using Pub/Sub auto-configuration from the Spring Cloud GCP Pub/Sub Starter, you should refer to the [configuration](#) section for other Pub/Sub parameters.



To use this binder with a [running emulator](#), configure its host and port via `spring.cloud.gcp.pubsub.emulator-host`.

Producer Destination Configuration

If automatic resource creation is turned ON and the topic corresponding to the destination name does not exist, it will be created.

For example, for the following configuration, a topic called `myEvents` would be created.

application.properties

```
spring.cloud.stream.bindings.events.destination=myEvents
spring.cloud.stream.gcp.pubsub.bindings.events.producer.auto-create-resources=true
```

Consumer Destination Configuration

A `PubSubInboundChannelAdapter` will be configured for your consumer endpoint. You may adjust the ack mode of the consumer endpoint using the `ack-mode` property. The ack mode controls how messages will be acknowledged when they are successfully received. The three possible options are: `AUTO` (default), `AUTO_ACK`, and `MANUAL`. These options are described in detail in the [Pub/Sub channel adapter documentation](#).

application.properties

```
# How to set the ACK mode of the consumer endpoint.
spring.cloud.stream.gcp.pubsub.bindings.{CONSUMER_NAME}.consumer.ack-mode=AUTO_ACK
```

If automatic resource creation is turned ON and the subscription and/or the topic do not exist for a consumer, a subscription and potentially a topic will be created. The topic name will be the same as the destination name, and the subscription name will be the destination name followed by the consumer group name.

Regardless of the `auto-create-resources` setting, if the consumer group is not specified, an anonymous one will be created with the name `anonymous.<destinationName>.<randomUUID>`. Then when the binder shuts down, all Pub/Sub subscriptions created for anonymous consumer groups will be automatically cleaned up.

For example, for the following configuration, a topic named `myEvents` and a subscription called `myEvents.consumerGroup1` would be created. If the consumer group is not specified, a subscription

called `anonymous.myEvents.a6d83782-c5a3-4861-ac38-e6e2af15a7be` would be created and later cleaned up.



If you are manually creating Pub/Sub subscriptions for consumers, make sure that they follow the naming convention of `<destinationName>.<consumerGroup>`.

application.properties

```
spring.cloud.stream.bindings.events.destination=myEvents
spring.cloud.stream.gcp.pubsub.bindings.events.consumer.auto-create-resources=true

# specify consumer group, and avoid anonymous consumer group generation
spring.cloud.stream.bindings.events.group=consumerGroup1
```

19.8.3. Binding with Functions

Since version 3.0, Spring Cloud Stream supports a functional programming model natively. This means that the only requirement for turning your application into a sink is presence of a `java.util.function.Consumer` bean in the application context.

```
@Bean
public Consumer<UserMessage> logUserMessage() {
    return userMessage -> {
        // process message
    }
};
```

A source application is one where a `Supplier` bean is present. It can return an object, in which case Spring Cloud Stream will invoke the supplier repeatedly. Alternatively, the function can return a reactive stream, which will be used as is.

```
@Bean
Supplier<Flux<UserMessage>> generateUserMessages() {
    return () -> /* flux creation logic */;
}
```

A processor application works similarly to a source application, except it is triggered by presence of a `Function` bean.

19.8.4. Binding with Annotations



As of version 3.0, annotation binding is considered legacy.

To set up a sink application in this style, you would associate a class with a binding interface, such as the built-in `Sink` interface.

```

@EnableBinding(Sink.class)
public class SinkExample {

    @StreamListener(Sink.INPUT)
    public void handleMessage(UserMessage userMessage) {
        // process message
    }
}

```

To set up a source application, you would similarly associate a class with a built-in `Source` interface, and inject an instance of it provided by Spring Cloud Stream.

```

@EnableBinding(Source.class)
public class SourceExample {

    @Autowired
    private Source source;

    public void sendMessage() {
        this.source.output().send(new GenericMessage<>(/* your object here */));
    }
}

```

19.8.5. Streaming vs. Polled Input

Many Spring Cloud Stream applications will use the built-in `Sink` binding, which triggers the *streaming* input binder creation. Messages can then be consumed with an input handler marked by `@StreamListener(Sink.INPUT)` annotation, at whatever rate Pub/Sub sends them.

For more control over the rate of message arrival, a polled input binder can be set up by defining a custom binding interface with an `@Input`-annotated method returning `PollableMessageSource`.

```

public interface PollableSink {

    @Input("input")
    PollableMessageSource input();
}

```

The `PollableMessageSource` can then be injected and queried, as needed.

```

@EnableBinding(PollableSink.class)
public class SinkExample {

    @Autowired
    PollableMessageSource destIn;

    @Bean
    public ApplicationRunner singlePollRunner() {
        return args -> {
            // This will poll only once.
            // Add a loop or a scheduler to get more messages.
            destIn.poll((message) -> System.out.println("Message retrieved: " +
message));
        };
    }
}

```

19.8.6. Sample

Sample applications are available:

- For [streaming input, annotation-based](#).
- For [streaming input, functional style](#).
- For [polled input](#).

19.9. Spring Cloud Bus

Using [Cloud Pub/Sub](#) as the [Spring Cloud Bus](#) implementation is as simple as importing the `spring-cloud-gcp-starter-bus-pubsub` starter.

This starter brings in the [Spring Cloud Stream binder for Cloud Pub/Sub](#), which is used to both publish and subscribe to the bus. If the bus topic (named `springCloudBus` by default) does not exist, the binder automatically creates it. The binder also creates anonymous subscriptions for each project using the `spring-cloud-gcp-starter-bus-pubsub` starter.

Maven coordinates, using [Spring Cloud GCP BOM](#):

```

<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-gcp-starter-bus-pubsub</artifactId>
</dependency>

```

Gradle coordinates:

```
dependencies {
    compile group: 'org.springframework.cloud', name: 'spring-cloud-gcp-starter-bus-
pubsub'
}
```

19.9.1. Configuration Management with Spring Cloud Config and Spring Cloud Bus

Spring Cloud Bus can be used to push configuration changes from a Spring Cloud Config server to the clients listening on the same bus.

To use GCP Pub/Sub as the bus implementation, both the configuration server and the configuration client need the `spring-cloud-gcp-starter-bus-pubsub` dependency.

All other configuration is standard to [Spring Cloud Config](#).

[spring cloud bus over pubsub] | [spring_cloud_bus_over_pubsub.png](#)

Spring Cloud Config Server typically runs on port 8888, and can read configuration from a [variety of source control systems](#) such as GitHub, and even from the local filesystem. When the server is notified that new configuration is available, it fetches the updated configuration and sends a notification (`RefreshRemoteApplicationEvent`) out via Spring Cloud Bus.

When configuration is stored locally, config server polls the parent directory for changes. With configuration stored in source control repository, such as GitHub, the config server needs to be notified that a new version of configuration is available. In a deployed server, this would be done automatically through a GitHub webhook, but in a local testing scenario, the `/monitor` HTTP endpoint needs to be invoked manually.

```
curl -X POST http://localhost:8888/monitor -H "X-Github-Event: push" -H "Content-Type:
application/json" -d '{"commits": [{"modified": ["application.properties"]}]}'
```

By adding the `spring-cloud-gcp-starter-bus-pubsub` dependency, you instruct Spring Cloud Bus to use Cloud Pub/Sub to broadcast configuration changes. Spring Cloud Bus will then create a topic named `springCloudBus`, as well as a subscription for each configuration client.

The configuration server happens to also be a configuration client, subscribing to the configuration changes that it sends out. Thus, in a scenario with one configuration server and one configuration client, two anonymous subscriptions to the `springCloudBus` topic are created. However, a config server disables configuration refresh by default (see [ConfigServerBootstrapApplicationListener](#) for more details).

A [demo application](#) showing configuration management and distribution over a Cloud Pub/Sub-powered bus is available. The sample contains two examples of configuration management with Spring Cloud Bus: one monitoring a local file system, and the other retrieving configuration from a GitHub repository.

19.10. Spring Cloud Sleuth

[Spring Cloud Sleuth](#) is an instrumentation framework for Spring Boot applications. It captures trace information and can forward traces to services like Zipkin for storage and analysis.

Google Cloud Platform provides its own managed distributed tracing service called [Stackdriver Trace](#). Instead of running and maintaining your own Zipkin instance and storage, you can use Stackdriver Trace to store traces, view trace details, generate latency distributions graphs, and generate performance regression reports.

This Spring Cloud GCP starter can forward Spring Cloud Sleuth traces to Stackdriver Trace without an intermediary Zipkin server.

Maven coordinates, using [Spring Cloud GCP BOM](#):

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-gcp-starter-trace</artifactId>
</dependency>
```

Gradle coordinates:

```
dependencies {
  compile group: 'org.springframework.cloud', name: 'spring-cloud-gcp-starter-trace'
}
```

You must enable Stackdriver Trace API from the Google Cloud Console in order to capture traces. Navigate to the [Stackdriver Trace API](#) for your project and make sure it's enabled.



If you are already using a Zipkin server capturing trace information from multiple platform/frameworks, you can also use a [Stackdriver Zipkin proxy](#) to forward those traces to Stackdriver Trace without modifying existing applications.

19.10.1. Tracing

Spring Cloud Sleuth uses the [Brave tracer](#) to generate traces. This integration enables Brave to use the [StackdriverTracePropagation](#) propagation.

A propagation is responsible for extracting trace context from an entity (e.g., an HTTP servlet request) and injecting trace context into an entity. A canonical example of the propagation usage is a web server that receives an HTTP request, which triggers other HTTP requests from the server before returning an HTTP response to the original caller. In the case of [StackdriverTracePropagation](#), first it looks for trace context in the `x-cloud-trace-context` key (e.g., an HTTP request header). The value of the `x-cloud-trace-context` key can be formatted in three different ways:

- `x-cloud-trace-context: TRACE_ID`

- `x-cloud-trace-context: TRACE_ID/SPAN_ID`
- `x-cloud-trace-context: TRACE_ID/SPAN_ID;o=TRACE_TRUE`

`TRACE_ID` is a 32-character hexadecimal value that encodes a 128-bit number.

`SPAN_ID` is an unsigned long. Since Stackdriver Trace doesn't support span joins, a new span ID is always generated, regardless of the one specified in `x-cloud-trace-context`.

`TRACE_TRUE` can either be `0` if the entity should be untraced, or `1` if it should be traced. This field forces the decision of whether or not to trace the request; if omitted then the decision is deferred to the sampler.

If a `x-cloud-trace-context` key isn't found, `StackdriverTracePropagation` falls back to tracing with the [X-B3 headers](#).

19.10.2. Spring Boot Starter for Stackdriver Trace

Spring Boot Starter for Stackdriver Trace uses Spring Cloud Sleuth and auto-configures a `StackdriverSender` that sends the Sleuth's trace information to Stackdriver Trace.

All configurations are optional:

Name	Description	Required	Default value
<code>spring.cloud.gcp.trace.enabled</code>	Auto-configure Spring Cloud Sleuth to send traces to Stackdriver Trace.	No	<code>true</code>
<code>spring.cloud.gcp.trace.project-id</code>	Overrides the project ID from the Spring Cloud GCP Module	No	
<code>spring.cloud.gcp.trace.credentials.location</code>	Overrides the credentials location from the Spring Cloud GCP Module	No	
<code>spring.cloud.gcp.trace.credentials.encoded-key</code>	Overrides the credentials encoded key from the Spring Cloud GCP Module	No	
<code>spring.cloud.gcp.trace.credentials.scopes</code>	Overrides the credentials scopes from the Spring Cloud GCP Module	No	
<code>spring.cloud.gcp.trace.num-executor-threads</code>	Number of threads used by the Trace executor	No	4

<code>spring.cloud.gcp.trace.authority</code>	HTTP/2 authority the channel claims to be connecting to.	No	
<code>spring.cloud.gcp.trace.compression</code>	Name of the compression to use in Trace calls	No	
<code>spring.cloud.gcp.trace.deadline-ms</code>	Call deadline in milliseconds	No	
<code>spring.cloud.gcp.trace.max-inbound-size</code>	Maximum size for inbound messages	No	
<code>spring.cloud.gcp.trace.max-outbound-size</code>	Maximum size for outbound messages	No	
<code>spring.cloud.gcp.trace.wait-for-ready</code>	Waits for the channel to be ready in case of a transient failure	No	<code>false</code>
<code>spring.cloud.gcp.trace.messageTimeout</code>	Timeout in seconds before pending spans will be sent in batches to GCP Stackdriver Trace. (previously <code>spring.zipkin.messageTimeout</code>)	No	1

You can use core Spring Cloud Sleuth properties to control Sleuth's sampling rate, etc. Read [Sleuth documentation](#) for more information on Sleuth configurations.

For example, when you are testing to see the traces are going through, you can set the sampling rate to 100%.

```
spring.sleuth.sampler.probability=1           # Send 100% of the request
traces to Stackdriver.
spring.sleuth.web.skipPattern=(^cleanup.*|.favicon.*) # Ignore some URL paths.
spring.sleuth.scheduled.enabled=false        # disable executor 'async'
traces
```



By default, Spring Cloud Sleuth auto-configuration instruments executor beans, which may cause recurring traces with the name `async` to appear in Stackdriver Trace if your application or one of its dependencies introduces scheduler beans into Spring application context. To avoid this noise, please disable automatic instrumentation of executors via `spring.sleuth.scheduled.enabled=false` in your application configuration.

Spring Cloud GCP Trace does override some Sleuth configurations:

- Always uses 128-bit Trace IDs. This is required by Stackdriver Trace.

- Does not use Span joins. Span joins will share the span ID between the client and server Spans. Stackdriver requires that every Span ID within a Trace to be unique, so Span joins are not supported.
- Uses `StackdriverHttpClientParser` and `StackdriverHttpServerParser` by default to populate Stackdriver related fields.

19.10.3. Overriding the auto-configuration

Spring Cloud Sleuth supports sending traces to multiple tracing systems as of version 2.1.0. In order to get this to work, every tracing system needs to have a `Reporter` and `Sender`. If you want to override the provided beans you need to give them a specific name. To do this you can use respectively `StackdriverTraceAutoConfiguration.REPORTER_BEAN_NAME` and `StackdriverTraceAutoConfiguration.SENDER_BEAN_NAME`.

19.10.4. Customizing spans

You can add additional tags and annotations to spans by using the `brave.SpanCustomizer`, which is available in the application context.

Here's an example that uses `WebMvcConfigurer` to configure an MVC interceptor that adds two extra tags to all web controller spans.

```
@SpringBootApplication
public class Application implements WebMvcConfigurer {

    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }

    @Autowired
    private SpanCustomizer spanCustomizer;

    @Override
    public void addInterceptors(InterceptorRegistry registry) {
        registry.addInterceptor(new HandlerInterceptor() {
            @Override
            public boolean preHandle(HttpServletRequest request, HttpServletResponse
response, Object handler) throws Exception {
                spanCustomizer.tag("session-id", request.getSession().getId());
                spanCustomizer.tag("environment", "QA");

                return true;
            }
        });
    }
}
```

You can then search and filter traces based on these additional tags in the Stackdriver Trace service.

19.10.5. Integration with Logging

Integration with Stackdriver Logging is available through the [Stackdriver Logging Support](#). If the Trace integration is used together with the Logging one, the request logs will be associated to the corresponding traces. The trace logs can be viewed by going to the [Google Cloud Console Trace List](#), selecting a trace and pressing the [Logs](#) → [View](#) link in the [Details](#) section.

19.10.6. Sample

A [sample application](#) and a [codelab](#) are available.

19.11. Stackdriver Logging

Maven coordinates, using [Spring Cloud GCP BOM](#):

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-gcp-starter-logging</artifactId>
</dependency>
```

Gradle coordinates:

```
dependencies {
    compile group: 'org.springframework.cloud', name: 'spring-cloud-gcp-starter-logging'
}
```

[Stackdriver Logging](#) is the managed logging service provided by Google Cloud Platform.

This module provides support for associating a web request trace ID with the corresponding log entries. It does so by retrieving the [X-B3-TraceId](#) value from the [Mapped Diagnostic Context \(MDC\)](#), which is set by Spring Cloud Sleuth. If Spring Cloud Sleuth isn't used, the configured [TraceIdExtractor](#) extracts the desired header value and sets it as the log entry's trace ID. This allows grouping of log messages by request, for example, in the [Google Cloud Console Logs viewer](#).



Due to the way logging is set up, the GCP project ID and credentials defined in `application.properties` are ignored. Instead, you should set the `GOOGLE_CLOUD_PROJECT` and `GOOGLE_APPLICATION_CREDENTIALS` environment variables to the project ID and credentials private key location, respectively. You can do this easily if you're using the [Google Cloud SDK](#), using the `gcloud config set project [YOUR_PROJECT_ID]` and `gcloud auth application-default login` commands, respectively.

19.11.1. Web MVC Interceptor

For use in Web MVC-based applications, [TraceIdLoggingWebMvcInterceptor](#) is provided that extracts

the request trace ID from an HTTP request using a `TraceIdExtractor` and stores it in a thread-local, which can then be used in a logging appender to add the trace ID metadata to log messages.



If Spring Cloud GCP Trace is enabled, the logging module disables itself and delegates log correlation to Spring Cloud Sleuth.

`LoggingWebMvcConfigurer` configuration class is also provided to help register the `TraceIdLoggingWebMvcInterceptor` in Spring MVC applications.

Applications hosted on the Google Cloud Platform include trace IDs under the `x-cloud-trace-context` header, which will be included in log entries. However, if Sleuth is used the trace ID will be picked up from the MDC.

19.11.2. Logback Support

Currently, only Logback is supported and there are 2 possibilities to log to Stackdriver via this library with Logback: via direct API calls and through JSON-formatted console logs.

Log via API

A Stackdriver appender is available using `org/springframework/cloud/gcp/logging/logback-appender.xml`. This appender builds a Stackdriver Logging log entry from a JUL or Logback log entry, adds a trace ID to it and sends it to Stackdriver Logging.

`STACKDRIVER_LOG_NAME` and `STACKDRIVER_LOG_FLUSH_LEVEL` environment variables can be used to customize the `STACKDRIVER` appender.

Your configuration may then look like this:

```
<configuration>
  <include resource="org/springframework/cloud/gcp/logging/logback-appender.xml" />

  <root level="INFO">
    <appender-ref ref="STACKDRIVER" />
  </root>
</configuration>
```

If you want to have more control over the log output, you can further configure the appender. The following properties are available:

Property	Default Value	Description
<code>log</code>	<code>spring.log</code>	The Stackdriver Log name. This can also be set via the <code>STACKDRIVER_LOG_NAME</code> environmental variable.

Property	Default Value	Description
<code>flushLevel</code>	<code>WARN</code>	If a log entry with this level is encountered, trigger a flush of locally buffered log to Stackdriver Logging. This can also be set via the <code>STACKDRIVER_LOG_FLUSH_LEVEL</code> environmental variable.

Log via Console

For Logback, a `org/springframework/cloud/gcp/logging/logback-json-appender.xml` file is made available for import to make it easier to configure the JSON Logback appender.

Your configuration may then look something like this:

```
<configuration>
  <include resource="org/springframework/cloud/gcp/logging/logback-json-appender.xml"
  />

  <root level="INFO">
    <appender-ref ref="CONSOLE_JSON" />
  </root>
</configuration>
```

If your application is running on Google Kubernetes Engine, Google Compute Engine or Google App Engine Flexible, your console logging is automatically saved to Google Stackdriver Logging. Therefore, you can just include `org/springframework/cloud/gcp/logging/logback-json-appender.xml` in your logging configuration, which logs JSON entries to the console. The trace id will be set correctly.

If you want to have more control over the log output, you can further configure the appender. The following properties are available:

Property	Default Value	Description
<code>projectId</code>	<p>If not set, default value is determined in the following order:</p> <ol style="list-style-type: none"> <code>SPRING_CLOUD_GCP_LOGGING_PROJECT_ID</code> Environmental Variable. Value of <code>DefaultGcpProjectIdProvider.getProjectId()</code> 	<p>This is used to generate fully qualified Stackdriver Trace ID format: <code>projects/[PROJECT-ID]/traces/[TRACE-ID]</code>.</p> <p>This format is required to correlate trace between Stackdriver Trace and Stackdriver Logging.</p> <p>If <code>projectId</code> is not set and cannot be determined, then it'll log <code>traceId</code> without the fully qualified format.</p>
<code>includeTraceId</code>	<code>true</code>	Should the <code>traceId</code> be included
<code>includeSpanId</code>	<code>true</code>	Should the <code>spanId</code> be included
<code>includeLevel</code>	<code>true</code>	Should the severity be included
<code>includeThreadName</code>	<code>true</code>	Should the thread name be included
<code>includeMDC</code>	<code>true</code>	Should all MDC properties be included. The MDC properties <code>X-B3-TraceId</code> , <code>X-B3-SpanId</code> and <code>X-Span-Export</code> provided by Spring Sleuth will get excluded as they get handled separately
<code>includeLoggerName</code>	<code>true</code>	Should the name of the logger be included
<code>includeFormattedMessage</code>	<code>true</code>	Should the formatted log message be included.
<code>includeExceptionInMessage</code>	<code>true</code>	Should the stacktrace be appended to the formatted log message. This setting is only evaluated if <code>includeFormattedMessage</code> is <code>true</code>
<code>includeContextName</code>	<code>true</code>	Should the logging context be included
<code>includeMessage</code>	<code>false</code>	Should the log message with blank placeholders be included
<code>includeException</code>	<code>false</code>	Should the stacktrace be included as a own field

Property	Default Value	Description
<code>serviceContext</code>	none	Define the Stackdriver service context data (service and version). This allows filtering of error reports for service and version in the Google Cloud Error Reporting View .
<code>customJson</code>	none	Defines custom json data. Data will be added to the json output.

This is an example of such an Logback configuration:

```
<configuration >
  <property name="projectId" value="\${projectId:-\${GOOGLE_CLOUD_PROJECT}}"/>

  <appender name="CONSOLE_JSON" class="ch.qos.logback.core.ConsoleAppender">
    <encoder class="ch.qos.logback.core.encoder.LayoutWrappingEncoder">
      <layout class="org.springframework.cloud.gcp.logging.StackdriverJsonLayout">
        <projectId>\${projectId}</projectId>

        <!--<includeTraceId>true</includeTraceId>-->
        <!--<includeSpanId>true</includeSpanId>-->
        <!--<includeLevel>true</includeLevel>-->
        <!--<includeThreadName>true</includeThreadName>-->
        <!--<includeMDC>true</includeMDC>-->
        <!--<includeLoggerName>true</includeLoggerName>-->
        <!--<includeFormattedMessage>true</includeFormattedMessage>-->
        <!--<includeExceptionInMessage>true</includeExceptionInMessage>-->
        <!--<includeContextName>true</includeContextName>-->
        <!--<includeMessage>>false</includeMessage>-->
        <!--<includeException>>false</includeException>-->
        <!--<serviceContext>
          <service>service-name</service>
          <version>service-version</version>
        </serviceContext>-->
        <!--<customJson>{"custom-key": "custom-value"}</customJson>-->
      </layout>
    </encoder>
  </appender>
</configuration>
```

19.11.3. Sample

A [Sample Spring Boot Application](#) is provided to show how to use the Cloud logging starter.

19.12. Spring Cloud Config

Spring Cloud GCP makes it possible to use the [Google Runtime Configuration API](#) as a [Spring Cloud Config](#) server to remotely store your application configuration data.

The Spring Cloud GCP Config support is provided via its own Spring Boot starter. It enables the use of the Google Runtime Configuration API as a source for Spring Boot configuration properties.



The Google Cloud Runtime Configuration service is in **Beta** status, and is only available in snapshot and milestone versions of the project. It's also not available in the Spring Cloud GCP BOM, unlike other modules.

Maven coordinates:

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-gcp-starter-config</artifactId>
  <version>{spring-cloud-gcp-version}</version>
</dependency>
```

Gradle coordinates:

```
dependencies {
    compile group: 'org.springframework.cloud',
            name: 'spring-cloud-gcp-starter-config',
            version: '{spring-cloud-gcp-version}'
}
```

19.12.1. Configuration

The following parameters are configurable in Spring Cloud GCP Config:

Name	Description	Required	Default value
<code>spring.cloud.gcp.config.enabled</code>	Enables the Config client	No	<code>false</code>
<code>spring.cloud.gcp.config.name</code>	Name of your application	No	Value of the <code>spring.application.name</code> property. If none, <code>application</code>
<code>spring.cloud.gcp.config.profile</code>	Active profile	No	Value of the <code>spring.profiles.active</code> property. If more than a single profile, last one is chosen

<code>spring.cloud.gcp.config.timeout-millis</code>	Timeout in milliseconds for connecting to the Google Runtime Configuration API	No	60000
<code>spring.cloud.gcp.config.project-id</code>	GCP project ID where the Google Runtime Configuration API is hosted	No	
<code>spring.cloud.gcp.config.credentials.location</code>	OAuth2 credentials for authenticating with the Google Runtime Configuration API	No	
<code>spring.cloud.gcp.config.credentials.encoded-key</code>	Base64-encoded OAuth2 credentials for authenticating with the Google Runtime Configuration API	No	
<code>spring.cloud.gcp.config.credentials.scopes</code>	OAuth2 scope for Spring Cloud GCP Config credentials	No	www.googleapis.com/auth/cloudruntimeconfig



These properties should be specified in a `bootstrap.yml/bootstrap.properties` file, rather than the usual `applications.yml/application.properties`.



Core properties, as described in [Spring Cloud GCP Core Module](#), do not apply to Spring Cloud GCP Config.

19.12.2. Quick start

1. Create a configuration in the Google Runtime Configuration API that is called `${spring.application.name}_${spring.profiles.active}`. In other words, if `spring.application.name` is `myapp` and `spring.profiles.active` is `prod`, the configuration should be called `myapp_prod`.

In order to do that, you should have the [Google Cloud SDK](#) installed, own a Google Cloud Project and run the following command:

```
gcloud init # if this is your first Google Cloud SDK run.
gcloud beta runtime-config configs create myapp_prod
gcloud beta runtime-config configs variables set myapp.queue-size 25 --config-name myapp_prod
```

2. Configure your `bootstrap.properties` file with your application's configuration data:

```
spring.application.name=myapp
spring.profiles.active=prod
```

3. Add the `@ConfigurationProperties` annotation to a Spring-managed bean:

```
@Component
@ConfigurationProperties("myapp")
public class SampleConfig {

    private int queueSize;

    public int getQueueSize() {
        return this.queueSize;
    }

    public void setQueueSize(int queueSize) {
        this.queueSize = queueSize;
    }
}
```

When your Spring application starts, the `queueSize` field value will be set to 25 for the above `SampleConfig` bean.

19.12.3. Refreshing the configuration at runtime

[Spring Cloud](#) provides support to have configuration parameters be reloadable with the POST request to `/actuator/refresh` endpoint.

1. Add the Spring Boot Actuator dependency:

Maven coordinates:

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
```

Gradle coordinates:

```
dependencies {
    compile group: 'org.springframework.boot', name: 'spring-boot-starter-actuator'
}
```

2. Add `@RefreshScope` to your Spring configuration class to have parameters be reloadable at runtime.

3. Add `management.endpoints.web.exposure.include=refresh` to your `application.properties` to allow unrestricted access to `/actuator/refresh`.
4. Update a property with `gcloud`:

```
$ gcloud beta runtime-config configs variables set \  
  myapp.queue_size 200 \  
  --config-name myapp_prod
```

5. Send a POST request to the refresh endpoint:

```
$ curl -XPOST https://myapp.host.com/actuator/refresh
```

19.12.4. Sample

A [sample application](#) and a [codelab](#) are available.

19.13. Spring Data Cloud Spanner

[Spring Data](#) is an abstraction for storing and retrieving POJOs in numerous storage technologies. Spring Cloud GCP adds Spring Data support for [Google Cloud Spanner](#).

Maven coordinates for this module only, using [Spring Cloud GCP BOM](#):

```
<dependency>  
  <groupId>org.springframework.cloud</groupId>  
  <artifactId>spring-cloud-gcp-data-spanner</artifactId>  
</dependency>
```

Gradle coordinates:

```
dependencies {  
  compile group: 'org.springframework.cloud', name: 'spring-cloud-gcp-data-spanner'  
}
```

We provide a [Spring Boot Starter for Spring Data Spanner](#), with which you can leverage our recommended auto-configuration setup. To use the starter, see the coordinates see below.

Maven:

```
<dependency>  
  <groupId>org.springframework.cloud</groupId>  
  <artifactId>spring-cloud-gcp-starter-data-spanner</artifactId>  
</dependency>
```

Gradle:

```
dependencies {  
    compile group: 'org.springframework.cloud', name: 'spring-cloud-gcp-starter-data-spanner'  
}
```

This setup takes care of bringing in the latest compatible version of Cloud Java Cloud Spanner libraries as well.

19.13.1. Configuration

To setup Spring Data Cloud Spanner, you have to configure the following:

- Setup the connection details to Google Cloud Spanner.
- Enable Spring Data Repositories (optional).

Cloud Spanner settings

You can use the [Spring Boot Starter for Spring Data Spanner](#) to autoconfigure Google Cloud Spanner in your Spring application. It contains all the necessary setup that makes it easy to authenticate with your Google Cloud project. The following configuration options are available:

Name	Description	Required	Default value
<code>spring.cloud.gcp.spanner.instance-id</code>	Cloud Spanner instance to use	Yes	
<code>spring.cloud.gcp.spanner.database</code>	Cloud Spanner database to use	Yes	
<code>spring.cloud.gcp.spanner.project-id</code>	GCP project ID where the Google Cloud Spanner API is hosted, if different from the one in the Spring Cloud GCP Core Module	No	
<code>spring.cloud.gcp.spanner.credentials.location</code>	OAuth2 credentials for authenticating with the Google Cloud Spanner API, if different from the ones in the Spring Cloud GCP Core Module	No	

<code>spring.cloud.gcp.spanner.credentials.encoded-key</code>	Base64-encoded OAuth2 credentials for authenticating with the Google Cloud Spanner API, if different from the ones in the Spring Cloud GCP Core Module	No	
<code>spring.cloud.gcp.spanner.credentials.scopes</code>	OAuth2 scope for Spring Cloud GCP Cloud Spanner credentials	No	www.googleapis.com/auth/spanner.data
<code>spring.cloud.gcp.spanner.createInterleavedTableDdlOnDeleteCascade</code>	If <code>true</code> , then schema statements generated by <code>SpannerSchemaUtils</code> for tables with interleaved parent-child relationships will be "ON DELETE CASCADE". The schema for the tables will be "ON DELETE NO ACTION" if <code>false</code> .	No	<code>true</code>
<code>spring.cloud.gcp.spanner.numRpcChannels</code>	Number of gRPC channels used to connect to Cloud Spanner	No	4 - Determined by Cloud Spanner client library
<code>spring.cloud.gcp.spanner.prefetchChunks</code>	Number of chunks prefetched by Cloud Spanner for read and query	No	4 - Determined by Cloud Spanner client library
<code>spring.cloud.gcp.spanner.minSessions</code>	Minimum number of sessions maintained in the session pool	No	0 - Determined by Cloud Spanner client library
<code>spring.cloud.gcp.spanner.maxSessions</code>	Maximum number of sessions session pool can have	No	400 - Determined by Cloud Spanner client library
<code>spring.cloud.gcp.spanner.maxIdleSessions</code>	Maximum number of idle sessions session pool will maintain	No	0 - Determined by Cloud Spanner client library
<code>spring.cloud.gcp.spanner.writeSessionsFraction</code>	Fraction of sessions to be kept prepared for write transactions	No	0.2 - Determined by Cloud Spanner client library
<code>spring.cloud.gcp.spanner.keepAliveIntervalMinutes</code>	How long to keep idle sessions alive	No	30 - Determined by Cloud Spanner client library

<code>spring.cloud.gcp.spanner.failIfPoolExhausted</code>	If all sessions are in use, fail the request by throwing an exception. Otherwise, by default, block until a session becomes available.	No	false
---	--	----	-------

Repository settings

Spring Data Repositories can be configured via the `@EnableSpannerRepositories` annotation on your main `@Configuration` class. With our Spring Boot Starter for Spring Data Cloud Spanner, `@EnableSpannerRepositories` is automatically added. It is not required to add it to any other class, unless there is a need to override finer grain configuration parameters provided by `@EnableSpannerRepositories`.

Autoconfiguration

Our Spring Boot autoconfiguration creates the following beans available in the Spring application context:

- an instance of `SpannerTemplate`
- an instance of `SpannerDatabaseAdminTemplate` for generating table schemas from object hierarchies and creating and deleting tables and databases
- an instance of all user-defined repositories extending `SpannerRepository`, `CrudRepository`, `PagingAndSortingRepository`, when repositories are enabled
- an instance of `DatabaseClient` from the Google Cloud Java Client for Spanner, for convenience and lower level API access

19.13.2. Object Mapping

Spring Data Cloud Spanner allows you to map domain POJOs to Cloud Spanner tables via annotations:

```
@Table(name = "traders")
public class Trader {

    @PrimaryKey
    @Column(name = "trader_id")
    String traderId;

    String firstName;

    String lastName;

    @NotMapped
    Double temporaryNumber;
}
```

Spring Data Cloud Spanner will ignore any property annotated with `@NotMapped`. These properties will not be written to or read from Spanner.

Constructors

Simple constructors are supported on POJOs. The constructor arguments can be a subset of the persistent properties. Every constructor argument needs to have the same name and type as a persistent property on the entity and the constructor should set the property from the given argument. Arguments that are not directly set to properties are not supported.

```
@Table(name = "traders")
public class Trader {
    @PrimaryKey
    @Column(name = "trader_id")
    String traderId;

    String firstName;

    String lastName;

    @NotMapped
    Double temporaryNumber;

    public Trader(String traderId, String firstName) {
        this.traderId = traderId;
        this.firstName = firstName;
    }
}
```

Table

The `@Table` annotation can provide the name of the Cloud Spanner table that stores instances of the annotated class, one per row. This annotation is optional, and if not given, the name of the table is inferred from the class name with the first character uncapitalized.

SpEL expressions for table names

In some cases, you might want the `@Table` table name to be determined dynamically. To do that, you can use [Spring Expression Language](#).

For example:

```
@Table(name = "trades_#{tableNameSuffix}")
public class Trade {
    // ...
}
```

The table name will be resolved only if the `tableNameSuffix` value/bean in the Spring application

context is defined. For example, if `tableNameSuffix` has the value "123", the table name will resolve to `trades_123`.

Primary Keys

For a simple table, you may only have a primary key consisting of a single column. Even in that case, the `@PrimaryKey` annotation is required. `@PrimaryKey` identifies the one or more ID properties corresponding to the primary key.

Spanner has first class support for composite primary keys of multiple columns. You have to annotate all of your POJO's fields that the primary key consists of with `@PrimaryKey` as below:

```
@Table(name = "trades")
public class Trade {
    @PrimaryKey(keyOrder = 2)
    @Column(name = "trade_id")
    private String tradeId;

    @PrimaryKey(keyOrder = 1)
    @Column(name = "trader_id")
    private String traderId;

    private String action;

    private Double price;

    private Double shares;

    private String symbol;
}
```

The `keyOrder` parameter of `@PrimaryKey` identifies the properties corresponding to the primary key columns in order, starting with 1 and increasing consecutively. Order is important and must reflect the order defined in the Cloud Spanner schema. In our example the DDL to create the table and its primary key is as follows:

```
CREATE TABLE trades (
    trader_id STRING(MAX),
    trade_id STRING(MAX),
    action STRING(15),
    symbol STRING(10),
    price FLOAT64,
    shares FLOAT64
) PRIMARY KEY (trader_id, trade_id)
```

Spanner does not have automatic ID generation. For most use-cases, sequential IDs should be used with caution to avoid creating data hotspots in the system. Read [Spanner Primary Keys documentation](#) for a better understanding of primary keys and recommended practices.

Columns

All accessible properties on POJOs are automatically recognized as a Cloud Spanner column. Column naming is generated by the `PropertyNameFieldNamingStrategy` by default defined on the `SpannerMappingContext` bean. The `@Column` annotation optionally provides a different column name than that of the property and some other settings:

- `name` is the optional name of the column
- `spannerTypeMaxLength` specifies for `STRING` and `BYTES` columns the maximum length. This setting is only used when generating DDL schema statements based on domain types.
- `nullable` specifies if the column is created as `NOT NULL`. This setting is only used when generating DDL schema statements based on domain types.
- `spannerType` is the Cloud Spanner column type you can optionally specify. If this is not specified then a compatible column type is inferred from the Java property type.
- `spannerCommitTimestamp` is a boolean specifying if this property corresponds to an auto-populated commit timestamp column. Any value set in this property will be ignored when writing to Cloud Spanner.

Embedded Objects

If an object of type `B` is embedded as a property of `A`, then the columns of `B` will be saved in the same Cloud Spanner table as those of `A`.

If `B` has primary key columns, those columns will be included in the primary key of `A`. `B` can also have embedded properties. Embedding allows reuse of columns between multiple entities, and can be useful for implementing parent-child situations, because Cloud Spanner requires child tables to include the key columns of their parents.

For example:

```

class X {
    @PrimaryKey
    String grandParentId;

    long age;
}

class A {
    @PrimaryKey
    @Embedded
    X grandParent;

    @PrimaryKey(keyOrder = 2)
    String parentId;

    String value;
}

@Table(name = "items")
class B {
    @PrimaryKey
    @Embedded
    A parent;

    @PrimaryKey(keyOrder = 2)
    String id;

    @Column(name = "child_value")
    String value;
}

```

Entities of **B** can be stored in a table defined as:

```

CREATE TABLE items (
    grandParentId STRING(MAX),
    parentId STRING(MAX),
    id STRING(MAX),
    value STRING(MAX),
    child_value STRING(MAX),
    age INT64
) PRIMARY KEY (grandParentId, parentId, id)

```

Note that embedded properties' column names must all be unique.

Relationships

Spring Data Cloud Spanner supports parent-child relationships using the Cloud Spanner [parent-child interleaved table mechanism](#). Cloud Spanner interleaved tables enforce the one-to-many

relationship and provide efficient queries and operations on entities of a single domain parent entity. These relationships can be up to 7 levels deep. Cloud Spanner also provides automatic cascading delete or enforces the deletion of child entities before parents.

While one-to-one and many-to-many relationships can be implemented in Cloud Spanner and Spring Data Cloud Spanner using constructs of interleaved parent-child tables, only the parent-child relationship is natively supported. Cloud Spanner does not support the foreign key constraint, though the parent-child key constraint enforces a similar requirement when used with interleaved tables.

For example, the following Java entities:

```
@Table(name = "Singers")
class Singer {
    @PrimaryKey
    long SingerId;

    String FirstName;

    String LastName;

    byte[] SingerInfo;

    @Interleaved
    List<Album> albums;
}

@Table(name = "Albums")
class Album {
    @PrimaryKey
    long SingerId;

    @PrimaryKey(keyOrder = 2)
    long AlbumId;

    String AlbumTitle;
}
```

These classes can correspond to an existing pair of interleaved tables. The `@Interleaved` annotation may be applied to `Collection` properties and the inner type is resolved as the child entity type. The schema needed to create them can also be generated using the `SpannerSchemaUtils` and executed using the `SpannerDatabaseAdminTemplate`:

```

@Autowired
SpannerSchemaUtils schemaUtils;

@Autowired
SpannerDatabaseAdminTemplate databaseAdmin;
...

// Get the create statmenets for all tables in the table structure rooted at Singer
List<String> createStrings =
this.schemaUtils.getCreateTableDdlStringsForInterleavedHierarchy(Singer.class);

// Create the tables and also create the database if necessary
this.databaseAdmin.executeDdlStrings(createStrings, true);

```

The `createStrings` list contains table schema statements using column names and types compatible with the provided Java type and any resolved child relationship types contained within based on the configured custom converters.

```

CREATE TABLE Singers (
  SingerId  INT64 NOT NULL,
  FirstName STRING(1024),
  LastName  STRING(1024),
  SingerInfo BYTES(MAX),
) PRIMARY KEY (SingerId);

CREATE TABLE Albums (
  SingerId  INT64 NOT NULL,
  AlbumId   INT64 NOT NULL,
  AlbumTitle STRING(MAX),
) PRIMARY KEY (SingerId, AlbumId),
INTERLEAVE IN PARENT Singers ON DELETE CASCADE;

```

The `ON DELETE CASCADE` clause indicates that Cloud Spanner will delete all Albums of a singer if the Singer is deleted. The alternative is `ON DELETE NO ACTION`, where a Singer cannot be deleted until all of its Albums have already been deleted. When using `SpannerSchemaUtils` to generate the schema strings, the `spring.cloud.gcp.spanner.createInterleavedTableDdlOnDeleteCascade` boolean setting determines if these schema are generated as `ON DELETE CASCADE` for `true` and `ON DELETE NO ACTION` for `false`.

Cloud Spanner restricts these relationships to 7 child layers. A table may have multiple child tables.

On updating or inserting an object to Cloud Spanner, all of its referenced children objects are also updated or inserted in the same request, respectively. On read, all of the interleaved child rows are also all read.

Lazy Fetch

`@Interleaved` properties are retrieved eagerly by default, but can be fetched lazily for performance

in both read and write:

```
@Interleaved(lazy = true)
List<Album> albums;
```

Lazily-fetched interleaved properties are retrieved upon the first interaction with the property. If a property marked for lazy fetching is never retrieved, then it is also skipped when saving the parent entity.

If used inside a transaction, subsequent operations on lazily-fetched properties use the same transaction context as that of the original parent entity.

Declarative Filtering with @Where

The `@Where` annotation could be applied to an entity class or to an interleaved property. This annotation provides an SQL where clause that will be applied at the fetching of interleaved collections or the entity itself.

Let's say we have an `Agreement` with a list of `Participants` which could be assigned to it. We would like to fetch a list of currently active participants. For security reasons, all records should remain in the database forever, even if participants become inactive. That can be easily achieved with the `@Where` annotation, which is demonstrated by this example:

```
@Table(name = "participants")
public class Participant {
    //...
    boolean active;
    //...
}

@Table(name = "agreements")
public class Agreement {
    //...
    @Interleaved
    @Where("active = true")
    List<Participant> participants;
    Person person;
    //...
}
```

Supported Types

Spring Data Cloud Spanner natively supports the following types for regular fields but also utilizes custom converters (detailed in following sections) and dozens of pre-defined Spring Data custom converters to handle other common Java types.

Natively supported types:

- `com.google.cloud.ByteArray`

- `com.google.cloud.Date`
- `com.google.cloud.Timestamp`
- `java.lang.Boolean`, `boolean`
- `java.lang.Double`, `double`
- `java.lang.Long`, `long`
- `java.lang.Integer`, `int`
- `java.lang.String`
- `double[]`
- `long[]`
- `boolean[]`
- `java.util.Date`
- `java.util.Instant`
- `java.sql.Date`
- `java.time.LocalDate`
- `java.time.LocalDateTime`

Lists

Spanner supports `ARRAY` types for columns. `ARRAY` columns are mapped to `List` fields in POJOS.

Example:

```
List<Double> curve;
```

The types inside the lists can be any singular property type.

Lists of Structs

Cloud Spanner queries can [construct STRUCT values](#) that appear as columns in the result. Cloud Spanner requires STRUCT values appear in ARRAYS at the root level: `SELECT ARRAY(SELECT STRUCT(1 as val1, 2 as val2)) as pair FROM Users`.

Spring Data Cloud Spanner will attempt to read the column STRUCT values into a property that is an `Iterable` of an entity type compatible with the schema of the column STRUCT value.

For the previous array-select example, the following property can be mapped with the constructed `ARRAY<STRUCT>` column: `List<TwoInts> pair;` where the `TwoInts` type is defined:

```
class TwoInts {  
    int val1;  
  
    int val2;  
}
```

Custom types

Custom converters can be used to extend the type support for user defined types.

1. Converters need to implement the `org.springframework.core.convert.converter.Converter` interface in both directions.
2. The user defined type needs to be mapped to one of the basic types supported by Spanner:
 - `com.google.cloud.ByteArray`
 - `com.google.cloud.Date`
 - `com.google.cloud.Timestamp`
 - `java.lang.Boolean`, `boolean`
 - `java.lang.Double`, `double`
 - `java.lang.Long`, `long`
 - `java.lang.String`
 - `double[]`
 - `long[]`
 - `boolean[]`
 - `enum` types
3. An instance of both Converters needs to be passed to a `ConverterAwareMappingSpannerEntityProcessor`, which then has to be made available as a `@Bean` for `SpannerEntityProcessor`.

For example:

We would like to have a field of type `Person` on our `Trade` POJO:

```
@Table(name = "trades")
public class Trade {
    //...
    Person person;
    //...
}
```

Where `Person` is a simple class:

```
public class Person {

    public String firstName;
    public String lastName;

}
```

We have to define the two converters:

```

public class PersonWriteConverter implements Converter<Person, String> {

    @Override
    public String convert(Person person) {
        return person.firstName + " " + person.lastName;
    }
}

public class PersonReadConverter implements Converter<String, Person> {

    @Override
    public Person convert(String s) {
        Person person = new Person();
        person.firstName = s.split(" ")[0];
        person.lastName = s.split(" ")[1];
        return person;
    }
}

```

That will be configured in our `@Configuration` file:

```

@Configuration
public class ConverterConfiguration {

    @Bean
    public SpannerEntityProcessor spannerEntityProcessor(SpannerMappingContext
spannerMappingContext) {
        return new ConverterAwareMappingSpannerEntityProcessor(spannerMappingContext,
            Arrays.asList(new PersonWriteConverter()),
            Arrays.asList(new PersonReadConverter()));
    }
}

```

Custom Converter for Struct Array Columns

If a `Converter<Struct, A>` is provided, then properties of type `List<A>` can be used in your entity types.

19.13.3. Spanner Operations & Template

`SpannerOperations` and its implementation, `SpannerTemplate`, provides the Template pattern familiar to Spring developers. It provides:

- Resource management
- One-stop-shop to Spanner operations with the Spring Data POJO mapping and conversion features
- Exception conversion

Using the `autoconfigure` provided by our Spring Boot Starter for Spanner, your Spring application context will contain a fully configured `SpannerTemplate` object that you can easily autowire in your application:

```
@SpringBootApplication
public class SpannerTemplateExample {

    @Autowired
    SpannerTemplate spannerTemplate;

    public void doSomething() {
        this.spannerTemplate.delete(Trade.class, KeySet.all());
        //...
        Trade t = new Trade();
        //...
        this.spannerTemplate.insert(t);
        //...
        List<Trade> tradesByAction = spannerTemplate.findAll(Trade.class);
        //...
    }
}
```

The Template API provides convenience methods for:

- [Reads](#), and by providing `SpannerReadOptions` and `SpannerQueryOptions`
 - Stale read
 - Read with secondary indices
 - Read with limits and offsets
 - Read with sorting
- [Queries](#)
- DML operations (delete, insert, update, upsert)
- Partial reads
 - You can define a set of columns to be read into your entity
- Partial writes
 - Persist only a few properties from your entity
- Read-only transactions
- Locking read-write transactions

SQL Query

Cloud Spanner has SQL support for running read-only queries. All the query related methods start with `query` on `SpannerTemplate`. Using `SpannerTemplate` you can execute SQL queries that map to POJOs:

```
List<Trade> trades = this.spannerTemplate.query(Trade.class, Statement.of("SELECT *  
FROM trades"));
```

Read

Spanner exposes a [Read API](#) for reading single row or multiple rows in a table or in a secondary index.

Using `SpannerTemplate` you can execute reads, for example:

```
List<Trade> trades = this.spannerTemplate.readAll(Trade.class);
```

Main benefit of reads over queries is reading multiple rows of a certain pattern of keys is much easier using the features of the `KeySet` class.

Advanced reads

Stale read

All reads and queries are **strong reads** by default. A **strong read** is a read at a current time and is guaranteed to see all data that has been committed up until the start of this read. An **exact staleness read** is read at a timestamp in the past. Cloud Spanner allows you to determine how current the data should be when you read data. With `SpannerTemplate` you can specify the `Timestamp` by setting it on `SpannerQueryOptions` or `SpannerReadOptions` to the appropriate read or query methods:

Reads:

```
// a read with options:  
SpannerReadOptions spannerReadOptions = new  
SpannerReadOptions().setTimestamp(myTimestamp);  
List<Trade> trades = this.spannerTemplate.readAll(Trade.class, spannerReadOptions);
```

Queries:

```
// a query with options:  
SpannerQueryOptions spannerQueryOptions = new  
SpannerQueryOptions().setTimestamp(myTimestamp);  
List<Trade> trades = this.spannerTemplate.query(Trade.class, Statement.of("SELECT *  
FROM trades"), spannerQueryOptions);
```

You can also read with **bounded staleness** by setting `.setTimestampBound(TimestampBound.ofMinReadTimestamp(myTimestamp))` on the query and read options objects. Bounded staleness lets Cloud Spanner choose any point in time later than or equal to the given `timestampBound`, but it cannot be used inside transactions.

Read from a secondary index

Using a [secondary index](#) is available for Reads via the Template API and it is also implicitly available via SQL for Queries.

The following shows how to read rows from a table using a [secondary index](#) simply by setting `index` on `SpannerReadOptions`:

```
SpannerReadOptions spannerReadOptions = new
SpannerReadOptions().setIndex("TradesByTrader");
List<Trade> trades = this.spannerTemplate.readAll(Trade.class, spannerReadOptions);
```

Read with offsets and limits

Limits and offsets are only supported by Queries. The following will get only the first two rows of the query:

```
SpannerQueryOptions spannerQueryOptions = new
SpannerQueryOptions().setLimit(2).setOffset(3);
List<Trade> trades = this.spannerTemplate.query(Trade.class, Statement.of("SELECT *
FROM trades"), spannerQueryOptions);
```

Note that the above is equivalent of executing `SELECT * FROM trades LIMIT 2 OFFSET 3`.

Sorting

Reads by keys do not support sorting. However, queries on the Template API support sorting through standard SQL and also via Spring Data Sort API:

```
List<Trade> trades = this.spannerTemplate.queryAll(Trade.class, Sort.by("action"));
```

If the provided sorted field name is that of a property of the domain type, then the column name corresponding to that property will be used in the query. Otherwise, the given field name is assumed to be the name of the column in the Cloud Spanner table. Sorting on columns of Cloud Spanner types `STRING` and `BYTES` can be done while ignoring case:

```
Sort.by(Order.desc("action").ignoreCase())
```

Partial read

Partial read is only possible when using Queries. In case the rows returned by the query have fewer columns than the entity that it will be mapped to, Spring Data will map the returned columns only. This setting also applies to nested structs and their corresponding nested POJO properties.

```
List<Trade> trades = this.spannerTemplate.query(Trade.class, Statement.of("SELECT
action, symbol FROM trades"),
    new SpannerQueryOptions().setAllowMissingResultSetColumns(true));
```

If the setting is set to `false`, then an exception will be thrown if there are missing columns in the query result.

Summary of options for Query vs Read

Feature	Query supports it	Read supports it
SQL	yes	no
Partial read	yes	no
Limits	yes	no
Offsets	yes	no
Secondary index	yes	yes
Read using index range	no	yes
Sorting	yes	no

Write / Update

The write methods of `SpannerOperations` accept a POJO and writes all of its properties to Spanner. The corresponding Spanner table and entity metadata is obtained from the given object's actual type.

If a POJO was retrieved from Spanner and its primary key properties values were changed and then written or updated, the operation will occur as if against a row with the new primary key values. The row with the original primary key values will not be affected.

Insert

The `insert` method of `SpannerOperations` accepts a POJO and writes all of its properties to Spanner, which means the operation will fail if a row with the POJO's primary key already exists in the table.

```
Trade t = new Trade();
this.spannerTemplate.insert(t);
```

Update

The `update` method of `SpannerOperations` accepts a POJO and writes all of its properties to Spanner, which means the operation will fail if the POJO's primary key does not already exist in the table.

```
// t was retrieved from a previous operation
this.spannerTemplate.update(t);
```

Upsert

The `upsert` method of `SpannerOperations` accepts a POJO and writes all of its properties to Spanner using update-or-insert.

```
// t was retrieved from a previous operation or it's new
this.spannerTemplate.upsert(t);
```

Partial Update

The update methods of `SpannerOperations` operate by default on all properties within the given object, but also accept `String[]` and `Optional<Set<String>>` of column names. If the `Optional` of set of column names is empty, then all columns are written to Spanner. However, if the `Optional` is occupied by an empty set, then no columns will be written.

```
// t was retrieved from a previous operation or it's new
this.spannerTemplate.update(t, "symbol", "action");
```

DML

DML statements can be executed using `SpannerOperations.executeDmlStatement`. Inserts, updates, and deletions can affect any number of rows and entities.

You can execute `partitioned DML` updates by using the `executePartitionedDmlStatement` method. Partitioned DML queries have performance benefits but also have restrictions and cannot be used inside transactions.

Transactions

`SpannerOperations` provides methods to run `java.util.Function` objects within a single transaction while making available the read and write methods from `SpannerOperations`.

Read/Write Transaction

Read and write transactions are provided by `SpannerOperations` via the `performReadWriteTransaction` method:

```

@Autowired
SpannerOperations mySpannerOperations;

public String doWorkInsideTransaction() {
    return mySpannerOperations.performReadWriteTransaction(
        transActionSpannerOperations -> {
            // Work with transActionSpannerOperations here.
            // It is also a SpannerOperations object.

            return "transaction completed";
        }
    );
}

```

The `performReadWriteTransaction` method accepts a `Function` that is provided an instance of a `SpannerOperations` object. The final returned value and type of the function is determined by the user. You can use this object just as you would a regular `SpannerOperations` with a few exceptions:

- Its read functionality cannot perform stale reads, because all reads and writes happen at the single point in time of the transaction.
- It cannot perform sub-transactions via `performReadWriteTransaction` or `performReadOnlyTransaction`.

As these read-write transactions are locking, it is recommended that you use the `performReadOnlyTransaction` if your function does not perform any writes.

Read-only Transaction

The `performReadOnlyTransaction` method is used to perform read-only transactions using a `SpannerOperations`:

```

@Autowired
SpannerOperations mySpannerOperations;

public String doWorkInsideTransaction() {
    return mySpannerOperations.performReadOnlyTransaction(
        transActionSpannerOperations -> {
            // Work with transActionSpannerOperations here.
            // It is also a SpannerOperations object.

            return "transaction completed";
        }
    );
}

```

The `performReadOnlyTransaction` method accepts a `Function` that is provided an instance of a `SpannerOperations` object. This method also accepts a `ReadOptions` object, but the only attribute used is the timestamp used to determine the snapshot in time to perform the reads in the transaction. If

the timestamp is not set in the read options the transaction is run against the current state of the database. The final returned value and type of the function is determined by the user. You can use this object just as you would a regular `SpannerOperations` with a few exceptions:

- Its read functionality cannot perform stale reads (other than the staleness set on the entire transaction), because all reads happen at the single point in time of the transaction.
- It cannot perform sub-transactions via `performReadWriteTransaction` or `performReadOnlyTransaction`
- It cannot perform any write operations.

Because read-only transactions are non-locking and can be performed on points in time in the past, these are recommended for functions that do not perform write operations.

Declarative Transactions with `@Transactional` Annotation

This feature requires a bean of `SpannerTransactionManager`, which is provided when using `spring-cloud-gcp-starter-data-spanner`.

`SpannerTemplate` and `SpannerRepository` support running methods with the `@Transactional` annotation as transactions. If a method annotated with `@Transactional` calls another method also annotated, then both methods will work within the same transaction. `performReadOnlyTransaction` and `performReadWriteTransaction` cannot be used in `@Transactional` annotated methods because Cloud Spanner does not support transactions within transactions.

DML Statements

`SpannerTemplate` supports `DML Statements`. DML statements can also be executed in transactions via `performReadWriteTransaction` or using the `@Transactional` annotation.

19.13.4. Repositories

`Spring Data Repositories` are a powerful abstraction that can save you a lot of boilerplate code.

For example:

```
public interface TraderRepository extends SpannerRepository<Trader, String> {  
}
```

Spring Data generates a working implementation of the specified interface, which can be conveniently autowired into an application.

The `Trader` type parameter to `SpannerRepository` refers to the underlying domain type. The second type parameter, `String` in this case, refers to the type of the key of the domain type.

For POJOs with a composite primary key, this ID type parameter can be any descendant of `Object[]` compatible with all primary key properties, any descendant of `Iterable`, or `com.google.cloud.spanner.Key`. If the domain POJO type only has a single primary key column, then the primary key property type can be used or the `Key` type.

For example in case of Trades, that belong to a Trader, `TradeRepository` would look like this:

```
public interface TradeRepository extends SpannerRepository<Trade, String[]> {  
  
}
```

```
public class MyApplication {  
  
    @Autowired  
    SpannerTemplate spannerTemplate;  
  
    @Autowired  
    StudentRepository studentRepository;  
  
    public void demo() {  
  
        this.tradeRepository.deleteAll();  
        String traderId = "demo_trader";  
        Trade t = new Trade();  
        t.symbol = stock;  
        t.action = action;  
        t.traderId = traderId;  
        t.price = 100.0;  
        t.shares = 12345.6;  
        this.spannerTemplate.insert(t);  
  
        Iterable<Trade> allTrades = this.tradeRepository.findAll();  
  
        int count = this.tradeRepository.countByAction("BUY");  
  
    }  
}
```

CRUD Repository

`CrudRepository` methods work as expected, with one thing Spanner specific: the `save` and `saveAll` methods work as update-or-insert.

Paging and Sorting Repository

You can also use `PagingAndSortingRepository` with Spanner Spring Data. The sorting and pageable `findAll` methods available from this interface operate on the current state of the Spanner database. As a result, beware that the state of the database (and the results) might change when moving page to page.

Spanner Repository

The `SpannerRepository` extends the `PagingAndSortingRepository`, but adds the read-only and the read-

write transaction functionality provided by `Spanner`. These transactions work very similarly to those of `SpannerOperations`, but is specific to the repository's domain type and provides repository functions instead of template functions.

For example, this is a read-only transaction:

```
@Autowired
SpannerRepository myRepo;

public String doWorkInsideTransaction() {
    return myRepo.performReadOnlyTransaction(
        transactionSpannerRepo -> {
            // Work with the single-transaction transactionSpannerRepo here.
            // This is a SpannerRepository object.

            return "transaction completed";
        }
    );
}
```

When creating custom repositories for your own domain types and query methods, you can extend `SpannerRepository` to access Cloud Spanner-specific features as well as all features from `PagingAndSortingRepository` and `CrudRepository`.

19.13.5. Query Methods

`SpannerRepository` supports Query Methods. Described in the following sections, these are methods residing in your custom repository interfaces of which implementations are generated based on their names and annotations. Query Methods can read, write, and delete entities in Cloud Spanner. Parameters to these methods can be any Cloud Spanner data type supported directly or via custom configured converters. Parameters can also be of type `Struct` or POJOs. If a POJO is given as a parameter, it will be converted to a `Struct` with the same type-conversion logic as used to create write mutations. Comparisons using `Struct` parameters are limited to [what is available with Cloud Spanner](#).

Query methods by convention

```
public interface TradeRepository extends SpannerRepository<Trade, String[]> {
    List<Trade> findByAction(String action);

    int countByAction(String action);

    // Named methods are powerful, but can get unwieldy
    List<Trade>
    findTop3DistinctByActionAndSymbolIgnoreCaseOrTraderIdOrderBySymbolDesc(
        String action, String symbol, String traderId);
}
```

In the example above, the [query methods](#) in `TradeRepository` are generated based on the name of the methods, using the [Spring Data Query creation naming convention](#).

`List<Trade> findByAction(String action)` would translate to a `SELECT * FROM trades WHERE action = ?`.

The `function` `List<Trade> findTop3DistinctByActionAndSymbolIgnoreCaseOrTraderIdOrderBySymbolDesc(String action, String symbol, String traderId);` will be translated as the equivalent of this SQL query:

```
SELECT DISTINCT * FROM trades
WHERE ACTION = ? AND LOWER(SYMBOL) = LOWER(?) AND TRADER_ID = ?
ORDER BY SYMBOL DESC
LIMIT 3
```

The following filter options are supported:

- Equality
- Greater than or equals
- Greater than
- Less than or equals
- Less than
- Is null
- Is not null
- Is true
- Is false
- Like a string
- Not like a string
- Contains a string
- Not contains a string
- In
- Not in

Note that the phrase `SymbolIgnoreCase` is translated to `LOWER(SYMBOL) = LOWER(?)` indicating a non-case-sensitive matching. The `IgnoreCase` phrase may only be appended to fields that correspond to columns of type `STRING` or `BYTES`. The Spring Data "AllIgnoreCase" phrase appended at the end of the method name is not supported.

The `Like` or `NotLike` naming conventions:

```
List<Trade> findBySymbolLike(String symbolFragment);
```


The param `symbolFragment` can contain [wildcard characters](#) for string matching such as `_` and `%`.

The `Contains` and `NotContains` naming conventions:

```
List<Trade> findBySymbolContains(String symbolFragment);
```

The param `symbolFragment` is a [regular expression](#) that is checked for occurrences.

The `In` and `NotIn` keywords must be used with `Iterable` corresponding parameters.

Delete queries are also supported. For example, query methods such as `deleteByAction` or `removeByAction` delete entities found by `findByAction`. The delete operation happens in a single transaction.

Delete queries can have the following return types: * An integer type that is the number of entities deleted * A collection of entities that were deleted * `void`

Custom SQL/DML query methods

The example above for `List<Trade> fetchByActionNamedQuery(String action)` does not match the [Spring Data Query creation naming convention](#), so we have to map a parametrized Spanner SQL query to it.

The SQL query for the method can be mapped to repository methods in one of two ways:

- `namedQueries` properties file
- using the `@Query` annotation

The names of the tags of the SQL correspond to the `@Param` annotated names of the method parameters.

Interleaved properties are loaded eagerly, unless they are annotated with `@Interleaved(lazy = true)`.

Custom SQL query methods can accept a single `Sort` or `Pageable` parameter that is applied on top of the specified custom query. It is the recommended way to control the sort order of the results, which is not guaranteed by the `ORDER BY` clause in the SQL query. This is due to the fact that the user-provided query is used as a sub-query, and Cloud Spanner doesn't preserve order in subquery results.

You might want to use `ORDER BY` with `LIMIT` to obtain the top records, according to a specified order. However, to ensure the correct sort order of the final result set, sort options have to be passed in with a `Pageable`.

```
@Query("SELECT * FROM trades")
List<Trade> fetchTrades(Pageable pageable);

@Query("SELECT * FROM trades ORDER BY price DESC LIMIT 1")
Trade topTrade(Pageable pageable);
```

This can be used:

```
List<Trade> customSortedTrades = tradeRepository.fetchTrades(PageRequest
    .of(2, 2, org.springframework.data.domain.Sort.by(Order.asc("id"))));
```

The results would be sorted by "id" in ascending order.

Your query method can also return non-entity types:

```
@Query("SELECT COUNT(1) FROM trades WHERE action = @action")
int countByActionQuery(String action);

@Query("SELECT EXISTS(SELECT COUNT(1) FROM trades WHERE action = @action)")
boolean existsByActionQuery(String action);

@Query("SELECT action FROM trades WHERE action = @action LIMIT 1")
String getFirstString(@Param("action") String action);

@Query("SELECT action FROM trades WHERE action = @action")
List<String> getFirstStringList(@Param("action") String action);
```

DML statements can also be executed by query methods, but the only possible return value is a **long** representing the number of affected rows. The **dmlStatement** boolean setting must be set on **@Query** to indicate that the query method is executed as a DML statement.

```
@Query(value = "DELETE FROM trades WHERE action = @action", dmlStatement = true)
long deleteByActionQuery(String action);
```

Query methods with named queries properties

By default, the **namedQueriesLocation** attribute on **@EnableSpannerRepositories** points to the **META-INF/spanner-named-queries.properties** file. You can specify the query for a method in the properties file by providing the SQL as the value for the "interface.method" property:

```
Trade.fetchByActionNamedQuery=SELECT * FROM trades WHERE trades.action = @tag0
```

```
public interface TradeRepository extends SpannerRepository<Trade, String[]> {
    // This method uses the query from the properties file instead of one generated
    based on name.
    List<Trade> fetchByActionNamedQuery(@Param("tag0") String action);
}
```

Query methods with annotation

Using the `@Query` annotation:

```
public interface TradeRepository extends SpannerRepository<Trade, String[]> {
    @Query("SELECT * FROM trades WHERE trades.action = @tag0")
    List<Trade> fetchByActionNamedQuery(@Param("tag0") String action);
}
```

Table names can be used directly. For example, "trades" in the above example. Alternatively, table names can be resolved from the `@Table` annotation on domain classes as well. In this case, the query should refer to table names with fully qualified class names between `:` characters: `:fully.qualified.ClassName:`. A full example would look like:

```
@Query("SELECT * FROM :com.example.Trade: WHERE trades.action = @tag0")
List<Trade> fetchByActionNamedQuery(String action);
```

This allows table names evaluated with SpEL to be used in custom queries.

SpEL can also be used to provide SQL parameters:

```
@Query("SELECT * FROM :com.example.Trade: WHERE trades.action = @tag0
AND price > #{#priceRadius * -1} AND price < #{#priceRadius * 2}")
List<Trade> fetchByActionNamedQuery(String action, Double priceRadius);
```

When using the `IN` SQL clause, remember to use `IN UNNEST(@iterableParam)` to specify a single `Iterable` parameter. You can also use a fixed number of singular parameters such as `IN (@stringParam1, @stringParam2)`.

Projections

Spring Data Spanner supports [projections](#). You can define projection interfaces based on domain types and add query methods that return them in your repository:

```

public interface TradeProjection {

    String getAction();

    @Value("#{target.symbol + ' ' + target.action}")
    String getSymbolAndAction();
}

public interface TradeRepository extends SpannerRepository<Trade, Key> {

    List<Trade> findByTraderId(String traderId);

    List<TradeProjection> findByAction(String action);

    @Query("SELECT action, symbol FROM trades WHERE action = @action")
    List<TradeProjection> findByQuery(String action);
}

```

Projections can be provided by name-convention-based query methods as well as by custom SQL queries. If using custom SQL queries, you can further restrict the columns retrieved from Spanner to just those required by the projection to improve performance.

Properties of projection types defined using SpEL use the fixed name `target` for the underlying domain object. As a result accessing underlying properties take the form `target.<property-name>`.

REST Repositories

When running with Spring Boot, repositories can be exposed as REST services by simply adding this dependency to your pom file:

```

<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-data-rest</artifactId>
</dependency>

```

If you prefer to configure parameters (such as path), you can use `@RepositoryRestResource` annotation:

```

@RepositoryRestResource(collectionResourceRel = "trades", path = "trades")
public interface TradeRepository extends SpannerRepository<Trade, Key> {
}

```



For classes that have composite keys (multiple `@PrimaryKey` fields), only the `Key` type is supported for the repository ID type.

For example, you can retrieve all `Trade` objects in the repository by using `curl`

`http://<server>:<port>/trades,` or any specific trade via `curl http://<server>:<port>/trades/<trader_id>,<trade_id>`.

The separator between your primary key components, `id` and `trader_id` in this case, is a comma by default, but can be configured to any string not found in your key values by extending the `SpannerKeyIdConverter` class:

```
@Component
class MySpecialIdConverter extends SpannerKeyIdConverter {

    @Override
    protected String getUrlIdSeparator() {
        return ":";
    }
}
```

You can also write trades using `curl -XPOST -H"Content-Type: application/json" -d@test.json http://<server>:<port>/trades/` where the file `test.json` holds the JSON representation of a `Trade` object.

19.13.6. Database and Schema Admin

Databases and tables inside Spanner instances can be created automatically from `SpannerPersistentEntity` objects:

```
@Autowired
private SpannerSchemaUtils spannerSchemaUtils;

@Autowired
private SpannerDatabaseAdminTemplate spannerDatabaseAdminTemplate;

public void createTable(SpannerPersistentEntity entity) {
    if(!spannerDatabaseAdminTemplate.tableExists(entity.tableName())){

        // The boolean parameter indicates that the database will be created if it does
        not exist.
        spannerDatabaseAdminTemplate.executeDdlStrings(Arrays.asList(
            spannerSchemaUtils.getCreateTableDDLString(entity.getType())), true);
    }
}
```

Schemas can be generated for entire object hierarchies with interleaved relationships and composite keys.

19.13.7. Events

Spring Data Cloud Spanner publishes events extending the Spring Framework's `ApplicationEvent` to the context that can be received by `ApplicationListener` beans you register.

Type	Description	Contents
<code>AfterReadEvent</code>	Published immediately after entities are read by key from Cloud Spanner by <code>SpannerTemplate</code>	The entities loaded. The read options and key-set originally specified for the load operation.
<code>AfterQueryEvent</code>	Published immediately after entities are read by query from Cloud Spanner by <code>SpannerTemplate</code>	The entities loaded. The query options and query statement originally specified for the load operation.
<code>BeforeExecuteDmlEvent</code>	Published immediately before DML statements are executed by <code>SpannerTemplate</code>	The DML statement to execute.
<code>AfterExecuteDmlEvent</code>	Published immediately after DML statements are executed by <code>SpannerTemplate</code>	The DML statement to execute and the number of rows affected by the operation as reported by Cloud Spanner.
<code>BeforeSaveEvent</code>	Published immediately before upsert/update/insert operations are executed by <code>SpannerTemplate</code>	The mutations to be sent to Cloud Spanner, the entities to be saved, and optionally the properties in those entities to save.
<code>AfterSaveEvent</code>	Published immediately after upsert/update/insert operations are executed by <code>SpannerTemplate</code>	The mutations sent to Cloud Spanner, the entities to be saved, and optionally the properties in those entities to save.
<code>BeforeDeleteEvent</code>	Published immediately before delete operations are executed by <code>SpannerTemplate</code>	The mutations to be sent to Cloud Spanner. The target entities, keys, or entity type originally specified for the delete operation.
<code>AfterDeleteEvent</code>	Published immediately after delete operations are executed by <code>SpannerTemplate</code>	The mutations sent to Cloud Spanner. The target entities, keys, or entity type originally specified for the delete operation.

19.13.8. Auditing

Spring Data Cloud Spanner supports the `@LastModifiedDate` and `@LastModifiedBy` auditing annotations for properties:

```

@Table
public class SimpleEntity {
    @PrimaryKey
    String id;

    @LastModifiedBy
    String lastUser;

    @LastModifiedDate
    DateTime lastTouched;
}

```

Upon insert, update, or save, these properties will be set automatically by the framework before mutations are generated and saved to Cloud Spanner.

To take advantage of these features, add the `@EnableSpannerAuditing` annotation to your configuration class and provide a bean for an `AuditorAware<A>` implementation where the type `A` is the desired property type annotated by `@LastModifiedBy`:

```

@Configuration
@EnableSpannerAuditing
public class Config {

    @Bean
    public AuditorAware<String> auditorProvider() {
        return () -> Optional.of("YOUR_USERNAME_HERE");
    }
}

```

The `AuditorAware` interface contains a single method that supplies the value for fields annotated by `@LastModifiedBy` and can be of any type. One alternative is to use Spring Security's `User` type:

```

class SpringSecurityAuditorAware implements AuditorAware<User> {

    public Optional<User> getCurrentAuditor() {

        return Optional.ofNullable(SecurityContextHolder.getContext())
            .map(SecurityContext::getAuthentication)
            .filter(Authentication::isAuthenticated)
            .map(Authentication::getPrincipal)
            .map(User.class::cast);
    }
}

```

You can also set a custom provider for properties annotated `@LastModifiedDate` by providing a bean for `DateTimeProvider` and providing the bean name to `@EnableSpannerAuditing(dateTimeProviderRef = "customDateTimeProviderBean")`.

19.13.9. Multi-Instance Usage

Your application can be configured to use multiple Cloud Spanner instances or databases by providing a custom bean for `DatabaseIdProvider`. The default bean uses the instance ID, database name, and project ID options you configured in `application.properties`.

```
@Bean
public DatabaseIdProvider databaseIdProvider() {
    // return custom connection options provider
}
```

The `DatabaseId` given by this provider is used as the target database name and instance of each operation Spring Data Cloud Spanner executes. By providing a custom implementation of this bean (for example, supplying a thread-local `DatabaseId`), you can direct your application to use multiple instances or databases.

Database administrative operations, such as creating tables using `SpannerDatabaseAdminTemplate`, will also utilize the provided `DatabaseId`.

If you would like to configure every aspect of each connection (such as pool size and retry settings), you can supply a bean for `Supplier<DatabaseClient>`.

19.13.10. Sample

A [sample application](#) is available.

19.14. Spring Data Cloud Datastore



This integration is fully compatible with [Firestore in Datastore Mode](#), but not with [Firestore in Native Mode](#).

[Spring Data](#) is an abstraction for storing and retrieving POJOs in numerous storage technologies. Spring Cloud GCP adds Spring Data support for [Google Cloud Firestore](#) in Datastore mode.

Maven coordinates for this module only, using [Spring Cloud GCP BOM](#):

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-gcp-data-datastore</artifactId>
</dependency>
```

Gradle coordinates:


```
dependencies {
    compile group: 'org.springframework.cloud', name: 'spring-cloud-gcp-data-
datastore'
}
```

We provide a [Spring Boot Starter for Spring Data Datastore](#), with which you can use our recommended auto-configuration setup. To use the starter, see the coordinates below.

Maven:

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-gcp-starter-data-datastore</artifactId>
</dependency>
```

Gradle:

```
dependencies {
    compile group: 'org.springframework.cloud', name: 'spring-cloud-gcp-starter-data-
datastore'
}
```

This setup takes care of bringing in the latest compatible version of Cloud Java Cloud Datastore libraries as well.

19.14.1. Configuration

To setup Spring Data Cloud Datastore, you have to configure the following:

- Setup the connection details to Google Cloud Datastore.

Cloud Datastore settings

You can use the [Spring Boot Starter for Spring Data Datastore](#) to autoconfigure Google Cloud Datastore in your Spring application. It contains all the necessary setup that makes it easy to authenticate with your Google Cloud project. The following configuration options are available:

Name	Description	Required	Default value
<code>spring.cloud.gcp.datastore.enabled</code>	Enables the Cloud Datastore client	No	<code>true</code>

<code>spring.cloud.gcp.datastore.project-id</code>	GCP project ID where the Google Cloud Datastore API is hosted, if different from the one in the Spring Cloud GCP Core Module	No	
<code>spring.cloud.gcp.datastore.credentials.location</code>	OAuth2 credentials for authenticating with the Google Cloud Datastore API, if different from the ones in the Spring Cloud GCP Core Module	No	
<code>spring.cloud.gcp.datastore.credentials.encoded-key</code>	Base64-encoded OAuth2 credentials for authenticating with the Google Cloud Datastore API, if different from the ones in the Spring Cloud GCP Core Module	No	
<code>spring.cloud.gcp.datastore.credentials.scopes</code>	OAuth2 scope for Spring Cloud GCP Cloud Datastore credentials	No	www.googleapis.com/auth/datastore
<code>spring.cloud.gcp.datastore.namespace</code>	The Cloud Datastore namespace to use	No	the Default namespace of Cloud Datastore in your GCP project
<code>spring.cloud.gcp.datastore.host</code>	The <code>hostname:port</code> of the datastore service or emulator to connect to. Can be used to connect to a manually started Datastore Emulator . If the autoconfigured emulator is enabled, this property will be ignored and <code>localhost:<emulator_port></code> will be used.	No	
<code>spring.cloud.gcp.datastore.emulator.enabled</code>	To enable the auto configuration to start a local instance of the Datastore Emulator.	No	<code>false</code>
<code>spring.cloud.gcp.datastore.emulator.port</code>	The local port to use for the Datastore Emulator	No	<code>8081</code>

<code>spring.cloud.gcp.datastore.emulator.consistency</code>	The <code>consistency</code> to use for the Datastore Emulator instance	No	0.9
--	---	----	-----

Repository settings

Spring Data Repositories can be configured via the `@EnableDatastoreRepositories` annotation on your main `@Configuration` class. With our Spring Boot Starter for Spring Data Cloud Datastore, `@EnableDatastoreRepositories` is automatically added. It is not required to add it to any other class, unless there is a need to override finer grain configuration parameters provided by `@EnableDatastoreRepositories`.

Autoconfiguration

Our Spring Boot autoconfiguration creates the following beans available in the Spring application context:

- an instance of `DatastoreTemplate`
- an instance of all user defined repositories extending `CrudRepository`, `PagingAndSortingRepository`, and `DatastoreRepository` (an extension of `PagingAndSortingRepository` with additional Cloud Datastore features) when repositories are enabled
- an instance of `Datastore` from the Google Cloud Java Client for Datastore, for convenience and lower level API access

Datastore Emulator Autoconfiguration

This Spring Boot autoconfiguration can also configure and start a local Datastore Emulator server if enabled by property.

It is useful for integration testing, but not for production.

When enabled, the `spring.cloud.gcp.datastore.host` property will be ignored and the Datastore autoconfiguration itself will be forced to connect to the autoconfigured local emulator instance.

It will create an instance of `LocalDatastoreHelper` as a bean that stores the `DatastoreOptions` to get the `Datastore` client connection to the emulator for convenience and lower level API for local access. The emulator will be properly stopped after the Spring application context shutdown.

19.14.2. Object Mapping

Spring Data Cloud Datastore allows you to map domain POJOs to Cloud Datastore kinds and entities via annotations:

```

@Entity(name = "traders")
public class Trader {

    @Id
    @Field(name = "trader_id")
    String traderId;

    String firstName;

    String lastName;

    @Transient
    Double temporaryNumber;
}

```

Spring Data Cloud Datastore will ignore any property annotated with `@Transient`. These properties will not be written to or read from Cloud Datastore.

Constructors

Simple constructors are supported on POJOs. The constructor arguments can be a subset of the persistent properties. Every constructor argument needs to have the same name and type as a persistent property on the entity and the constructor should set the property from the given argument. Arguments that are not directly set to properties are not supported.

```

@Entity(name = "traders")
public class Trader {

    @Id
    @Field(name = "trader_id")
    String traderId;

    String firstName;

    String lastName;

    @Transient
    Double temporaryNumber;

    public Trader(String traderId, String firstName) {
        this.traderId = traderId;
        this.firstName = firstName;
    }
}

```

Kind

The `@Entity` annotation can provide the name of the Cloud Datastore kind that stores instances of

the annotated class, one per row.

Keys

`@Id` identifies the property corresponding to the ID value.

You must annotate one of your POJO's fields as the ID value, because every entity in Cloud Datastore requires a single ID value:

```
@Entity(name = "trades")
public class Trade {
    @Id
    @Field(name = "trade_id")
    String tradeId;

    @Field(name = "trader_id")
    String traderId;

    String action;

    Double price;

    Double shares;

    String symbol;
}
```

Datastore can automatically allocate integer ID values. If a POJO instance with a `Long` ID property is written to Cloud Datastore with `null` as the ID value, then Spring Data Cloud Datastore will obtain a newly allocated ID value from Cloud Datastore and set that in the POJO for saving. Because primitive `long` ID properties cannot be `null` and default to `0`, keys will not be allocated.

Fields

All accessible properties on POJOs are automatically recognized as a Cloud Datastore field. Field naming is generated by the `PropertyNameFieldNamingStrategy` by default defined on the `DatastoreMappingContext` bean. The `@Field` annotation optionally provides a different field name than that of the property.

Supported Types

Spring Data Cloud Datastore supports the following types for regular fields and elements of collections:

Type	Stored as
<code>com.google.cloud.Timestamp</code>	<code>com.google.cloud.datastore.TimestampValue</code>
<code>com.google.cloud.datastore.Blob</code>	<code>com.google.cloud.datastore.BlobValue</code>
<code>com.google.cloud.datastore.LatLng</code>	<code>com.google.cloud.datastore.LatLngValue</code>

Type	Stored as
<code>java.lang.Boolean, boolean</code>	<code>com.google.cloud.datastore.BooleanValue</code>
<code>java.lang.Double, double</code>	<code>com.google.cloud.datastore.DoubleValue</code>
<code>java.lang.Long, long</code>	<code>com.google.cloud.datastore.LongValue</code>
<code>java.lang.Integer, int</code>	<code>com.google.cloud.datastore.LongValue</code>
<code>java.lang.String</code>	<code>com.google.cloud.datastore.StringValue</code>
<code>com.google.cloud.datastore.Entity</code>	<code>com.google.cloud.datastore.EntityValue</code>
<code>com.google.cloud.datastore.Key</code>	<code>com.google.cloud.datastore.KeyValue</code>
<code>byte[]</code>	<code>com.google.cloud.datastore.BlobValue</code>
Java <code>enum</code> values	<code>com.google.cloud.datastore.StringValue</code>

In addition, all types that can be converted to the ones listed in the table by `org.springframework.core.convert.support.DefaultConversionService` are supported.

Custom types

Custom converters can be used extending the type support for user defined types.

1. Converters need to implement the `org.springframework.core.convert.converter.Converter` interface in both directions.
2. The user defined type needs to be mapped to one of the basic types supported by Cloud Datastore.
3. An instance of both Converters (read and write) needs to be passed to the `DatastoreCustomConversions` constructor, which then has to be made available as a `@Bean` for `DatastoreCustomConversions`.

For example:

We would like to have a field of type `Album` on our `Singer` POJO and want it to be stored as a string property:

```
@Entity
public class Singer {

    @Id
    String singerId;

    String name;

    Album album;
}
```

Where `Album` is a simple class:

```
public class Album {
    String albumName;

    LocalDate date;
}
```

We have to define the two converters:

```
//Converter to write custom Album type
static final Converter<Album, String> ALBUM_STRING_CONVERTER =
    new Converter<Album, String>() {
        @Override
        public String convert(Album album) {
            return album.getAlbumName() + " " +
album.getDate().format(DateTimeFormatter.ISO_DATE);
        }
    };

//Converters to read custom Album type
static final Converter<String, Album> STRING_ALBUM_CONVERTER =
    new Converter<String, Album>() {
        @Override
        public Album convert(String s) {
            String[] parts = s.split(" ");
            return new Album(parts[0], LocalDate.parse(parts[parts.length -
1], DateTimeFormatter.ISO_DATE));
        }
    };
};
```

That will be configured in our `@Configuration` file:

```
@Configuration
public class ConverterConfiguration {
    @Bean
    public DatastoreCustomConversions datastoreCustomConversions() {
        return new DatastoreCustomConversions(
            Arrays.asList(
                ALBUM_STRING_CONVERTER,
                STRING_ALBUM_CONVERTER));
    }
}
```

Collections and arrays

Arrays and collections (types that implement `java.util.Collection`) of supported types are supported. They are stored as `com.google.cloud.datastore.ListValue`. Elements are converted to Cloud Datastore supported types individually. `byte[]` is an exception, it is converted to

`com.google.cloud.datastore.Blob`.

Custom Converter for collections

Users can provide converters from `List<?>` to the custom collection type. Only read converter is necessary, the Collection API is used on the write side to convert a collection to the internal list type.

Collection converters need to implement the `org.springframework.core.convert.converter.Converter` interface.

Example:

Let's improve the Singer class from the previous example. Instead of a field of type `Album`, we would like to have a field of type `Set<Album>`:

```
@Entity
public class Singer {

    @Id
    String singerId;

    String name;

    Set<Album> albums;
}
```

We have to define a read converter only:

```
static final Converter<List<?>, Set<?>> LIST_SET_CONVERTER =
    new Converter<List<?>, Set<?>>() {
        @Override
        public Set<?> convert(List<?> source) {
            return Collections.unmodifiableSet(new HashSet<>(source));
        }
    };
```

And add it to the list of custom converters:


```
@Configuration
public class ConverterConfiguration {
    @Bean
    public DatastoreCustomConversions datastoreCustomConversions() {
        return new DatastoreCustomConversions(
            Arrays.asList(
                LIST_SET_CONVERTER,
                ALBUM_STRING_CONVERTER,
                STRING_ALBUM_CONVERTER));
    }
}
```

Inheritance Hierarchies

Java entity types related by inheritance can be stored in the same Kind. When reading and querying entities using `DatastoreRepository` or `DatastoreTemplate` with a superclass as the type parameter, you can receive instances of subclasses if you annotate the superclass and its subclasses with `DiscriminatorField` and `DiscriminatorValue`:

```

@Entity(name = "pets")
@DiscriminatorField(field = "pet_type")
abstract class Pet {
    @Id
    Long id;

    abstract String speak();
}

@DiscriminatorValue("cat")
class Cat extends Pet {
    @Override
    String speak() {
        return "meow";
    }
}

@DiscriminatorValue("dog")
class Dog extends Pet {
    @Override
    String speak() {
        return "woof";
    }
}

@DiscriminatorValue("pug")
class Pug extends Dog {
    @Override
    String speak() {
        return "woof woof";
    }
}

```

Instances of all 3 types are stored in the `pets` Kind. Because a single Kind is used, all classes in the hierarchy must share the same ID property and no two instances of any type in the hierarchy can share the same ID value.

Entity rows in Cloud Datastore store their respective types' `DiscriminatorValue` in a field specified by the root superclass's `DiscriminatorField` (`pet_type` in this case). Reads and queries using a given type parameter will match each entity with its specific type. For example, reading a `List<Pet>` will produce a list containing instances of all 3 types. However, reading a `List<Dog>` will produce a list containing only `Dog` and `Pug` instances. You can include the `pet_type` discrimination field in your Java entities, but its type must be convertible to a collection or array of `String`. Any value set in the discrimination field will be overwritten upon write to Cloud Datastore.

19.14.3. Relationships

There are three ways to represent relationships between entities that are described in this section:

- Embedded entities stored directly in the field of the containing entity
- `@Descendant` annotated properties for one-to-many relationships
- `@Reference` annotated properties for general relationships without hierarchy
- `@LazyReference` similar to `@Reference`, but the entities are lazy-loaded when the property is accessed. (Note that the keys of the children are retrieved when the parent entity is loaded.)

Embedded Entities

Fields whose types are also annotated with `@Entity` are converted to `EntityValue` and stored inside the parent entity.

Here is an example of Cloud Datastore entity containing an embedded entity in JSON:

```
{
  "name" : "Alexander",
  "age" : 47,
  "child" : {"name" : "Philip" }
}
```

This corresponds to a simple pair of Java entities:

```
import org.springframework.cloud.gcp.data.datastore.core.mapping.Entity;
import org.springframework.data.annotation.Id;

@Entity("parents")
public class Parent {
    @Id
    String name;

    Child child;
}

@Entity
public class Child {
    String name;
}
```

`Child` entities are not stored in their own kind. They are stored in their entirety in the `child` field of the `parents` kind.

Multiple levels of embedded entities are supported.



Embedded entities don't need to have `@Id` field, it is only required for top level entities.

Example:

Entities can hold embedded entities that are their own type. We can store trees in Cloud Datastore using this feature:

```
import org.springframework.cloud.gcp.data.datastore.core.mapping.Embedded;
import org.springframework.cloud.gcp.data.datastore.core.mapping.Entity;
import org.springframework.data.annotation.Id;

@Entity
public class EmbeddableTreeNode {
    @Id
    long value;

    EmbeddableTreeNode left;

    EmbeddableTreeNode right;

    Map<String, Long> longValues;

    Map<String, List<Timestamp>> listTimestamps;

    public EmbeddableTreeNode(long value, EmbeddableTreeNode left, EmbeddableTreeNode
right) {
        this.value = value;
        this.left = left;
        this.right = right;
    }
}
```

Maps

Maps will be stored as embedded entities where the key values become the field names in the embedded entity. The value types in these maps can be any regularly supported property type, and the key values will be converted to String using the configured converters.

Also, a collection of entities can be embedded; it will be converted to `ListValue` on write.

Example:

Instead of a binary tree from the previous example, we would like to store a general tree (each node can have an arbitrary number of children) in Cloud Datastore. To do that, we need to create a field of type `List<EmbeddableTreeNode>`:

```
import org.springframework.cloud.gcp.data.datastore.core.mapping.Embedded;
import org.springframework.data.annotation.Id;

public class EmbeddableTreeNode {
    @Id
    long value;

    List<EmbeddableTreeNode> children;

    Map<String, EmbeddableTreeNode> siblingNodes;

    Map<String, Set<EmbeddableTreeNode>> subNodeGroups;

    public EmbeddableTreeNode(List<EmbeddableTreeNode> children) {
        this.children = children;
    }
}
```

Because Maps are stored as entities, they can further hold embedded entities:

- Singular embedded objects in the value can be stored in the values of embedded Maps.
- Collections of embedded objects in the value can also be stored as the values of embedded Maps.
- Maps in the value are further stored as embedded entities with the same rules applied recursively for their values.

Ancestor-Descendant Relationships

Parent-child relationships are supported via the `@Descendants` annotation.

Unlike embedded children, descendants are fully-formed entities residing in their own kinds. The parent entity does not have an extra field to hold the descendant entities. Instead, the relationship is captured in the descendants' keys, which refer to their parent entities:

```

import org.springframework.cloud.gcp.data.datastore.core.mapping.Descendants;
import org.springframework.cloud.gcp.data.datastore.core.mapping.Entity;
import org.springframework.data.annotation.Id;

@Entity("orders")
public class ShoppingOrder {
    @Id
    long id;

    @Descendants
    List<Item> items;
}

@Entity("purchased_item")
public class Item {
    @Id
    Key purchasedItemKey;

    String name;

    Timestamp timeAddedToOrder;
}

```

For example, an instance of a GQL key-literal representation for `Item` would also contain the parent `ShoppingOrder` ID value:

```
Key(orders, '12345', purchased_item, 'eggs')
```

The GQL key-literal representation for the parent `ShoppingOrder` would be:

```
Key(orders, '12345')
```

The Cloud Datastore entities exist separately in their own kinds.

The `ShoppingOrder`:

```
{
  "id" : 12345
}
```

The two items inside that order:

```
{
  "purchasedItemKey" : Key(orders, '12345', purchased_item, 'eggs'),
  "name" : "eggs",
  "timeAddedToOrder" : "2014-09-27 12:30:00.45-8:00"
}

{
  "purchasedItemKey" : Key(orders, '12345', purchased_item, 'sausage'),
  "name" : "sausage",
  "timeAddedToOrder" : "2014-09-28 11:30:00.45-9:00"
}
```

The parent-child relationship structure of objects is stored in Cloud Datastore using Datastore's [ancestor relationships](#). Because the relationships are defined by the Ancestor mechanism, there is no extra column needed in either the parent or child entity to store this relationship. The relationship link is part of the descendant entity's key value. These relationships can be many levels deep.

Properties holding child entities must be collection-like, but they can be any of the supported inter-convertible collection-like types that are supported for regular properties such as [List](#), arrays, [Set](#), etc... Child items must have [Key](#) as their ID type because Cloud Datastore stores the ancestor relationship link inside the keys of the children.

Reading or saving an entity automatically causes all subsequent levels of children under that entity to be read or saved, respectively. If a new child is created and added to a property annotated [@Descendants](#) and the key property is left null, then a new key will be allocated for that child. The ordering of the retrieved children may not be the same as the ordering in the original property that was saved.

Child entities cannot be moved from the property of one parent to that of another unless the child's key property is set to [null](#) or a value that contains the new parent as an ancestor. Since Cloud Datastore entity keys can have multiple parents, it is possible that a child entity appears in the property of multiple parent entities. Because entity keys are immutable in Cloud Datastore, to change the key of a child you must delete the existing one and re-save it with the new key.

Key Reference Relationships

General relationships can be stored using the [@Reference](#) annotation.

```

import org.springframework.data.annotation.Reference;
import org.springframework.data.annotation.Id;

@Entity
public class ShoppingOrder {
    @Id
    long id;

    @Reference
    List<Item> items;

    @Reference
    Item specialSingleItem;
}

@Entity
public class Item {
    @Id
    Key purchasedItemKey;

    String name;

    Timestamp timeAddedToOrder;
}

```

@Reference relationships are between fully-formed entities residing in their own kinds. The relationship between **ShoppingOrder** and **Item** entities are stored as a **Key** field inside **ShoppingOrder**, which are resolved to the underlying Java entity type by Spring Data Cloud Datastore:

```

{
  "id" : 12345,
  "specialSingleItem" : Key(item, "milk"),
  "items" : [ Key(item, "eggs"), Key(item, "sausage") ]
}

```

Reference properties can either be singular or collection-like. These properties correspond to actual columns in the entity and Cloud Datastore Kind that hold the key values of the referenced entities. The referenced entities are full-fledged entities of other Kinds.

Similar to the **@Descendants** relationships, reading or writing an entity will recursively read or write all of the referenced entities at all levels. If referenced entities have **null** ID values, then they will be saved as new entities and will have ID values allocated by Cloud Datastore. There are no requirements for relationships between the key of an entity and the keys that entity holds as references. The order of collection-like reference properties is not preserved when reading back from Cloud Datastore.

19.14.4. Datastore Operations & Template

`DatastoreOperations` and its implementation, `DatastoreTemplate`, provides the Template pattern familiar to Spring developers.

Using the auto-configuration provided by Spring Boot Starter for Datastore, your Spring application context will contain a fully configured `DatastoreTemplate` object that you can autowire in your application:

```
@SpringBootApplication
public class DatastoreTemplateExample {

    @Autowired
    DatastoreTemplate datastoreTemplate;

    public void doSomething() {
        this.datastoreTemplate.deleteAll(Trader.class);
        //...
        Trader t = new Trader();
        //...
        this.datastoreTemplate.save(t);
        //...
        List<Trader> traders = datastoreTemplate.findAll(Trader.class);
        //...
    }
}
```

The Template API provides convenience methods for:

- Write operations (saving and deleting)
- Read-write transactions

GQL Query

In addition to retrieving entities by their IDs, you can also submit queries.

```
<T> Iterable<T> query(Query<? extends BaseEntity> query, Class<T> entityClass);

<A, T> Iterable<T> query(Query<A> query, Function<A, T> entityFunc);

Iterable<Key> queryKeys(Query<Key> query);
```

These methods, respectively, allow querying for: * entities mapped by a given entity class using all the same mapping and converting features * arbitrary types produced by a given mapping function * only the Cloud Datastore keys of the entities found by the query

Find by ID(s)

Using `DatastoreTemplate` you can find entities by id. For example:

```
Trader trader = this.datastoreTemplate.findById("trader1", Trader.class);

List<Trader> traders = this.datastoreTemplate.findAllById(Arrays.asList("trader1",
"trader2"), Trader.class);

List<Trader> allTraders = this.datastoreTemplate.findAll(Trader.class);
```

Cloud Datastore executes key-based reads with strong consistency, but queries with eventual consistency. In the example above the first two reads utilize keys, while the third is executed using a query based on the corresponding Kind of `Trader`.

Indexes

By default, all fields are indexed. To disable indexing on a particular field, `@Unindexed` annotation can be used.

Example:

```
import org.springframework.cloud.gcp.data.datastore.core.mapping.Unindexed;

public class ExampleItem {
    long indexedField;

    @Unindexed
    long unindexedField;

    @Unindexed
    List<String> unindexedListField;
}
```

When using queries directly or via Query Methods, Cloud Datastore requires [composite custom indexes](#) if the select statement is not `SELECT *` or if there is more than one filtering condition in the `WHERE` clause.

Read with offsets, limits, and sorting

`DatastoreRepository` and custom-defined entity repositories implement the Spring Data `PagingAndSortingRepository`, which supports offsets and limits using page numbers and page sizes. Paging and sorting options are also supported in `DatastoreTemplate` by supplying a `DatastoreQueryOptions` to `findAll`.

Partial read

This feature is not supported yet.

Write / Update

The write methods of `DatastoreOperations` accept a POJO and writes all of its properties to Datastore. The required Datastore kind and entity metadata is obtained from the given object's actual type.

If a POJO was retrieved from Datastore and its ID value was changed and then written or updated, the operation will occur as if against a row with the new ID value. The entity with the original ID value will not be affected.

```
Trader t = new Trader();
this.datastoreTemplate.save(t);
```

The `save` method behaves as update-or-insert.

Partial Update

This feature is not supported yet.

Transactions

Read and write transactions are provided by `DatastoreOperations` via the `performTransaction` method:

```
@Autowired
DatastoreOperations myDatastoreOperations;

public String doWorkInsideTransaction() {
    return myDatastoreOperations.performTransaction(
        transactionDatastoreOperations -> {
            // Work with transactionDatastoreOperations here.
            // It is also a DatastoreOperations object.

            return "transaction completed";
        }
    );
}
```

The `performTransaction` method accepts a `Function` that is provided an instance of a `DatastoreOperations` object. The final returned value and type of the function is determined by the user. You can use this object just as you would a regular `DatastoreOperations` with an exception:

- It cannot perform sub-transactions.

Because of Cloud Datastore's consistency guarantees, there are [limitations](#) to the operations and relationships among entities used inside transactions.

Declarative Transactions with @Transactional Annotation

This feature requires a bean of `DatastoreTransactionManager`, which is provided when using `spring-cloud-gcp-starter-data-datastore`.

`DatastoreTemplate` and `DatastoreRepository` support running methods with the `@Transactional` annotation as transactions. If a method annotated with `@Transactional` calls another method also annotated, then both methods will work within the same transaction. `performTransaction` cannot be used in `@Transactional` annotated methods because Cloud Datastore does not support transactions within transactions.

Read-Write Support for Maps

You can work with Maps of type `Map<String, ?>` instead of with entity objects by directly reading and writing them to and from Cloud Datastore.



This is a different situation than using entity objects that contain Map properties.

The map keys are used as field names for a Datastore entity and map values are converted to Datastore supported types. Only simple types are supported (i.e. collections are not supported). Converters for custom value types can be added (see [Custom types](#) section).

Example:

```
Map<String, Long> map = new HashMap<>();
map.put("field1", 1L);
map.put("field2", 2L);
map.put("field3", 3L);

keyForMap = datastoreTemplate.createKey("kindName", "id");

//write a map
datastoreTemplate.writeMap(keyForMap, map);

//read a map
Map<String, Long> loadedMap = datastoreTemplate.findByIdAsMap(keyForMap, Long.class);
```

19.14.5. Repositories

[Spring Data Repositories](#) are an abstraction that can reduce boilerplate code.

For example:

```
public interface TraderRepository extends DatastoreRepository<Trader, String> {
}
```

Spring Data generates a working implementation of the specified interface, which can be autowired into an application.

The `Trader` type parameter to `DatastoreRepository` refers to the underlying domain type. The second type parameter, `String` in this case, refers to the type of the key of the domain type.

```
public class MyApplication {

    @Autowired
    TraderRepository traderRepository;

    public void demo() {

        this.traderRepository.deleteAll();
        String traderId = "demo_trader";
        Trader t = new Trader();
        t.traderId = traderId;
        this.traderRepository.save(t);

        Iterable<Trader> allTraders = this.traderRepository.findAll();

        int count = this.traderRepository.count();
    }
}
```

Repositories allow you to define custom Query Methods (detailed in the following sections) for retrieving, counting, and deleting based on filtering and paging parameters. Filtering parameters can be of types supported by your configured custom converters.

Query methods by convention

```

public interface TradeRepository extends DatastoreRepository<Trade, String[]> {
    List<Trader> findByAction(String action);

    //throws an exception if no results
    Trader findOneByAction(String action);

    //because of the annotation, returns null if no results
    @Nullable
    Trader getByAction(String action);

    Optional<Trader> getOneByAction(String action);

    int countByAction(String action);

    boolean existsByAction(String action);

    List<Trade>
    findTop3ByActionAndSymbolAndPriceGreaterThanAndPriceLessThanOrEqualToOrderBySymbolDesc(
        String action, String symbol, double priceFloor, double priceCeiling);

    Page<TestEntity> findByAction(String action, Pageable pageable);

    Slice<TestEntity> findBySymbol(String symbol, Pageable pageable);

    List<TestEntity> findBySymbol(String symbol, Sort sort);
}

```

In the example above the [query methods](#) in `TradeRepository` are generated based on the name of the methods using the [Spring Data Query creation naming convention](#).

Cloud Datastore only supports filter components joined by AND, and the following operations:

- `equals`
- `greater than or equals`
- `greater than`
- `less than or equals`
- `less than`
- `is null`

After writing a custom repository interface specifying just the signatures of these methods, implementations are generated for you and can be used with an auto-wired instance of the repository. Because of Cloud Datastore's requirement that explicitly selected fields must all appear in a composite index together, `find` name-based query methods are run as `SELECT *`.

Delete queries are also supported. For example, query methods such as `deleteByAction` or `removeByAction` delete entities found by `findByAction`. Delete queries are executed as separate read and delete operations instead of as a single transaction because Cloud Datastore cannot query in transactions unless ancestors for queries are specified. As a result, `removeBy` and `deleteBy` name-convention query methods cannot be used inside transactions via either `performInTransaction` or

`@Transactional` annotation.

Delete queries can have the following return types:

- An integer type that is the number of entities deleted
- A collection of entities that were deleted
- 'void'

Methods can have `org.springframework.data.domain.Pageable` parameter to control pagination and sorting, or `org.springframework.data.domain.Sort` parameter to control sorting only. See [Spring Data documentation](#) for details.

For returning multiple items in a repository method, we support Java collections as well as `org.springframework.data.domain.Page` and `org.springframework.data.domain.Slice`. If a method's return type is `org.springframework.data.domain.Page`, the returned object will include current page, total number of results and total number of pages.



Methods that return `Page` execute an additional query to compute total number of pages. Methods that return `Slice`, on the other hand, don't execute any additional queries and therefore are much more efficient.

Query by example

Query by Example is an alternative querying technique. It enables dynamic query generation based on a user-provided object. See [Spring Data Documentation](#) for details.

Unsupported features:

1. Currently, only equality queries are supported (no ignore-case matching, regexp matching, etc.).
2. Per-field matchers are not supported.
3. Embedded entities matching is not supported.

For example, if you want to find all users with the last name "Smith", you would use the following code:

```
userRepository.findAll(  
    Example.of(new User(null, null, "Smith"))
```

`null` fields are not used in the filter by default. If you want to include them, you would use the following code:

```
userRepository.findAll(  
    Example.of(new User(null, null, "Smith"),  
    ExampleMatcher.matching().withIncludeNullValues())
```

Custom GQL query methods

Custom GQL queries can be mapped to repository methods in one of two ways:

- `namedQueries` properties file
- using the `@Query` annotation

Query methods with annotation

Using the `@Query` annotation:

The names of the tags of the GQL correspond to the `@Param` annotated names of the method parameters.

```
public interface TraderRepository extends DatastoreRepository<Trader, String> {

    @Query("SELECT * FROM traders WHERE name = @trader_name")
    List<Trader> tradersByName(@Param("trader_name") String traderName);

    @Query("SELECT * FROM test_entities_ci WHERE name = @trader_name")
    TestEntity getOneTestEntity(@Param("trader_name") String traderName);

    @Query("SELECT * FROM traders WHERE name = @trader_name")
    List<Trader> tradersByNameSort(@Param("trader_name") String traderName, Sort sort);

    @Query("SELECT * FROM traders WHERE name = @trader_name")
    Slice<Trader> tradersByNameSlice(@Param("trader_name") String traderName, Pageable
pageable);

    @Query("SELECT * FROM traders WHERE name = @trader_name")
    Page<Trader> tradersByNamePage(@Param("trader_name") String traderName, Pageable
pageable);
}
```

When the return type is `Slice` or `Pageable`, the result set cursor that points to the position just after the page is preserved in the returned `Slice` or `Page` object. To take advantage of the cursor to query for the next page or slice, use `result.getPageable().next()`.



`Page` requires the total count of entities produced by the query. Therefore, the first query will have to retrieve all of the records just to count them. Instead, we recommend using the `Slice` return type, because it does not require an additional count query.

```
Slice<Trader> slice1 = tradersByNamePage("Dave", PageRequest.of(0, 5));
Slice<Trader> slice2 = tradersByNamePage("Dave", slice1.getPageable().next());
```




You cannot use these Query Methods in repositories where the type parameter is a subclass of another class annotated with `DiscriminatorField`.

The following parameter types are supported:

- `com.google.cloud.Timestamp`
- `com.google.cloud.datastore.Blob`
- `com.google.cloud.datastore.Key`
- `com.google.cloud.datastore.Cursor`
- `java.lang.Boolean`
- `java.lang.Double`
- `java.lang.Long`
- `java.lang.String`
- `enum` values. These are queried as `String` values.

With the exception of `Cursor`, array forms of each of the types are also supported.

If you would like to obtain the count of items of a query or if there are any items returned by the query, set the `count = true` or `exists = true` properties of the `@Query` annotation, respectively. The return type of the query method in these cases should be an integer type or a boolean type.

Cloud Datastore provides the `SELECT __key__ FROM ...` special column for all kinds that retrieves the `Key` of each row. Selecting this special `__key__` column is especially useful and efficient for `count` and `exists` queries.

You can also query for non-entity types:

```
@Query(value = "SELECT __key__ from test_entities_ci")
List<Key> getKeys();

@Query(value = "SELECT __key__ from test_entities_ci limit 1")
Key getKey();
```

In order to use `@Id` annotated fields in custom queries, use `__key__` keyword for the field name. The parameter type should be of `Key`, as in the following example.

Repository method:

```
@Query("select * from test_entities_ci where size = @size and __key__ = @id")
LinkedList<TestEntity> findEntities(@Param("size") long size, @Param("id") Key id);
```

Generate a key from id value using `DatastoreTemplate.createKey` method and use it as a parameter for the repository method:

```
this.testEntityRepository.findEntities(1L,
datastoreTemplate.createKey(TestEntity.class, 1L))
```

SpEL can be used to provide GQL parameters:

```
@Query("SELECT * FROM |com.example.Trade| WHERE trades.action = @act
AND price > :#{#priceRadius * -1} AND price < :#{#priceRadius * 2}")
List<Trade> fetchByActionNamedQuery(@Param("act") String action, @Param("priceRadius")
Double r);
```

Kind names can be directly written in the GQL annotations. Kind names can also be resolved from the `@Entity` annotation on domain classes.

In this case, the query should refer to table names with fully qualified class names surrounded by `|` characters: `|fully.qualified.ClassName|`. This is useful when SpEL expressions appear in the kind name provided to the `@Entity` annotation. For example:

```
@Query("SELECT * FROM |com.example.Trade| WHERE trades.action = @act")
List<Trade> fetchByActionNamedQuery(@Param("act") String action);
```

Query methods with named queries properties

You can also specify queries with Cloud Datastore parameter tags and SpEL expressions in properties files.

By default, the `namedQueriesLocation` attribute on `@EnableDatastoreRepositories` points to the `META-INF/datastore-named-queries.properties` file. You can specify the query for a method in the properties file by providing the GQL as the value for the "interface.method" property:



You cannot use these Query Methods in repositories where the type parameter is a subclass of another class annotated with `DiscriminatorField`.

```
Trader.fetchByName=SELECT * FROM traders WHERE name = @tag0
```

```
public interface TraderRepository extends DatastoreRepository<Trader, String> {

    // This method uses the query from the properties file instead of one generated
    based on name.
    List<Trader> fetchByName(@Param("tag0") String traderName);

}
```

Transactions

These transactions work very similarly to those of [DatastoreOperations](#), but is specific to the repository's domain type and provides repository functions instead of template functions.

For example, this is a read-write transaction:

```
@Autowired
DatastoreRepository myRepo;

public String doWorkInsideTransaction() {
    return myRepo.performTransaction(
        transactionDatastoreRepo -> {
            // Work with the single-transaction transactionDatastoreRepo here.
            // This is a DatastoreRepository object.

            return "transaction completed";
        }
    );
}
```

Projections

Spring Data Cloud Datastore supports [projections](#). You can define projection interfaces based on domain types and add query methods that return them in your repository:

```
public interface TradeProjection {

    String getAction();

    @Value("#{target.symbol + ' ' + target.action}")
    String getSymbolAndAction();
}

public interface TradeRepository extends DatastoreRepository<Trade, Key> {

    List<Trade> findByTraderId(String traderId);

    List<TradeProjection> findByAction(String action);

    @Query("SELECT action, symbol FROM trades WHERE action = @action")
    List<TradeProjection> findByQuery(String action);
}
```

Projections can be provided by name-convention-based query methods as well as by custom GQL queries. If using custom GQL queries, you can further restrict the fields retrieved from Cloud Datastore to just those required by the projection. However, custom select statements (those not using **SELECT ***) require composite indexes containing the selected fields.

Properties of projection types defined using SpEL use the fixed name `target` for the underlying domain object. As a result, accessing underlying properties take the form `target.<property-name>`.

REST Repositories

When running with Spring Boot, repositories can be exposed as REST services by simply adding this dependency to your pom file:

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-data-rest</artifactId>
</dependency>
```

If you prefer to configure parameters (such as path), you can use `@RepositoryRestResource` annotation:

```
@RepositoryRestResource(collectionResourceRel = "trades", path = "trades")
public interface TradeRepository extends DatastoreRepository<Trade, String[]> {
}
```

For example, you can retrieve all `Trade` objects in the repository by using `curl http://<server>:<port>/trades`, or any specific trade via `curl http://<server>:<port>/trades/<trader_id>`.

You can also write trades using `curl -XPOST -H"Content-Type: application/json" -d@test.json http://<server>:<port>/trades/` where the file `test.json` holds the JSON representation of a `Trade` object.

To delete trades, you can use `curl -XDELETE http://<server>:<port>/trades/<trader_id>`

19.14.6. Events

Spring Data Cloud Datastore publishes events extending the Spring Framework's `ApplicationEvent` to the context that can be received by `ApplicationListener` beans you register.

Type	Description	Contents
<code>AfterFindByKeyEvent</code>	Published immediately after read by-key operations are executed by <code>DatastoreTemplate</code>	The entities read from Cloud Datastore and the original keys in the request.
<code>AfterQueryEvent</code>	Published immediately after read byquery operations are executed by <code>DatastoreTemplate</code>	The entities read from Cloud Datastore and the original query in the request.
<code>BeforeSaveEvent</code>	Published immediately before save operations are executed by <code>DatastoreTemplate</code>	The entities to be sent to Cloud Datastore and the original Java objects being saved.

Type	Description	Contents
<code>AfterSaveEvent</code>	Published immediately after save operations are executed by <code>DatastoreTemplate</code>	The entities sent to Cloud Datastore and the original Java objects being saved.
<code>BeforeDeleteEvent</code>	Published immediately before delete operations are executed by <code>DatastoreTemplate</code>	The keys to be sent to Cloud Datastore. The target entities, ID values, or entity type originally specified for the delete operation.
<code>AfterDeleteEvent</code>	Published immediately after delete operations are executed by <code>DatastoreTemplate</code>	The keys sent to Cloud Datastore. The target entities, ID values, or entity type originally specified for the delete operation.

19.14.7. Auditing

Spring Data Cloud Datastore supports the `@LastModifiedDate` and `@LastModifiedBy` auditing annotations for properties:

```
@Entity
public class SimpleEntity {
    @Id
    String id;

    @LastModifiedBy
    String lastUser;

    @LastModifiedDate
    DateTime lastTouched;
}
```

Upon insert, update, or save, these properties will be set automatically by the framework before Datastore entities are generated and saved to Cloud Datastore.

To take advantage of these features, add the `@EnableDatastoreAuditing` annotation to your configuration class and provide a bean for an `AuditorAware<A>` implementation where the type `A` is the desired property type annotated by `@LastModifiedBy`:

```

@Configuration
@EnableDatastoreAuditing
public class Config {

    @Bean
    public AuditorAware<String> auditorProvider() {
        return () -> Optional.of("YOUR_USERNAME_HERE");
    }
}

```

The `AuditorAware` interface contains a single method that supplies the value for fields annotated by `@LastModifiedBy` and can be of any type. One alternative is to use Spring Security's `User` type:

```

class SpringSecurityAuditorAware implements AuditorAware<User> {

    public Optional<User> getCurrentAuditor() {

        return Optional.ofNullable(SecurityContextHolder.getContext())
            .map(SecurityContext::getAuthentication)
            .filter(Authentication::isAuthenticated)
            .map(Authentication::getPrincipal)
            .map(User.class::cast);
    }
}

```

You can also set a custom provider for properties annotated `@LastModifiedDate` by providing a bean for `DateProvider` and providing the bean name to `@EnableDatastoreAuditing(dateTimeProviderRef = "customDateProviderBean")`.

19.14.8. Partitioning Data by Namespace

You can [partition your data by using more than one namespace](#). This is the recommended method for multi-tenancy in Cloud Datastore.

```

@Bean
public DatastoreNamespaceProvider namespaceProvider() {
    // return custom Supplier of a namespace string.
}

```

The `DatastoreNamespaceProvider` is a synonym for `Supplier<String>`. By providing a custom implementation of this bean (for example, supplying a thread-local namespace name), you can direct your application to use multiple namespaces. Every read, write, query, and transaction you perform will utilize the namespace provided by this supplier.

Note that your provided namespace in `application.properties` will be ignored if you define a namespace provider bean.

19.14.9. Spring Boot Actuator Support

Cloud Datastore Health Indicator

If you are using Spring Boot Actuator, you can take advantage of the Cloud Datastore health indicator called `datastore`. The health indicator will verify whether Cloud Datastore is up and accessible by your application. To enable it, all you need to do is add the [Spring Boot Actuator](#) to your project.

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
```

19.14.10. Sample

A [Simple Spring Boot Application](#) and more advanced [Sample Spring Boot Application](#) are provided to show how to use the Spring Data Cloud Datastore starter and template.

19.15. Spring Data Reactive Repositories for Cloud Firestore



Currently some features are not supported: transactions, sorting, query by example, projections, auditing.

[Spring Data](#) is an abstraction for storing and retrieving POJOs in numerous storage technologies. Spring Cloud GCP adds Spring Data support for [Google Cloud Firestore](#) in native mode, providing reactive template and repositories support. To begin using this library, add the `spring-cloud-gcp-data-firestore` artifact to your project.

Maven coordinates for this module only, using [Spring Cloud GCP BOM](#):

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-gcp-data-firestore</artifactId>
</dependency>
```

Gradle coordinates:

```
dependencies {
  compile group: 'org.springframework.cloud', name: 'spring-cloud-gcp-data-firestore'
}
```

We provide a Spring Boot Starter for Spring Data Firestore, with which you can use our recommended auto-configuration setup. To use the starter, see the coordinates below.

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-gcp-starter-data-firestore</artifactId>
</dependency>
```

Gradle coordinates:

```
dependencies {
  compile group: 'org.springframework.cloud', name: 'spring-cloud-gcp-starter-data-
  firestore'
}
```

19.15.1. Configuration

Properties

The Spring Boot starter for Google Cloud Firestore provides the following configuration options:

Name	Description	Required	Default value
<code>spring.cloud.gcp.firestore.enabled</code>	Enables or disables Firestore auto-configuration	No	<code>true</code>
<code>spring.cloud.gcp.firestore.project-id</code>	GCP project ID where the Google Cloud Firestore API is hosted, if different from the one in the Spring Cloud GCP Core Module	No	
<code>spring.cloud.gcp.firestore.credentials.location</code>	OAuth2 credentials for authenticating with the Google Cloud Firestore API, if different from the ones in the Spring Cloud GCP Core Module	No	
<code>spring.cloud.gcp.firestore.credentials.encoded-key</code>	Base64-encoded OAuth2 credentials for authenticating with the Google Cloud Firestore API, if different from the ones in the Spring Cloud GCP Core Module	No	
<code>spring.cloud.gcp.firestore.credentials.scopes</code>	OAuth2 scope for Spring Cloud GCP Cloud Firestore credentials	No	www.googleapis.com/auth/datastore

Supported types

You may use the following field types when defining your persistent entities or when binding query parameters:

- Long
- Integer
- Double
- Float
- String
- Boolean
- Character
- Date
- Map
- List
- Enum
- `com.google.cloud.Timestamp`
- `com.google.cloud.firestore.GeoPoint`
- `com.google.cloud.firestore.Blob`

Reactive Repository settings

Spring Data Repositories can be configured via the `@EnableReactiveFirestoreRepositories` annotation on your main `@Configuration` class. With our Spring Boot Starter for Spring Data Cloud Firestore, `@EnableReactiveFirestoreRepositories` is automatically added. It is not required to add it to any other class, unless there is a need to override finer grain configuration parameters provided by `@EnableReactiveFirestoreRepositories`.

Autoconfiguration

Our Spring Boot autoconfiguration creates the following beans available in the Spring application context:

- an instance of `FirestoreTemplate`
- instances of all user defined repositories extending `FirestoreReactiveRepository` (an extension of `ReactiveCrudRepository` with additional Cloud Firestore features) when repositories are enabled
- an instance of `Firestore` from the Google Cloud Java Client for Firestore, for convenience and lower level API access

19.15.2. Object Mapping

Spring Data Cloud Firestore allows you to map domain POJOs to [Cloud Firestore collections](#) and documents via annotations:

```

import com.google.cloud.firestore.annotation.DocumentId;
import org.springframework.cloud.gcp.data.firestore.Document;

@Document(collectionName = "usersCollection")
public class User {
    @DocumentId
    private String name;

    private Integer age;

    public User() {
    }

    public String getName() {
        return this.name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public Integer getAge() {
        return this.age;
    }

    public void setAge(Integer age) {
        this.age = age;
    }
}

```

`@Document(collectionName = "usersCollection")` annotation configures the collection name for the documents of this type. This annotation is optional, by default the collection name is derived from the class name.

`@DocumentId` annotation marks a field to be used as document id. This annotation is required.



Internally we use Firestore client library object mapping. See [the documentation](#) for supported annotations.

Embedded entities and lists

Spring Data Cloud Firestore supports embedded properties of custom types and lists. Given a custom POJO definition, you can have properties of this type or lists of this type in your entities. They are stored as embedded documents (or arrays, correspondingly) in the Cloud Firestore.

Example:

```

@Document(collectionName = "usersCollection")
public class User {
    @DocumentId
    private String name;

    private Integer age;

    private List<String> pets;

    private List<Address> addresses;

    private Address homeAddress;

    public List<String> getPets() {
        return this.pets;
    }

    public void setPets(List<String> pets) {
        this.pets = pets;
    }

    public List<Address> getAddresses() {
        return this.addresses;
    }

    public void setAddresses(List<Address> addresses) {
        this.addresses = addresses;
    }

    public Address getHomeAddress() {
        return this.homeAddress;
    }

    public void setHomeAddress(Address homeAddress) {
        this.homeAddress = homeAddress;
    }

    public static class Address {
        String streetAddress;
        String country;

        public Address() {
        }
    }
}

```

19.15.3. Reactive Repositories

[Spring Data Repositories](#) is an abstraction that can reduce boilerplate code.

For example:

```
public interface UserRepository extends FirestoreReactiveRepository<User> {
    Flux<User> findByAge(Integer age);

    Flux<User> findByAgeGreaterThanAndAgeLessThan(Integer age1, Integer age2);

    Flux<User> findByAgeGreaterThan(Integer age);

    Flux<User> findByAgeGreaterThan(Integer age, Pageable pageable);

    Flux<User> findByAgeIn(List<Integer> ages);

    Flux<User> findByAgeAndPetsContains(Integer age, List<String> pets);

    Flux<User> findByPetsContains(List<String> pets);

    Flux<User> findByPetsContainsAndAgeIn(String pets, List<Integer> ages);

    Mono<Long> countByAgeIsGreaterThan(Integer age);
}
```

Spring Data generates a working implementation of the specified interface, which can be autowired into an application.

The `User` type parameter to `FirestoreReactiveRepository` refers to the underlying domain type.

```

public class MyApplication {

    @Autowired
    UserRepository userRepository;

    public void writeReadDeleteTest() {
        List<User.Address> addresses = Arrays.asList(new User.Address("123 Alice st",
"US"),
            new User.Address("1 Alice ave", "US"));
        User.Address homeAddress = new User.Address("10 Alice blvd", "UK");
        User alice = new User("Alice", 29, null, addresses, homeAddress);
        User bob = new User("Bob", 60);

        this.userRepository.save(alice).block();
        this.userRepository.save(bob).block();

        assertThat(this.userRepository.count().block()).isEqualTo(2);

        assertThat(this.userRepository.findAll().map(User::getName).collectList().block())
            .containsExactlyInAnyOrder("Alice", "Bob");

        User aliceLoaded = this.userRepository.findById("Alice").block();
        assertThat(aliceLoaded.getAddresses()).isEqualTo(addresses);
        assertThat(aliceLoaded.getHomeAddress()).isEqualTo(homeAddress);
    }
}

```

Repositories allow you to define custom Query Methods (detailed in the following sections) for retrieving and counting based on filtering and paging parameters.



Custom queries with `@Query` annotation are not supported since there is no query language in Cloud Firestore

19.15.4. Query methods by convention

```

public class MyApplication {
    public void partTreeRepositoryMethodTest() {
        User u1 = new User("Cloud", 22);
        User u2 = new User("Squall", 17);
        Flux<User> users = Flux.fromArray(new User[] {u1, u2});

        this.userRepository.saveAll(users).blockLast();

        assertThat(this.userRepository.count().block()).isEqualTo(2);

        assertThat(this.userRepository.findByAge(22).collectList().block()).containsExactly(u1
);
            assertThat(this.userRepository.findByAgeGreaterThanAndAgeLessThan(20,
30).collectList().block())
                .containsExactly(u1);

        assertThat(this.userRepository.findByAgeGreaterThan(10).collectList().block()).contains
ExactlyInAnyOrder(u1,
                u2);
    }
}

```

In the example above the query method implementations in `UserRepository` are generated based on the name of the methods using the [Spring Data Query creation naming convention](#).

Cloud Firestore only supports filter components joined by AND, and the following operations:

- `equals`
- `greater than or equals`
- `greater than`
- `less than or equals`
- `less than`
- `is null`
- `contains` (accepts a `List` with up to 10 elements, or a singular value)
- `in` (accepts a `List` with up to 10 elements)



If `in` operation is used in combination with `contains` operation, the argument to `contains` operation has to be a singular value.

After writing a custom repository interface specifying just the signatures of these methods, implementations are generated for you and can be used with an auto-wired instance of the repository.

19.15.5. Transactions

Read-only and read-write transactions are provided by `TransactionalOperator` (see this [blog post](#) on reactive transactions for details). In order to use it, you would need to autowire

`ReactiveFirestoreTransactionManager` like this:

```
public class MyApplication {
    @Autowired
    ReactiveFirestoreTransactionManager txManager;
}
```

After that you will be able to use it to create an instance of `TransactionalOperator`. Note that you can switch between read-only and read-write transactions using `TransactionDefinition` object:

```
DefaultTransactionDefinition transactionDefinition = new
DefaultTransactionDefinition();
transactionDefinition.setReadOnly(false);
TransactionalOperator operator = TransactionalOperator.create(this.txManager,
transactionDefinition);
```

When you have an instance of `TransactionalOperator`, you can execute a sequence of Firestore operations in a transaction using `operator::transactional`:

```
User alice = new User("Alice", 29);
User bob = new User("Bob", 60);

this.userRepository.save(alice)
    .then(this.userRepository.save(bob))
    .as(operator::transactional)
    .block();

this.userRepository.findAll()
    .flatMap(a -> {
        a.setAge(a.getAge() - 1);
        return this.userRepository.save(a);
    })
    .as(operator::transactional).collectList().block();

assertThat(this.userRepository.findAll().map(User::getAge).collectList().block())
    .containsExactlyInAnyOrder(28, 59);
```



Read operations in a transaction can only happen before write operations. All write operations are applied atomically. Read documents are locked until the transaction finishes with a commit or a rollback, which are handled by Spring Data. If an `Exception` is thrown within a transaction, the rollback operation is executed. Otherwise, the commit operation is executed.

Declarative Transactions with `@Transactional` Annotation

This feature requires a bean of `SpannerTransactionManager`, which is provided when using `spring-`

cloud-gcp-starter-data-firestore.

`FirestoreTemplate` and `FirestoreReactiveRepository` support running methods with the `@Transactional` annotation as transactions. If a method annotated with `@Transactional` calls another method also annotated, then both methods will work within the same transaction.

One way to use this feature is illustrated here. You would need to do the following:

1. Annotate your configuration class with the `@EnableTransactionManagement` annotation.
2. Create a service class that has methods annotated with `@Transactional`:

```
class UserService {
    @Autowired
    private UserRepository userRepository;

    @Transactional
    public Mono<Void> updateUsers() {
        return this.userRepository.findAll()
            .flatMap(a -> {
                a.setAge(a.getAge() - 1);
                return this.userRepository.save(a);
            })
            .then();
    }
}
```

3. Make a Spring Bean provider that creates an instance of that class:

```
@Bean
public UserService userService() {
    return new UserService();
}
```

After that, you can autowire your service like so:

```
public class MyApplication {
    @Autowired
    UserService userService;
}
```

Now when you call the methods annotated with `@Transactional` on your service object, a transaction will be automatically started. If an error occurs during the execution of a method annotated with `@Transactional`, the transaction will be rolled back. If no error occurs, the transaction will be committed.

19.15.6. Reactive Repository Sample

A [sample application](#) is available.

19.15.7. Cloud Firestore Spring Boot Starter

If you prefer using Firestore client only, Spring Cloud GCP provides a convenience starter which automatically configures authentication settings and client objects needed to begin using [Google Cloud Firestore](#) in native mode.

See [documentation](#) to learn more about Cloud Firestore.

To begin using this library, add the `spring-cloud-gcp-starter-firestore` artifact to your project.

Maven coordinates, using [Spring Cloud GCP BOM](#):

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-gcp-starter-firestore</artifactId>
</dependency>
```

Gradle coordinates:

```
dependencies {
  compile group: 'org.springframework.cloud', name: 'spring-cloud-gcp-starter-
  firestore'
}
```

Using Cloud Firestore

The starter automatically configures and registers a `Firestore` bean in the Spring application context. To start using it, simply use the `@Autowired` annotation.

```

@Autowired
Firestore firestore;

void writeDocumentFromObject() throws ExecutionException, InterruptedException {
    // Add document data with id "joe" using a custom User class
    User data = new User("Joe",
        Arrays.asList(
            new Phone(12345, PhoneType.CELL),
            new Phone(54321, PhoneType.WORK)));

    // .get() blocks on response
    WriteResult writeResult = this.firestore.document("users/joe").set(data).get();

    LOGGER.info("Update time: " + writeResult.getUpdateTime());
}

User readDocumentToObject() throws ExecutionException, InterruptedException {
    ApiFuture<DocumentSnapshot> documentFuture =
        this.firestore.document("users/joe").get();

    User user = documentFuture.get().toObject(User.class);

    LOGGER.info("read: " + user);

    return user;
}

```

Sample

A [sample application](#) is available.

19.16. Cloud Memorystore for Redis

19.16.1. Spring Caching

[Cloud Memorystore for Redis](#) provides a fully managed in-memory data store service. Cloud Memorystore is compatible with the Redis protocol, allowing easy integration with [Spring Caching](#).

All you have to do is create a Cloud Memorystore instance and use its IP address in `application.properties` file as `spring.redis.host` property value. Everything else is exactly the same as setting up redis-backed Spring caching.



Memorystore instances and your application instances have to be located in the same region.

In short, the following dependencies are needed:

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-cache</artifactId>
</dependency>
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-data-redis</artifactId>
</dependency>
```

And then you can use `org.springframework.cache.annotation.Cacheable` annotation for methods you'd like to be cached.

```
@Cacheable("cache1")
public String hello(@PathVariable String name) {
    ....
}
```

If you are interested in a detailed how-to guide, please check [Spring Boot Caching using Cloud Memorystore codelab](#).

Cloud Memorystore documentation can be found [here](#).

19.17. Cloud Identity-Aware Proxy (IAP) Authentication

[Cloud Identity-Aware Proxy \(IAP\)](#) provides a security layer over applications deployed to Google Cloud.

The IAP starter uses [Spring Security OAuth 2.0 Resource Server](#) functionality to automatically extract user identity from the proxy-injected `x-goog-iap-jwt-assertion` HTTP header.

The following claims are validated automatically:

- Issue time
- Expiration time
- Issuer
- Audience

The *audience* ("aud" claim) validation string is automatically determined when the application is running on App Engine Standard or App Engine Flexible. This functionality relies on Cloud Resource Manager API to retrieve project details, so the following setup is needed:

- Enable Cloud Resource Manager API in [GCP Console](#).
- Make sure your application has `resourcemanager.projects.get` permission.

App Engine automatic *audience* determination can be overridden by using

`spring.cloud.gcp.security.iap.audience` property. It supports multiple allowable audiences by providing a comma-delimited list.

For Compute Engine or Kubernetes Engine `spring.cloud.gcp.security.iap.audience` property **must** be provided, as the *audience* string depends on the specific Backend Services setup and cannot be inferred automatically. To determine the *audience* value, follow directions in IAP [Verify the JWT payload](#) guide. If `spring.cloud.gcp.security.iap.audience` is not provided, the application will fail to start the following message:

```
No qualifying bean of type
'org.springframework.cloud.gcp.security.iap.AudienceProvider' available.
```



If you create a custom `WebSecurityConfigurerAdapter`, enable extracting user identity by adding `.oauth2ResourceServer().jwt()` configuration to the `HttpSecurity` object. If no custom `WebSecurityConfigurerAdapter` is present, nothing needs to be done because Spring Boot will add this customization by default.

Starter Maven coordinates, using [Spring Cloud GCP BOM](#):

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-gcp-starter-security-iap</artifactId>
</dependency>
```

Starter Gradle coordinates:

```
dependencies {
  compile group: 'org.springframework.cloud', name: 'spring-cloud-gcp-starter-
security-iap'
}
```

19.17.1. Configuration

The following properties are available.



Modifying registry, algorithm, and header properties might be useful for testing, but the defaults should not be changed in production.

Name	Description	Required	Default
<code>spring.cloud.gcp.security.iap.registry</code>	Link to JWK public key registry.	true	www.gstatic.com/iap/verify/public_key-jwk
<code>spring.cloud.gcp.security.iap.algorithm</code>	Encryption algorithm used to sign the JWK token.	true	ES256

Name	Description	Required	Default
<code>spring.cloud.gcp.security.iap.header</code>	Header from which to extract the JWK key.	true	<code>x-goog-iap-jwt-assertion</code>
<code>spring.cloud.gcp.security.iap.issuer</code>	JWK issuer to verify.	true	<code>cloud.google.com/iap</code>
<code>spring.cloud.gcp.security.iap.audience</code>	Custom JWK audience to verify.	false on App Engine; true on GCE/GKE	

19.17.2. Sample

A [sample application](#) is available.

19.18. Google Cloud Vision

The [Google Cloud Vision API](#) allows users to leverage machine learning algorithms for processing images and documents including: image classification, face detection, text extraction, optical character recognition, and others.

Spring Cloud GCP provides:

- A convenience starter which automatically configures authentication settings and client objects needed to begin using the [Google Cloud Vision API](#).
- `CloudVisionTemplate` which simplifies interactions with the Cloud Vision API.
 - Allows you to easily send images to the API as Spring Resources.
 - Offers convenience methods for common operations, such as classifying content of an image.
- `DocumentOcrTemplate` which offers convenient methods for running [optical character recognition \(OCR\)](#) on PDF and TIFF documents.

19.18.1. Dependency Setup

To begin using this library, add the `spring-cloud-gcp-starter-vision` artifact to your project.

Maven coordinates, using [Spring Cloud GCP BOM](#):

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-gcp-starter-vision</artifactId>
</dependency>
```

Gradle coordinates:

```
dependencies {
    compile group: 'org.springframework.cloud', name: 'spring-cloud-gcp-starter-vision'
}
```

Cloud Vision OCR Dependencies

If you are interested in applying optical character recognition (OCR) on documents for your project, you'll need to add both `spring-cloud-gcp-starter-vision` and `spring-cloud-gcp-starter-storage` to your dependencies. The storage starter is necessary because the Cloud Vision API will process your documents and write OCR output files all within your Google Cloud Storage buckets.

Maven coordinates using [Spring Cloud GCP BOM](#):

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-gcp-starter-vision</artifactId>
</dependency>
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-gcp-starter-storage</artifactId>
</dependency>
```

Gradle coordinates:

```
dependencies {
    compile group: 'org.springframework.cloud', name: 'spring-cloud-gcp-starter-vision'
    compile group: 'org.springframework.cloud', name: 'spring-cloud-gcp-starter-storage'
}
```

19.18.2. Image Analysis

The `CloudVisionTemplate` allows you to easily analyze images; it provides the following method for interfacing with Cloud Vision:

```
public AnnotateImageResponse analyzeImage(Resource imageResource, Feature.Type... featureTypes)
```

Parameters:

- `Resource imageResource` refers to the Spring Resource of the image object you wish to analyze. The Google Cloud Vision documentation provides a [list of the image types that they support](#).
- `Feature.Type... featureTypes` refers to a var-arg array of Cloud Vision Features to extract from the image. A feature refers to a kind of image analysis one wishes to perform on an image, such as label detection, OCR recognition, facial detection, etc. One may specify multiple features to analyze within one request. A full list of Cloud Vision Features is provided in the [Cloud Vision Feature docs](#).

Returns:

- `AnnotateImageResponse` contains the results of all the feature analyses that were specified in the request. For each feature type that you provide in the request, `AnnotateImageResponse` provides a getter method to get the result of that feature analysis. For example, if you analyzed an image using the `LABEL_DETECTION` feature, you would retrieve the results from the response using `annotateImageResponse.getLabelAnnotationsList()`.

`AnnotateImageResponse` is provided by the Google Cloud Vision libraries; please consult the [RPC reference](#) or [Javadoc](#) for more details. Additionally, you may consult the [Cloud Vision docs](#) to familiarize yourself with the concepts and features of the API.

Detect Image Labels Example

[Image labeling](#) refers to producing labels that describe the contents of an image. Below is a code sample of how this is done using the Cloud Vision Spring Template.

```
@Autowired
private ResourceLoader resourceLoader;

@Autowired
private CloudVisionTemplate cloudVisionTemplate;

public void processImage() {
    Resource imageResource = this.resourceLoader.getResource("my_image.jpg");
    AnnotateImageResponse response = this.cloudVisionTemplate.analyzeImage(
        imageResource, Type.LABEL_DETECTION);
    System.out.println("Image Classification results: " +
        response.getLabelAnnotationsList());
}
```

19.18.3. Document OCR Template

The `DocumentOcrTemplate` allows you to easily run [optical character recognition \(OCR\)](#) on your PDF and TIFF documents stored in your Google Storage bucket.

First, you will need to create a bucket in [Google Cloud Storage](#) and [upload the documents you wish to process into the bucket](#).

Running OCR on a Document

When OCR is run on a document, the Cloud Vision APIs will output a collection of OCR output files in JSON which describe the text content, bounding rectangles of words and letters, and other information about the document.

The `DocumentOcrTemplate` provides the following method for running OCR on a document saved in Google Cloud Storage:

```
ListenableFuture<DocumentOcrResultSet> runOcrForDocument(GoogleStorageLocation document,
    GoogleStorageLocation outputPathPrefix)
```

The method allows you to specify the location of the document and the output location for where all the JSON output files will be saved in Google Cloud Storage. It returns a `ListenableFuture` containing `DocumentOcrResultSet` which contains the OCR content of the document.



Running OCR on a document is an operation that can take between several minutes to several hours depending on how large the document is. It is recommended to register callbacks to the returned `ListenableFuture` or ignore it and process the JSON output files at a later point in time using `readOcrOutputFile` or `readOcrOutputFileSet`.

Running OCR Example

Below is a code snippet of how to run OCR on a document stored in a Google Storage bucket and read the text in the first page of the document.

```
@Autowired
private DocumentOcrTemplate documentOcrTemplate;

public void runOcrOnDocument() {
    GoogleStorageLocation document = GoogleStorageLocation.forFile(
        "your-bucket", "test.pdf");
    GoogleStorageLocation outputLocationPrefix = GoogleStorageLocation.forFolder(
        "your-bucket", "output_folder/test.pdf/");

    ListenableFuture<DocumentOcrResultSet> result =
        this.documentOcrTemplate.runOcrForDocument(
            document, outputLocationPrefix);

    DocumentOcrResultSet ocrPages = result.get(5, TimeUnit.MINUTES);

    String page1Text = ocrPages.getPage(1).getText();
    System.out.println(page1Text);
}
```

Reading OCR Output Files

In some use-cases, you may need to directly read OCR output files stored in Google Cloud Storage.

`DocumentOcrTemplate` offers the following methods for reading and processing OCR output files:

- `readOcrOutputFileSet(GoogleStorageLocation jsonOutputFilePathPrefix)`: Reads a collection of OCR output files under a file path prefix and returns the parsed contents. All of the files under the path should correspond to the same document.
- `readOcrOutputFile(GoogleStorageLocation jsonFile)`: Reads a single OCR output file and returns the parsed contents.

Reading OCR Output Files Example

The code snippet below describes how to read the OCR output files of a single document.

```
@Autowired
private DocumentOcrTemplate documentOcrTemplate;

// Parses the OCR output files corresponding to a single document in a directory
public void parseOutputFileSet() {
    GoogleStorageLocation ocrOutputPrefix = GoogleStorageLocation.forFolder(
        "your-bucket", "json_output_set/");

    DocumentOcrResultSet result =
this.documentOcrTemplate.readOcrOutputFileSet(ocrOutputPrefix);
    System.out.println("Page 2 text: " + result.getPage(2).getText());
}

// Parses a single OCR output file
public void parseSingleOutputFile() {
    GoogleStorageLocation ocrOutputFile = GoogleStorageLocation.forFile(
        "your-bucket", "json_output_set/test_output-2-to-2.json");

    DocumentOcrResultSet result =
this.documentOcrTemplate.readOcrOutputFile(ocrOutputFile);
    System.out.println("Page 2 text: " + result.getPage(2).getText());
}
```

19.18.4. Configuration

The following options may be configured with Spring Cloud GCP Vision libraries.

Name	Description	Required	Default value
<code>spring.cloud.gcp.vision.enabled</code>	Enables or disables Cloud Vision autoconfiguration	No	<code>true</code>
<code>spring.cloud.gcp.vision.executors-threads-count</code>	Number of threads used during document OCR processing for waiting on long-running OCR operations	No	1
<code>spring.cloud.gcp.vision.json-output-batch-size</code>	Number of document pages to include in each OCR output file.	No	20

19.18.5. Sample

Samples are provided to show example usages of Spring Cloud GCP with Google Cloud Vision.

- The [Image Labeling Sample](#) shows you how to use image labelling in your Spring application. The application generates labels describing the content inside the images you specify in the application.
- The [Document OCR demo](#) shows how you can apply OCR processing on your PDF/TIFF documents in order to extract their text contents.

19.19. Google Cloud BigQuery

[Google Cloud BigQuery](#) is a fully managed, petabyte scale, low cost analytics data warehouse.

Spring Cloud GCP provides:

- A convenience starter which provides autoconfiguration for the [BigQuery](#) client objects with credentials needed to interface with BigQuery.
- A Spring Integration message handler for loading data into BigQuery tables in your Spring integration pipelines.

Maven coordinates, using [Spring Cloud GCP BOM](#):

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-gcp-bigquery-starter</artifactId>
</dependency>
```

Gradle coordinates:

```
dependencies {
    compile group: 'org.springframework.cloud', name: 'spring-cloud-gcp-bigquery-
starter'
}
```

19.19.1. BigQuery Autoconfiguration

Adding the [spring-cloud-gcp-bigquery-starter](#) dependency to the POM will enable autoconfiguration of the client objects needed to interface with BigQuery.

BigQuery Client Object

The [GcpBigQueryAutoConfiguration](#) class configures an instance of [BigQuery](#) for you by inferring your credentials and Project ID from the machine's environment.

Example usage:

```

// BigQuery client object provided by our autoconfiguration.
@Autowired
BigQuery bigquery;

public void runQuery() throws InterruptedException {
    String query = "SELECT column FROM table;";
    QueryJobConfiguration queryConfig =
        QueryJobConfiguration.newBuilder(query).build();

    // Run the query using the BigQuery object
    for (FieldValueList row : bigquery.query(queryConfig).iterateAll()) {
        for (FieldValue val : row) {
            System.out.println(val);
        }
    }
}

```

This object is used to interface with all BigQuery services. For more information, see the [BigQuery Client Library usage examples](#).

BigQueryTemplate

The `BigQueryTemplate` class is a wrapper over the `BigQuery` client object and makes it easier to load data into BigQuery tables. A `BigQueryTemplate` is scoped to a single dataset. The autoconfigured `BigQueryTemplate` instance will use the dataset provided through the property `spring.cloud.gcp.bigquery.datasetName`.

Below is a code snippet of how to load a CSV data `InputStream` to a BigQuery table.

```

// BigQuery client object provided by our autoconfiguration.
@Autowired
BigQueryTemplate bigQueryTemplate;

public void loadData(InputStream dataInputStream, String tableName) {
    ListenableFuture<Job> bigQueryJobFuture =
        bigQueryTemplate.writeDataToTable(
            tableName,
            dataFile.getInputStream(),
            FormatOptions.csv());

    // After the future is complete, the data is successfully loaded.
    Job job = bigQueryJobFuture.get();
}

```

19.19.2. Spring Integration

Spring Cloud GCP BigQuery also provides a Spring Integration message handler `BigQueryFileMessageHandler`. This is useful for incorporating BigQuery data loading operations in a

Spring Integration pipeline.

Below is an example configuring a `ServiceActivator` bean using the `BigQueryFileMessageHandler`.

```
@Bean
public DirectChannel bigQueryWriteDataChannel() {
    return new DirectChannel();
}

@Bean
public DirectChannel bigQueryJobReplyChannel() {
    return new DirectChannel();
}

@Bean
@ServiceActivator(inputChannel = "bigQueryWriteDataChannel")
public MessageHandler messageSender(BigQueryTemplate bigQueryTemplate) {
    BigQueryFileMessageHandler messageHandler = new
    BigQueryFileMessageHandler(bigQueryTemplate);
    messageHandler.setFormatOptions(FormatOptions.csv());
    messageHandler.setOutputChannel(bigQueryJobReplyChannel());
    return messageHandler;
}
```

BigQuery Message Handling

The `BigQueryFileMessageHandler` accepts the following message payload types for loading into BigQuery: `java.io.File`, `byte[]`, `org.springframework.core.io.Resource`, and `java.io.InputStream`. The message payload will be streamed and written to the BigQuery table you specify.

By default, the `BigQueryFileMessageHandler` is configured to read the headers of the messages it receives to determine how to load the data. The headers are specified by the class `BigQuerySpringMessageHeaders` and summarized below.

Header	Description
<code>BigQuerySpringMessageHeaders.TABLE_NAME</code>	Specifies the BigQuery table within your dataset to write to.
<code>BigQuerySpringMessageHeaders.FORMAT_OPTIONS</code>	Describes the data format of your data to load (i.e. CSV, JSON, etc.).

Alternatively, you may omit these headers and explicitly set the table name or format options by calling `setTableName(...)` and `setFormatOptions(...)`.

BigQuery Message Reply

After the `BigQueryFileMessageHandler` processes a message to load data to your BigQuery table, it will respond with a `Job` on the reply channel. The `Job` object provides metadata and information about the load file operation.

By default, the `BigQueryFileMessageHandler` is run in asynchronous mode, with `setSync(false)`, and it will reply with a `ListenableFuture<Job>` on the reply channel. The future is tied to the status of the data loading job and will complete when the job completes.

If the handler is run in synchronous mode with `setSync(true)`, then the handler will block on the completion of the loading job and block until it is complete.



If you decide to use Spring Integration Gateways and you wish to receive `ListenableFuture<Job>` as a reply object in the Gateway, you will have to call `.setAsyncExecutor(null)` on your `GatewayProxyFactoryBean`. This is needed to indicate that you wish to reply on the built-in async support rather than rely on async handling of the gateway.

19.19.3. Configuration

The following application properties may be configured with Spring Cloud GCP BigQuery libraries.

Name	Description	Required	Default value
<code>spring.cloud.gcp.bigquery.datasetName</code>	The BigQuery dataset that the <code>BigQueryTemplate</code> and <code>BigQueryFileMessageHandler</code> is scoped to.	Yes	
<code>spring.cloud.gcp.bigquery.enabled</code>	Enables or disables Spring Cloud GCP BigQuery autoconfiguration.	No	<code>true</code>
<code>spring.cloud.gcp.bigquery.project-id</code>	GCP project ID of the project using BigQuery APIs, if different from the one in the Spring Cloud GCP Core Module .	No	Project ID is typically inferred from <code>gcloud</code> configuration.
<code>spring.cloud.gcp.bigquery.credentials.location</code>	Credentials file location for authenticating with the Google Cloud BigQuery APIs, if different from the ones in the Spring Cloud GCP Core Module	No	Inferred from Application Default Credentials , typically set by <code>gcloud</code> .

19.20. Secret Manager

[Google Cloud Secret Manager](#) is a secure and convenient method for storing API keys, passwords, certificates, and other sensitive data. A detailed summary of its features can be found in the [Secret Manager documentation](#).

Spring Cloud GCP provides:

- A Spring Boot starter which automatically loads the secrets of your GCP project into your application context as a [Bootstrap Property Source](#).
- A [SecretManagerTemplate](#) which allows you to read, write, and update secrets in Secret Manager.

19.20.1. Dependency Setup

To begin using this library, add the `spring-cloud-gcp-starter-secretmanager` artifact to your project.

Maven coordinates, using [Spring Cloud GCP BOM](#):

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-gcp-starter-secretmanager</artifactId>
</dependency>
```

Gradle coordinates:

```
dependencies {
  compile group: 'org.springframework.cloud', name: 'spring-cloud-gcp-starter-
secretmanager'
}
```

19.20.2. Secret Manager Property Source

The Spring Cloud GCP integration for Google Cloud Secret Manager enables you to use Secret Manager as a bootstrap property source.

This feature allows you to automatically load your GCP project's secrets as properties into the application context during the [Bootstrap Phase](#), which refers to the initial phase when a Spring application is being loaded.



This feature disabled by default; to use it, you must set `spring.cloud.gcp.secretmanager.bootstrap.enabled` to `true` in your `bootstrap.properties` file.

Spring Cloud GCP will load the **latest** version of each secret into the application context.

All secrets will be loaded into the application environment using their `secretId` as the property name. For example, if your secret's id is `my-secret` then it will be accessible as a property using the name `my-secret`. If you would like to append a prefix string to all property names imported from Secret Manager, you may use the `spring.cloud.gcp.secretmanager.secret-name-prefix` setting described below.

Spring Cloud GCP Secret Manager offers several bootstrap configuration properties to customize the behavior.



All of the below settings must be specified in a `bootstrap.properties` (or `bootstrap.yaml`) file which is the properties file used to configure settings for bootstrap-phase Spring configuration.

Name	Description	Required	Default value
<code>spring.cloud.gcp.secretmanager.bootstrap.enabled</code>	Enables loading secrets from Secret Manager as a bootstrap property source. Set this to true to enable the feature.	No	<code>false</code>
<code>spring.cloud.gcp.secretmanager.secret-name-prefix</code>	A prefix string to prepend to the property names of secrets read from Secret Manager	No	"" (empty string)
<code>spring.cloud.gcp.secretmanager.versions.<secret-id></code>	Defines a version for a specific secret-id to read from Secret Manager instead of using the latest version.	No	"" (empty string)

See the Authentication Settings section below for information on how to set properties to authenticate to Secret Manager.

19.20.3. Secret Manager Template

The `SecretManagerTemplate` class simplifies operations of creating, updating, and reading secrets.

To begin using this class, you may inject an instance of the class using `@Autowired` after adding the starter dependency to your project.

```
@Autowired
private SecretManagerTemplate secretManagerTemplate;
```

Please consult `SecretManagerOperations` for information on what operations are available for the Secret Manager template.

Name	Description	Required	Default value
<code>spring.cloud.gcp.secretmanager.enabled</code>	Enables the autowiring of the <code>SecretManagerTemplate</code> in the application context.	No	<code>true</code>

See the Authentication Settings section below for information on how to set properties to

authenticate to Secret Manager.

19.20.4. Authentication Settings

By default, Spring Cloud GCP Secret Manager will authenticate using Application Default Credentials. This can be overridden using the authentication properties below.

Name	Description	Required	Default value
<code>spring.cloud.gcp.secretmanager.credentials.location</code>	OAuth2 credentials for authenticating to the Google Cloud Secret Manager API.	No	By default, infers credentials from Application Default Credentials .
<code>spring.cloud.gcp.secretmanager.credentials.encoded-key</code>	Base64-encoded contents of OAuth2 account private key for authenticating to the Google Cloud Secret Manager API.	No	By default, infers credentials from Application Default Credentials .
<code>spring.cloud.gcp.secretmanager.project-id</code>	The GCP Project used to access Secret Manager API.	No	By default, infers the project from Application Default Credentials .

19.20.5. Sample Application

A [Secret Manager Sample Application](#) is provided which demonstrates basic property source loading and usage of the template class.

19.21. Cloud Foundry

Spring Cloud GCP provides support for Cloud Foundry's [GCP Service Broker](#). Our Pub/Sub, Cloud Spanner, Storage, Stackdriver Trace and Cloud SQL MySQL and PostgreSQL starters are Cloud Foundry aware and retrieve properties like project ID, credentials, etc., that are used in auto configuration from the Cloud Foundry environment.

In order to take advantage of the Cloud Foundry support make sure the following dependency is added:

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-gcp-starter-cloudfoundry</artifactId>
</dependency>
```

In cases like Pub/Sub's topic and subscription, or Storage's bucket name, where those parameters are not used in auto configuration, you can fetch them using the VCAP mapping provided by Spring Boot. For example, to retrieve the provisioned Pub/Sub topic, you can use the

`vcap.services.mypubsub.credentials.topic_name` property from the application environment.



If the same service is bound to the same application more than once, the auto configuration will not be able to choose among bindings and will not be activated for that service. This includes both MySQL and PostgreSQL bindings to the same app.



In order for the Cloud SQL integration to work in Cloud Foundry, auto-reconfiguration must be disabled. You can do so using the `cf set-env <APP> JBP_CONFIG_SPRING_AUTO_RECONFIGURATION '{enabled: false}'` command. Otherwise, Cloud Foundry will produce a `DataSource` with an invalid JDBC URL (i.e., `jdbc:mysql://null/null`).

19.21.1. User-Provided Services

[User-provided services](#) enable developers to use services that are not available in the marketplace with their apps running on Cloud Foundry. For example, you may want to use a user-provided service that points to a shared Google Service (like Cloud Spanner) used across your organization.

In order for Spring Cloud GCP to detect your user-provided service as a Google Cloud Service, you must add an [instance tag](#) indicating the Google Cloud Service it uses. The tag should simply be the Cloud Foundry name for the Google Service.

For example, if you create a user-provided service using Cloud Spanner, you might run:

```
$ cf create-user-provided-service user-spanner-service -t "google-spanner" ...
```

This allows Spring Cloud GCP to retrieve the correct service properties from Cloud Foundry and use them in the auto configuration for your application.

A mapping of Google service names to Cloud Foundry names are provided below:

Google Cloud Service	Cloud Foundry Name (add this as a tag)
Google Cloud Pub/Sub	<code>google-pubsub</code>
Google Cloud Storage	<code>google-storage</code>
Google Cloud Spanner	<code>google-spanner</code>
Datastore	<code>google-datastore</code>
Firestore	<code>google-firestore</code>
BigQuery	<code>google-bigquery</code>
Stackdriver Trace	<code>google-stackdriver-trace</code>
Cloud Sql (MySQL)	<code>google-cloudsql-mysql</code>
Cloud Sql (PostgreSQL)	<code>google-cloudsql-postgres</code>

19.22. Kotlin Support

The latest version of the Spring Framework provides first-class support for Kotlin. For Kotlin users of Spring, the Spring Cloud GCP libraries work out-of-the-box and are fully interoperable with Kotlin applications.

For more information on building a Spring application in Kotlin, please consult the [Spring Kotlin documentation](#).

19.22.1. Prerequisites

Ensure that your Kotlin application is properly set up. Based on your build system, you will need to include the correct Kotlin build plugin in your project:

- [Kotlin Maven Plugin](#)
- [Kotlin Gradle Plugin](#)

Depending on your application's needs, you may need to augment your build configuration with compiler plugins:

- [Kotlin Spring Plugin](#): Makes your Spring configuration classes/members non-final for convenience.
- [Kotlin JPA Plugin](#): Enables using JPA in Kotlin applications.

Once your Kotlin project is properly configured, the Spring Cloud GCP libraries will work within your application without any additional setup.

19.22.2. Sample

A [Kotlin sample application](#) is provided to demonstrate a working Maven setup and various Spring Cloud GCP integrations from within Kotlin.

19.23. Configuration properties

To see the list of all GCP related configuration properties please check [the Appendix page](#).

Chapter 20. Spring Cloud Circuit Breaker

2020.0.0-M2

20.1. Configuring Resilience4J Circuit Breakers

Starters

There are two starters for the Resilience4J implementations, one for reactive applications and one for non-reactive applications.

- `org.springframework.cloud:spring-cloud-starter-circuitbreaker-resilience4j` - non-reactive applications
- `org.springframework.cloud:spring-cloud-starter-circuitbreaker-reactor-resilience4j` - reactive applications

Auto-Configuration

You can disable the Resilience4J auto-configuration by setting `spring.cloud.circuitbreaker.resilience4j.enabled` to `false`.

Default Configuration

To provide a default configuration for all of your circuit breakers create a `Customize` bean that is passed a `Resilience4JCircuitBreakerFactory` or `ReactiveResilience4JCircuitBreakerFactory`. The `configureDefault` method can be used to provide a default configuration.

```
@Bean
public Customizer<Resilience4JCircuitBreakerFactory> defaultCustomizer() {
    return factory -> factory.configureDefault(id -> new
        Resilience4JConfigBuilder(id)

            .timeLimiterConfig(TimeLimiterConfig.custom().timeoutDuration(Duration.ofSeconds(4))
                .build())

            .circuitBreakerConfig(CircuitBreakerConfig.ofDefaults())
                .build());
}
```

Reactive Example

```

@Bean
public Customizer<ReactiveResilience4JCircuitBreakerFactory> defaultCustomizer() {
    return factory -> factory.configureDefault(id -> new
        Resilience4JConfigBuilder(id)
            .circuitBreakerConfig(CircuitBreakerConfig.ofDefaults())

            .timeLimiterConfig(TimeLimiterConfig.custom().timeoutDuration(Duration.ofSeconds(4
                )).build()).build());
}

```

Specific Circuit Breaker Configuration

Similarly to providing a default configuration, you can create a `Customize` bean this is passed a `Resilience4JCircuitBreakerFactory` or `ReactiveResilience4JCircuitBreakerFactory`.

```

@Bean
public Customizer<Resilience4JCircuitBreakerFactory> slowCustomizer() {
    return factory -> factory.configure(builder ->
        builder.circuitBreakerConfig(CircuitBreakerConfig.ofDefaults())

            .timeLimiterConfig(TimeLimiterConfig.custom().timeoutDuration(Duration.ofSeconds(2
                )).build()), "slow");
}

```

In addition to configuring the circuit breaker that is created you can also customize the circuit breaker after it has been created but before it is returned to the caller. To do this you can use the `addCircuitBreakerCustomizer` method. This can be useful for adding event handlers to Resilience4J circuit breakers.

```

@Bean
public Customizer<Resilience4JCircuitBreakerFactory> slowCustomizer() {
    return factory -> factory.addCircuitBreakerCustomizer(circuitBreaker ->
        circuitBreaker.getEventPublisher()
            .onError(normalFluxErrorConsumer).onSuccess(normalFluxSuccessConsumer),
        "normalflux");
}

```

Reactive Example

```

@Bean
public Customizer<ReactiveResilience4JCircuitBreakerFactory> slowCustomizer() {
    return factory -> {
        factory.configure(builder -> builder

            .timeLimiterConfig(TimeLimiterConfig.custom().timeoutDuration(Duration.ofSeconds(2))
                .build())
            .circuitBreakerConfig(CircuitBreakerConfig.ofDefaults()), "slow",
            "slowflux");
        factory.addCircuitBreakerCustomizer(circuitBreaker ->
            circuitBreaker.getEventPublisher()

                .onError(normalFluxErrorConsumer).onSuccess(normalFluxSuccessConsumer),
                "normalflux");
    };
}

```

Collecting Metrics

Spring Cloud Circuit Breaker Resilience4j includes auto-configuration to setup metrics collection as long as the right dependencies are on the classpath. To enable metric collection you must include `org.springframework.boot:spring-boot-starter-actuator`, and `io.github.resilience4j:resilience4j-micrometer`. For more information on the metrics that get produced when these dependencies are present, see the [Resilience4j documentation](#).



You don't have to include `micrometer-core` directly as it is brought in by `spring-boot-starter-actuator`

20.2. Configuring Spring Retry Circuit Breakers

Spring Retry provides declarative retry support for Spring applications. A subset of the project includes the ability to implement circuit breaker functionality. Spring Retry provides a circuit breaker implementation via a combination of its `CircuitBreakerRetryPolicy` and a `stateful retry`. All circuit breakers created using Spring Retry will be created using the `CircuitBreakerRetryPolicy` and a `DefaultRetryState`. Both of these classes can be configured using `SpringRetryConfigBuilder`.

Default Configuration

To provide a default configuration for all of your circuit breakers create a `Customize` bean that is passed a `SpringRetryCircuitBreakerFactory`. The `configureDefault` method can be used to provide a default configuration.

```
@Bean
public Customizer<SpringRetryCircuitBreakerFactory> defaultCustomizer() {
    return factory -> factory.configureDefault(id -> new
        SpringRetryConfigBuilder(id)
            .retryPolicy(new TimeoutRetryPolicy()).build());
}
```

Specific Circuit Breaker Configuration

Similarly to providing a default configuration, you can create a `Customize` bean this is passed a `SpringRetryCircuitBreakerFactory`.

```
@Bean
public Customizer<SpringRetryCircuitBreakerFactory> slowCustomizer() {
    return factory -> factory.configure(builder -> builder.retryPolicy(new
        SimpleRetryPolicy(1)).build(), "slow");
}
```

In addition to configuring the circuit breaker that is created you can also customize the circuit breaker after it has been created but before it is returned to the caller. To do this you can use the `addRetryTemplateCustomizers` method. This can be useful for adding event handlers to the `RetryTemplate`.

```

@Bean
public Customizer<SpringRetryCircuitBreakerFactory> slowCustomizer() {
    return factory -> factory.addRetryTemplateCustomizers(retryTemplate ->
retryTemplate.registerListener(new RetryListener() {

        @Override
        public <T, E extends Throwable> boolean open(RetryContext context,
RetryCallback<T, E> callback) {
            return false;
        }

        @Override
        public <T, E extends Throwable> void close(RetryContext context,
RetryCallback<T, E> callback, Throwable throwable) {

        }

        @Override
        public <T, E extends Throwable> void onError(RetryContext context,
RetryCallback<T, E> callback, Throwable throwable) {

        }
    }));
}

```

20.3. Building

20.3.1. Basic Compile and Test

To build the source you will need to install JDK 1.8.

Spring Cloud uses Maven for most build-related activities, and you should be able to get off the ground quite quickly by cloning the project you are interested in and typing

```
$ ./mvnw install
```



You can also install Maven (>=3.3.3) yourself and run the `mvn` command in place of `./mvnw` in the examples below. If you do that you also might need to add `-P spring` if your local Maven settings do not contain repository declarations for spring pre-release artifacts.



Be aware that you might need to increase the amount of memory available to Maven by setting a `MAVEN_OPTS` environment variable with a value like `-Xmx512m -XX:MaxPermSize=128m`. We try to cover this in the `.mvn` configuration, so if you find you have to do it to make a build succeed, please raise a ticket to get the settings added to source control.

For hints on how to build the project look in `.travis.yml` if there is one. There should be a "script" and maybe "install" command. Also look at the "services" section to see if any services need to be running locally (e.g. mongo or rabbit). Ignore the git-related bits that you might find in "before_install" since they're related to setting git credentials and you already have those.

The projects that require middleware generally include a `docker-compose.yml`, so consider using [Docker Compose](#) to run the middleware servers in Docker containers. See the README in the [scripts demo repository](#) for specific instructions about the common cases of mongo, rabbit and redis.



If all else fails, build with the command from `.travis.yml` (usually `./mvnw install`).

20.3.2. Documentation

The spring-cloud-build module has a "docs" profile, and if you switch that on it will try to build asciidoc sources from `src/main/asciidoc`. As part of that process it will look for a `README.adoc` and process it by loading all the includes, but not parsing or rendering it, just copying it to `${main.basedir}` (defaults to `$/home/marcin/repo/spring-cloud-scripts`, i.e. the root of the project). If there are any changes in the README it will then show up after a Maven build as a modified file in the correct place. Just commit it and push the change.

20.3.3. Working with the code

If you don't have an IDE preference we would recommend that you use [Spring Tools Suite](#) or [Eclipse](#) when working with the code. We use the [m2eclipse](#) eclipse plugin for maven support. Other IDEs and tools should also work without issue as long as they use Maven 3.3.3 or better.

Activate the Spring Maven profile

Spring Cloud projects require the 'spring' Maven profile to be activated to resolve the spring milestone and snapshot repositories. Use your preferred IDE to set this profile to be active, or you may experience build errors.

Importing into eclipse with m2eclipse

We recommend the [m2eclipse](#) eclipse plugin when working with eclipse. If you don't already have m2eclipse installed it is available from the "eclipse marketplace".



Older versions of m2e do not support Maven 3.3, so once the projects are imported into Eclipse you will also need to tell m2eclipse to use the right profile for the projects. If you see many different errors related to the POMs in the projects, check that you have an up to date installation. If you can't upgrade m2e, add the "spring" profile to your `settings.xml`. Alternatively you can copy the repository settings from the "spring" profile of the parent pom into your `settings.xml`.

Importing into eclipse without m2eclipse

If you prefer not to use m2eclipse you can generate eclipse project metadata using the following command:

```
$ ./mvnw eclipse:eclipse
```

The generated eclipse projects can be imported by selecting `import existing projects` from the `file` menu.

20.4. Contributing

Spring Cloud is released under the non-restrictive Apache 2.0 license, and follows a very standard Github development process, using Github tracker for issues and merging pull requests into master. If you want to contribute even something trivial please do not hesitate, but follow the guidelines below.

20.4.1. Sign the Contributor License Agreement

Before we accept a non-trivial patch or pull request we will need you to sign the [Contributor License Agreement](#). Signing the contributor's agreement does not grant anyone commit rights to the main repository, but it does mean that we can accept your contributions, and you will get an author credit if we do. Active contributors might be asked to join the core team, and given the ability to merge pull requests.

20.4.2. Code of Conduct

This project adheres to the Contributor Covenant [code of conduct](#). By participating, you are expected to uphold this code. Please report unacceptable behavior to spring-code-of-conduct@pivotal.io.

20.4.3. Code Conventions and Housekeeping

None of these is essential for a pull request, but they will all help. They can also be added after the original pull request but before a merge.

- Use the Spring Framework code format conventions. If you use Eclipse you can import formatter settings using the `eclipse-code-formatter.xml` file from the [Spring Cloud Build](#) project. If using IntelliJ, you can use the [Eclipse Code Formatter Plugin](#) to import the same file.
- Make sure all new `.java` files to have a simple Javadoc class comment with at least an `@author`

tag identifying you, and preferably at least a paragraph on what the class is for.

- Add the ASF license header comment to all new `.java` files (copy from existing files in the project)
- Add yourself as an `@author` to the `.java` files that you modify substantially (more than cosmetic changes).
- Add some Javadocs and, if you change the namespace, some XSD doc elements.
- A few unit tests would help a lot as well — someone has to do it.
- If no-one else is using your branch, please rebase it against the current master (or other target branch in the main project).
- When writing a commit message please follow [these conventions](#), if you are fixing an existing issue please add `Fixes gh-XXXX` at the end of the commit message (where `XXXX` is the issue number).

20.4.4. Checkstyle

Spring Cloud Build comes with a set of checkstyle rules. You can find them in the `spring-cloud-build-tools` module. The most notable files under the module are:

`spring-cloud-build-tools/`

```
├── src
│   ├── checkstyle
│   │   ├── checkstyle-suppressions.xml ③
│   │   └── main
│   │       └── resources
│   │           ├── checkstyle-header.txt ②
│   │           └── checkstyle.xml ①
```

- ① Default Checkstyle rules
- ② File header setup
- ③ Default suppression rules

Checkstyle configuration

Checkstyle rules are **disabled by default**. To add checkstyle to your project just define the following properties and plugins.

```
<properties>
<maven-checkstyle-plugin.failOnError>true</maven-checkstyle-plugin.failOnError> ①
  <maven-checkstyle-plugin.failOnViolation>true
  </maven-checkstyle-plugin.failOnViolation> ②
  <maven-checkstyle-plugin.includeTestSourceDirectory>true
  </maven-checkstyle-plugin.includeTestSourceDirectory> ③
</properties>

<build>
  <plugins>
    <plugin> ④
      <groupId>io.spring.javaformat</groupId>
      <artifactId>spring-javaformat-maven-plugin</artifactId>
    </plugin>
    <plugin> ⑤
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-checkstyle-plugin</artifactId>
    </plugin>
  </plugins>

  <reporting>
    <plugins>
      <plugin> ⑤
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-checkstyle-plugin</artifactId>
      </plugin>
    </plugins>
  </reporting>
</build>
```

- ① Fails the build upon Checkstyle errors
- ② Fails the build upon Checkstyle violations
- ③ Checkstyle analyzes also the test sources
- ④ Add the Spring Java Format plugin that will reformat your code to pass most of the Checkstyle formatting rules
- ⑤ Add checkstyle plugin to your build and reporting phases

If you need to suppress some rules (e.g. line length needs to be longer), then it's enough for you to define a file under `${project.root}/src/checkstyle/checkstyle-suppressions.xml` with your suppressions. Example:

projectRoot/src/checkstyle/checkstyle-suppressions.xml

```
<?xml version="1.0"?>
<!DOCTYPE suppressions PUBLIC
    "-//Puppy Crawl//DTD Suppressions 1.1//EN"
    "https://www.puppycrawl.com/dtds/suppressions_1_1.dtd">
<suppressions>
  <suppress files=".*ConfigServerApplication\.java"
checks="HideUtilityClassConstructor"/>
  <suppress files=".*ConfigClientWatch\.java" checks="LineLengthCheck"/>
</suppressions>
```

It's advisable to copy the `${spring-cloud-build.rootFolder}/.editorconfig` and `${spring-cloud-build.rootFolder}/.springformat` to your project. That way, some default formatting rules will be applied. You can do so by running this script:

```
$ curl https://raw.githubusercontent.com/spring-cloud/spring-cloud-
build/master/.editorconfig -o .editorconfig
$ touch .springformat
```

20.4.5. IDE setup

IntelliJ IDEA

In order to setup IntelliJ you should import our coding conventions, inspection profiles and set up the checkstyle plugin. The following files can be found in the [Spring Cloud Build](#) project.

spring-cloud-build-tools/

```
├── src
│   ├── checkstyle
│   │   ├── checkstyle-suppressions.xml ③
│   │   └── main
│   │       ├── resources
│   │       │   ├── checkstyle-header.txt ②
│   │       │   ├── checkstyle.xml ①
│   │       │   └── intellij
│   │       │       ├── IntelliJ_Project_Defaults.xml ④
│   │       │       └── IntelliJ_Spring_Boot_Java_Conventions.xml ⑤
```

- ① Default Checkstyle rules
- ② File header setup
- ③ Default suppression rules
- ④ Project defaults for IntelliJ that apply most of Checkstyle rules
- ⑤ Project style conventions for IntelliJ that apply most of Checkstyle rules

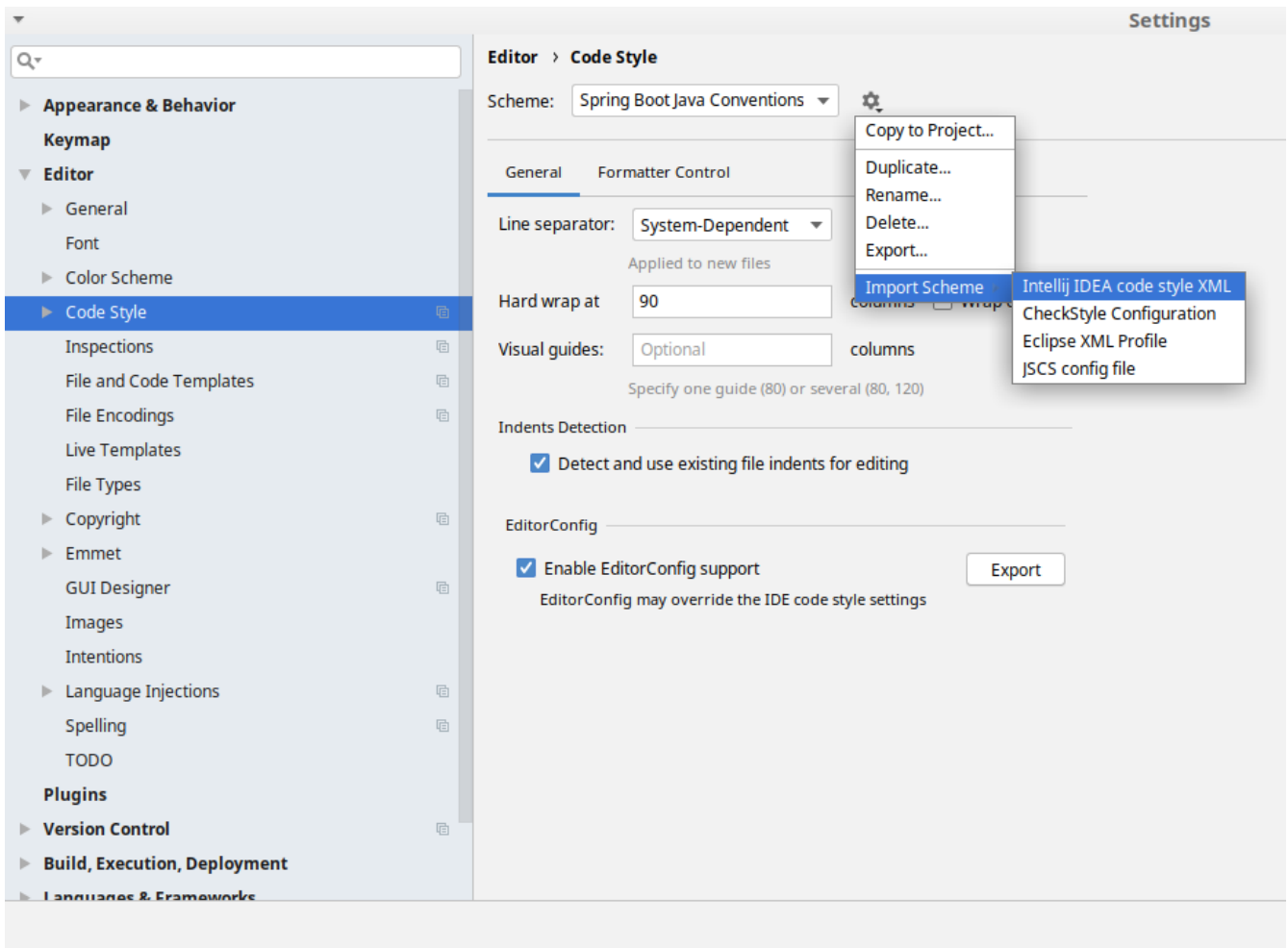


Figure 3. Code style

Go to **File** → **Settings** → **Editor** → **Code style**. There click on the icon next to the **Scheme** section. There, click on the **Import Scheme** value and pick the **IntelliJ IDEA code style XML** option. Import the `spring-cloud-build-tools/src/main/resources/intellij/IntelliJ_Spring_Boot_Java_Conventions.xml` file.

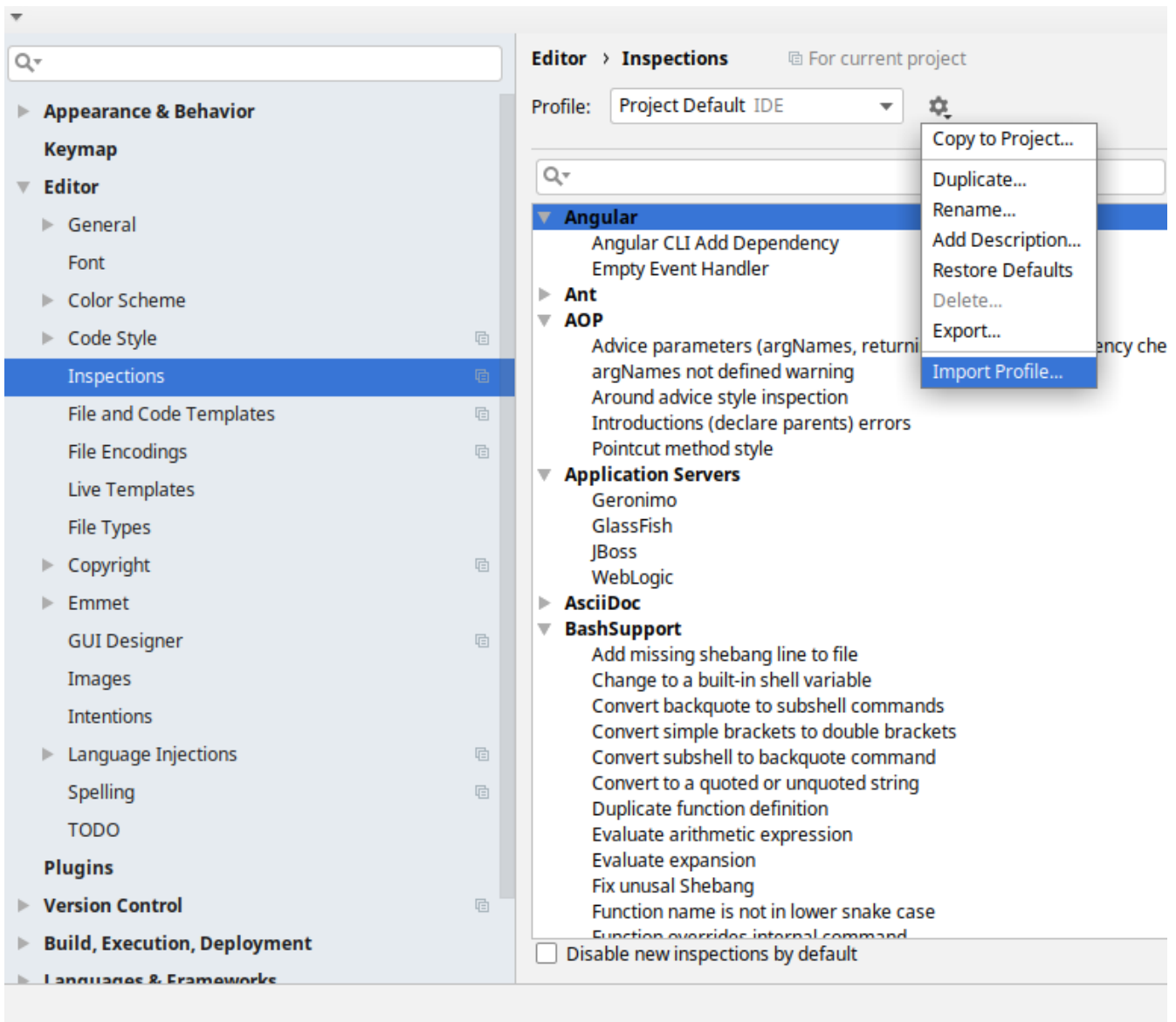
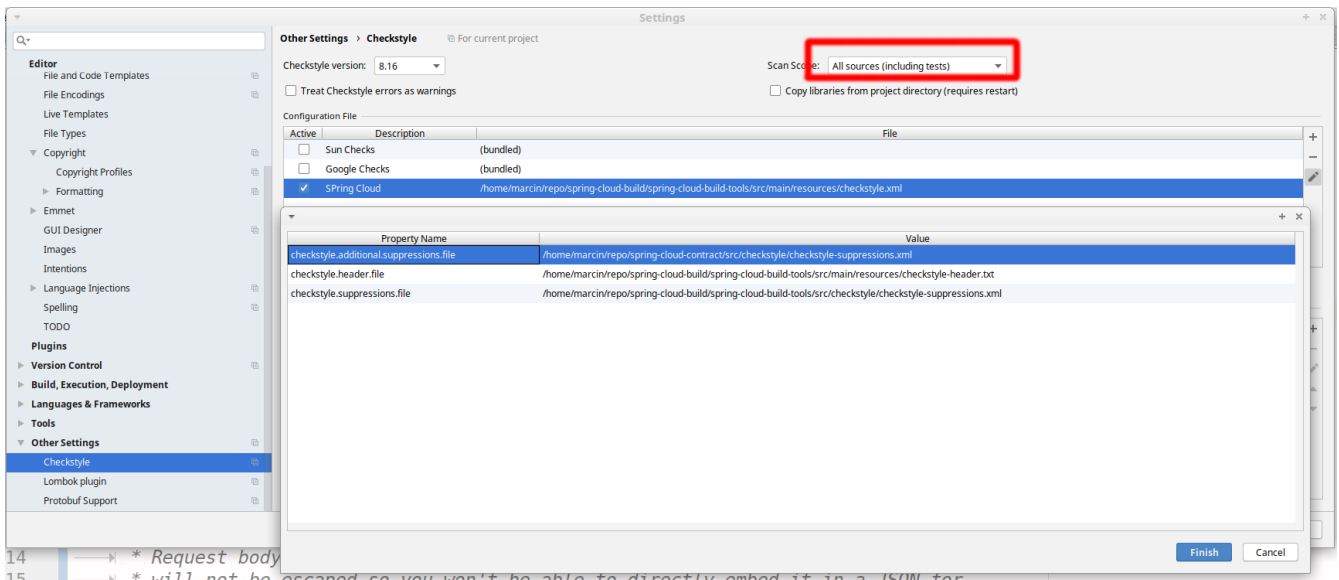


Figure 4. Inspection profiles

Go to **File** → **Settings** → **Editor** → **Inspections**. There click on the icon next to the **Profile** section. There, click on the **Import Profile** and import the `spring-cloud-build-tools/src/main/resources/intellij/Intellij_Project_Defaults.xml` file.

Checkstyle

To have IntelliJ work with Checkstyle, you have to install the **Checkstyle** plugin. It's advisable to also install the **Assertions2Assertj** to automatically convert the JUnit assertions



Go to **File** → **Settings** → **Other settings** → **Checkstyle**. There click on the **+** icon in the **Configuration file** section. There, you'll have to define where the checkstyle rules should be picked from. In the image above, we've picked the rules from the cloned Spring Cloud Build repository. However, you can point to the Spring Cloud Build's GitHub repository (e.g. for the `checkstyle.xml` : raw.githubusercontent.com/spring-cloud/spring-cloud-build/master/spring-cloud-build-tools/src/main/resources/checkstyle.xml). We need to provide the following variables:

- `checkstyle.header.file` - please point it to the Spring Cloud Build's, `spring-cloud-build-tools/src/main/resources/checkstyle-header.txt` file either in your cloned repo or via the raw.githubusercontent.com/spring-cloud/spring-cloud-build/master/spring-cloud-build-tools/src/main/resources/checkstyle-header.txt URL.
- `checkstyle.suppressions.file` - default suppressions. Please point it to the Spring Cloud Build's, `spring-cloud-build-tools/src/checkstyle/checkstyle-suppressions.xml` file either in your cloned repo or via the raw.githubusercontent.com/spring-cloud/spring-cloud-build/master/spring-cloud-build-tools/src/checkstyle/checkstyle-suppressions.xml URL.
- `checkstyle.additional.suppressions.file` - this variable corresponds to suppressions in your local project. E.g. you're working on `spring-cloud-contract`. Then point to the `project-root/src/checkstyle/checkstyle-suppressions.xml` folder. Example for `spring-cloud-contract` would be: `/home/username/spring-cloud-contract/src/checkstyle/checkstyle-suppressions.xml`.



Remember to set the **Scan Scope** to **All sources** since we apply checkstyle rules for production and test sources.

{spring-cloud-stream-version}

20.5. Preface

20.5.1. A Brief History of Spring's Data Integration Journey

Spring's journey on Data Integration started with [Spring Integration](#). With its programming model, it provided a consistent developer experience to build applications that can embrace [Enterprise Integration Patterns](#) to connect with external systems such as, databases, message brokers, and among others.

Fast forward to the cloud-era, where microservices have become prominent in the enterprise setting. [Spring Boot](#) transformed the way how developers built Applications. With Spring's programming model and the runtime responsibilities handled by Spring Boot, it became seamless to develop stand-alone, production-grade Spring-based microservices.

To extend this to Data Integration workloads, Spring Integration and Spring Boot were put together into a new project. Spring Cloud Stream was born.

With Spring Cloud Stream, developers can:

- Build, test and deploy data-centric applications in isolation.
- Apply modern microservices architecture patterns, including composition through messaging.
- Decouple application responsibilities with event-centric thinking. An event can represent something that has happened in time, to which the downstream consumer applications can react without knowing where it originated or the producer's identity.
- Port the business logic onto message brokers (such as RabbitMQ, Apache Kafka, Amazon Kinesis).
- Rely on the framework's automatic content-type support for common use-cases. Extending to different data conversion types is possible.
- and many more. . .

20.5.2. Quick Start

You can try Spring Cloud Stream in less than 5 min even before you jump into any details by following this three-step guide.

We show you how to create a Spring Cloud Stream application that receives messages coming from the messaging middleware of your choice (more on this later) and logs received messages to the console. We call it `LoggingConsumer`. While not very practical, it provides a good introduction to some of the main concepts and abstractions, making it easier to digest the rest of this user guide.

The three steps are as follows:

1. [Creating a Sample Application by Using Spring Initializr](#)
2. [Importing the Project into Your IDE](#)
3. [Adding a Message Handler, Building, and Running](#)

Creating a Sample Application by Using Spring Initializr

To get started, visit the [Spring Initializr](#). From there, you can generate our `LoggingConsumer` application. To do so:

1. In the **Dependencies** section, start typing `stream`. When the "Cloud Stream" option should appear, select it.
2. Start typing either 'kafka' or 'rabbit'.
3. Select "Kafka" or "RabbitMQ".

Basically, you choose the messaging middleware to which your application binds. We recommend using the one you have already installed or feel more comfortable with installing and running. Also, as you can see from the Initializer screen, there are a few other options you can choose. For example, you can choose Gradle as your build tool instead of Maven (the default).

4. In the **Artifact** field, type 'logging-consumer'.

The value of the **Artifact** field becomes the application name. If you chose RabbitMQ for the middleware, your Spring Initializr should now be as follows:

[spring initializr] | <https://raw.githubusercontent.com/spring-cloud/spring-cloud->

1. Click the **Generate Project** button.

Doing so downloads the zipped version of the generated project to your hard drive.

2. Unzip the file into the folder you want to use as your project directory.



We encourage you to explore the many possibilities available in the Spring Initializr. It lets you create many different kinds of Spring applications.

Importing the Project into Your IDE

Now you can import the project into your IDE. Keep in mind that, depending on the IDE, you may need to follow a specific import procedure. For example, depending on how the project was generated (Maven or Gradle), you may need to follow specific import procedure (for example, in Eclipse or STS, you need to use File → Import → Maven → Existing Maven Project).

Once imported, the project must have no errors of any kind. Also, `src/main/java` should contain `com.example.loggingconsumer.LoggingConsumerApplication`.

Technically, at this point, you can run the application's main class. It is already a valid Spring Boot application. However, it does not do anything, so we want to add some code.

Adding a Message Handler, Building, and Running

Modify the `com.example.loggingconsumer.LoggingConsumerApplication` class to look as follows:

```

@SpringBootApplication
public class LoggingConsumerApplication {

    public static void main(String[] args) {
        SpringApplication.run(LoggingConsumerApplication.class, args);
    }

    @Bean
    public Consumer<Person> log() {
        return person -> {
            System.out.println("Received: " + person);
        };
    }

    public static class Person {
        private String name;
        public String getName() {
            return name;
        }
        public void setName(String name) {
            this.name = name;
        }
        public String toString() {
            return this.name;
        }
    }
}

```

As you can see from the preceding listing:

- We are using functional programming model (see [Spring Cloud Function support](#)) to define a single message handler as `Consumer`.
- We are relying on framework conventions to bind such handler to the input destination binding exposed by the binder.

Doing so also lets you see one of the core features of the framework: It tries to automatically convert incoming message payloads to type `Person`.

You now have a fully functional Spring Cloud Stream application that does listens for messages. From here, for simplicity, we assume you selected RabbitMQ in [step one](#). Assuming you have RabbitMQ installed and running, you can start the application by running its `main` method in your IDE.

You should see following output:

```

--- [ main] c.s.b.r.p.RabbitExchangeQueueProvisioner : declaring queue for
inbound: input.anonymous.CbMIwdkJSB01ZoPDOtHtCg, bound to: input
--- [ main] o.s.a.r.c.CachingConnectionFactory      : Attempting to connect to:
[localhost:5672]
--- [ main] o.s.a.r.c.CachingConnectionFactory      : Created new connection:
rabbitConnectionFactory#2a3a299:0/SimpleConnection@66c83fc8. . .
. . .
--- [ main] o.s.i.a.i.AmqpInboundChannelAdapter     : started
inbound.input.anonymous.CbMIwdkJSB01ZoPDOtHtCg
. . .
--- [ main] c.e.l.LoggingConsumerApplication        : Started
LoggingConsumerApplication in 2.531 seconds (JVM running for 2.897)

```

Go to the RabbitMQ management console or any other RabbitMQ client and send a message to `input.anonymous.CbMIwdkJSB01ZoPDOtHtCg`. The `anonymous.CbMIwdkJSB01ZoPDOtHtCg` part represents the group name and is generated, so it is bound to be different in your environment. For something more predictable, you can use an explicit group name by setting `spring.cloud.stream.bindings.input.group=hello` (or whatever name you like).

The contents of the message should be a JSON representation of the `Person` class, as follows:

```
{"name": "Sam Spade"}
```

Then, in your console, you should see:

```
Received: Sam Spade
```

You can also build and package your application into a boot jar (by using `./mvnw clean install`) and run the built JAR by using the `java -jar` command.

Now you have a working (albeit very basic) Spring Cloud Stream application.

20.6. What's New in 3.0?

20.6.1. New Features and Enhancements

- **Routing Function** - see [\[Routing with functions\]](#) for more details.
- **Multiple bindings with functions** (multiple message handlers) - see [Multiple functions in a single application](#) for more details.
- **Functions with multiple inputs/outputs** (single function that can subscribe or target multiple destinations) - see [Functions with multiple input and output arguments](#) for more details.
- **Native support for reactive programming** - since v3.0.0 we no longer distribute `spring-cloud-stream-reactive` modules and instead relying on native reactive support provided by `spring cloud function`. For backward compatibility you can still bring `spring-cloud-stream-reactive` from previous versions.

20.6.2. Notable Deprecations

- *Reactive module* (`spring-cloud-stream-reactive`) is discontinued and no longer distributed in favor of native support via `spring-cloud-function`. For backward compatibility you can still bring `spring-cloud-stream-reactive` from previous versions.
- *Test support binder* `spring-cloud-stream-test-support` with `MessageCollector` in favor of a new test binder. See [Testing](#) for more details.
- `@StreamMessageConverter` - deprecated as it is no longer required.
- The `original-content-type` header references have been removed after it's been deprecated in v2.0.
- The `BinderAwareChannelResolver` is deprecated in favor of providing `spring.cloud.stream.sendto.destination` property. This is primarily for function-based programming model. For `StreamListener` it would still be required and thus will stay until we deprecate and eventually discontinue `StreamListener` and annotation-based programming model.

This section goes into more detail about how you can work with Spring Cloud Stream. It covers topics such as creating and running stream applications.

20.7. Introducing Spring Cloud Stream

Spring Cloud Stream is a framework for building message-driven microservice applications. Spring Cloud Stream builds upon Spring Boot to create standalone, production-grade Spring applications and uses Spring Integration to provide connectivity to message brokers. It provides opinionated configuration of middleware from several vendors, introducing the concepts of persistent publish-subscribe semantics, consumer groups, and partitions.

By simply adding `spring-cloud-stream` dependencies to the classpath of your application you'll get immediate connectivity to a message broker exposed via provided `spring-cloud-stream` binder (more on that later), and you can implement your functional requirement that will be executed based on the incoming message using simple `java.util.function.Function`

Here is the quick example:

```
@SpringBootApplication
public class SampleApplication {

    public static void main(String[] args) {
        SpringApplication.run(SampleApplication.class, args);
    }

    @Bean
    public Function<String, String> uppercase() {
        return value -> value.toUpperCase();
    }
}
```

and corresponding test

```
@SpringBootTest(classes = SampleApplication.class)
@Import({TestChannelBinderConfiguration.class})
class BootTestStreamApplicationTests {

    @Autowired
    private InputDestination input;

    @Autowired
    private OutputDestination output;

    @Test
    void contextLoads() {
        input.send(new GenericMessage<byte[]>("hello".getBytes()));
        assertThat(output.receive().getPayload()).isEqualTo("HELLO".getBytes());
    }
}
```

20.8. Main Concepts

Spring Cloud Stream provides a number of abstractions and primitives that simplify the writing of message-driven microservice applications. This section gives an overview of the following:

- [Spring Cloud Stream's application model](#)
- [The Binder Abstraction](#)
- [Persistent publish-subscribe support](#)
- [Consumer group support](#)
- [Partitioning support](#)
- [A pluggable Binder SPI](#)

20.8.1. Application Model

A Spring Cloud Stream application consists of a middleware-neutral core. The application communicates with the outside world by establishing *bindings* between destinations exposed by the external brokers and input/output arguments in your code. Broker specific details necessary to establish bindings are handled by middleware-specific *Binder* implementations.

[SCSt with binder] | <https://raw.githubusercontent.com/spring-cloud/spring-cloud->

Figure 5. Spring Cloud Stream Application

Fat JAR

Spring Cloud Stream applications can be run in stand-alone mode from your IDE for testing. To run a Spring Cloud Stream application in production, you can create an executable (or “fat”) JAR by using the standard Spring Boot tooling provided for Maven or Gradle. See the [Spring Boot Reference Guide](#) for more details.

20.8.2. The Binder Abstraction

Spring Cloud Stream provides Binder implementations for [Kafka](#) and [Rabbit MQ](#). The framework also includes a test binder for integration testing of your applications as `spring-cloud-stream` application. See [Testing](#) section for more details.

Binder abstraction is also one of the extension points of the framework, which means you can implement your own binder on top of Spring Cloud Stream. In the [How to create a Spring Cloud Stream Binder from scratch](#) post a community member documents in details, with an example, a set of steps necessary to implement a custom binder. The steps are also highlighted in the [Implementing Custom Binders](#) section.

Spring Cloud Stream uses Spring Boot for configuration, and the Binder abstraction makes it possible for a Spring Cloud Stream application to be flexible in how it connects to middleware. For example, deployers can dynamically choose, at runtime, the mapping between the external destinations (such as the Kafka topics or RabbitMQ exchanges) and inputs and outputs of the message handler (such as input parameter of the function and its return argument). Such configuration can be provided through external configuration properties and in any form supported by Spring Boot (including application arguments, environment variables, and `application.yml` or `application.properties` files). In the sink example from the [Introducing Spring Cloud Stream](#) section, setting the `spring.cloud.stream.bindings.input.destination` application property to `raw-sensor-data` causes it to read from the `raw-sensor-data` Kafka topic or from a queue bound to the `raw-sensor-data` RabbitMQ exchange.

Spring Cloud Stream automatically detects and uses a binder found on the classpath. You can use different types of middleware with the same code. To do so, include a different binder at build time. For more complex use cases, you can also package multiple binders with your application and have it choose the binder(and even whether to use different binders for different bindings) at runtime.

20.8.3. Persistent Publish-Subscribe Support

Communication between applications follows a publish-subscribe model, where data is broadcast through shared topics. This can be seen in the following figure, which shows a typical deployment for a set of interacting Spring Cloud Stream applications.

[SCSt sensors] | <https://raw.github.com/spring-cloud/spring-cloud->

Figure 6. Spring Cloud Stream Publish-Subscribe

Data reported by sensors to an HTTP endpoint is sent to a common destination named `raw-sensor-data`. From the destination, it is independently processed by a microservice application that computes time-windowed averages and by another microservice application that ingests the raw data into HDFS (Hadoop Distributed File System). In order to process the data, both applications declare the topic as their input at runtime.

The publish-subscribe communication model reduces the complexity of both the producer and the consumer and lets new applications be added to the topology without disruption of the existing flow. For example, downstream from the average-calculating application, you can add an application that calculates the highest temperature values for display and monitoring. You can then add another application that interprets the same flow of averages for fault detection. Doing all communication through shared topics rather than point-to-point queues reduces coupling between microservices.

While the concept of publish-subscribe messaging is not new, Spring Cloud Stream takes the extra step of making it an opinionated choice for its application model. By using native middleware support, Spring Cloud Stream also simplifies use of the publish-subscribe model across different platforms.

20.8.4. Consumer Groups

While the publish-subscribe model makes it easy to connect applications through shared topics, the ability to scale up by creating multiple instances of a given application is equally important. When doing so, different instances of an application are placed in a competing consumer relationship, where only one of the instances is expected to handle a given message.

Spring Cloud Stream models this behavior through the concept of a consumer group. (Spring Cloud Stream consumer groups are similar to and inspired by Kafka consumer groups.) Each consumer binding can use the `spring.cloud.stream.bindings.<bindingName>.group` property to specify a group name. For the consumers shown in the following figure, this property would be set as `spring.cloud.stream.bindings.<bindingName>.group=hdfsWrite` or `spring.cloud.stream.bindings.<bindingName>.group=average`.

[SCSt groups] | <https://raw.githubusercontent.com/spring-cloud/spring-cloud->

Figure 7. Spring Cloud Stream Consumer Groups

All groups that subscribe to a given destination receive a copy of published data, but only one member of each group receives a given message from that destination. By default, when a group is not specified, Spring Cloud Stream assigns the application to an anonymous and independent single-member consumer group that is in a publish-subscribe relationship with all other consumer groups.

20.8.5. Consumer Types

Two types of consumer are supported:

- Message-driven (sometimes referred to as Asynchronous)
- Polled (sometimes referred to as Synchronous)

Prior to version 2.0, only asynchronous consumers were supported. A message is delivered as soon as it is available and a thread is available to process it.

When you wish to control the rate at which messages are processed, you might want to use a synchronous consumer.

Durability

Consistent with the opinionated application model of Spring Cloud Stream, consumer group subscriptions are durable. That is, a binder implementation ensures that group subscriptions are persistent and that, once at least one subscription for a group has been created, the group receives messages, even if they are sent while all applications in the group are stopped.



Anonymous subscriptions are non-durable by nature. For some binder implementations (such as RabbitMQ), it is possible to have non-durable group subscriptions.

In general, it is preferable to always specify a consumer group when binding an application to a given destination. When scaling up a Spring Cloud Stream application, you must specify a consumer group for each of its input bindings. Doing so prevents the application's instances from receiving duplicate messages (unless that behavior is desired, which is unusual).

20.8.6. Partitioning Support

Spring Cloud Stream provides support for partitioning data between multiple instances of a given application. In a partitioned scenario, the physical communication medium (such as the broker topic) is viewed as being structured into multiple partitions. One or more producer application instances send data to multiple consumer application instances and ensure that data identified by common characteristics are processed by the same consumer instance.

Spring Cloud Stream provides a common abstraction for implementing partitioned processing use cases in a uniform fashion. Partitioning can thus be used whether the broker itself is naturally partitioned (for example, Kafka) or not (for example, RabbitMQ).

[SCSt partitioning] | <https://raw.githubusercontent.com/spring-cloud/spring-cloud->

Figure 8. Spring Cloud Stream Partitioning

Partitioning is a critical concept in stateful processing, where it is critical (for either performance or consistency reasons) to ensure that all related data is processed together. For example, in the time-windowed average calculation example, it is important that all measurements from any given sensor are processed by the same application instance.



To set up a partitioned processing scenario, you must configure both the data-producing and the data-consuming ends.

20.9. Programming Model

To understand the programming model, you should be familiar with the following core concepts:

- **Destination Binders:** Components responsible to provide integration with the external messaging systems.
- **Bindings:** Bridge between the external messaging systems and application provided *Producers* and *Consumers* of messages (created by the Destination Binders).
- **Message:** The canonical data structure used by producers and consumers to communicate with Destination Binders (and thus other applications via external messaging systems).

[SCSt overview] | <https://raw.githubusercontent.com/spring-cloud/spring-cloud->

20.9.1. Destination Binders

Destination Binders are extension components of Spring Cloud Stream responsible for providing the necessary configuration and implementation to facilitate integration with external messaging systems. This integration is responsible for connectivity, delegation, and routing of messages to and from producers and consumers, data type conversion, invocation of the user code, and more.

Binders handle a lot of the boiler plate responsibilities that would otherwise fall on your shoulders. However, to accomplish that, the binder still needs some help in the form of minimalistic yet required set of instructions from the user, which typically come in the form of some type of *binding* configuration.

While it is out of scope of this section to discuss all of the available binder and binding configuration options (the rest of the manual covers them extensively), *Binding* as a concept, does require special attention. The next section discusses it in detail.

20.9.2. Bindings

As stated earlier, *Bindings* provide a bridge between the external messaging system (e.g., queue, topic etc.) and application-provided *Producers* and *Consumers*.

The following example shows a fully configured and functioning Spring Cloud Stream application that receives the payload of the message as a `String` type (see [Content Type Negotiation](#) section), logs it to the console and sends it down stream after converting it to upper case.

```
@SpringBootApplication
public class SampleApplication {

    public static void main(String[] args) {
        SpringApplication.run(SampleApplication.class, args);
    }

    @Bean
    public Function<String, String> uppercase() {
        return value -> {
            System.out.println("Received: " + value);
            return value.toUpperCase()
        };
    }
}
```

Unlike previous versions of spring-cloud-stream which relied on `@EnableBinding` and `@StreamListener` annotations, the above example looks no different than any vanilla spring-boot application. It defines a single bean of type `Function` and that it is. So, how does it become spring-cloud-stream application? It becomes spring-cloud-stream application simply based on the presence of spring-cloud-stream and binder dependencies and auto-configuration classes on the classpath effectively setting the context for your boot application as spring-cloud-stream

application. And in this context beans of type `Supplier`, `Function` or `Consumer` are treated as defacto message handlers triggering binding of to destinations exposed by the provided binder following certain naming conventions and rules to avoid extra configuration.

Binding and Binding names

Binding is an abstraction that represents a bridge between sources and targets exposed by the binder and user code, This abstraction has a name and while we try to do our best to limit configuration required to run spring-cloud-stream applications, being aware of such name(s) is necessary for cases where additional per-binding configuration is required.

Throughout this manual you will see examples of configuration properties such as `spring.cloud.stream.bindings.input.destination=myQueue`. The `input` segment in this property name is what we refer to as *binding name* and it could derive via several mechanisms. The following subsections will describe the naming conventions and configuration elements used by spring-cloud-stream to control binding names.

Functional binding names

Unlike the explicit naming required by annotation-based support (legacy) used in the previous versions of spring-cloud-stream, the functional programming model defaults to a simple convention when it comes to binding names, thus greatly simplifying application configuration. Let's look at the first example:

```
@SpringBootApplication
public class SampleApplication {

    @Bean
    public Function<String, String> uppercase() {
        return value -> value.toUpperCase();
    }
}
```

In the preceding example we have an application with a single function which acts as message handler. As a `Function` it has an input and output. The naming convention used to name input and output bindings is as follows:

- input - `<functionName> + -in- + <index>`
- output - `<functionName> + -out- + <index>`

The `in` and `out` corresponds to the type of binding (such as *input* or *output*). The `index` is the index of the input or output binding. It is always 0 for typical single input/output function, so it's only relevant for [Functions with multiple input and output arguments](#).

So if for example you would want to map the input of this function to a remote destination (e.g., topic, queue etc) called "my-topic" you would do so with the following property:

```
--spring.cloud.stream.bindings.uppercase-in-0.destination=my-topic
```

Note how `uppercase-in-0` is used as a segment in property name. The same goes for `uppercase-out-0`.

Descriptive Binding Names

Some times to improve readability you may want to give your binding a more descriptive name (such as 'account', 'orders` etc). Another way of looking at it is you can map an *implicit binding name* to an *explicit binding name*. And you can do it with `spring.cloud.stream.function.bindings.<binding-name>` property. This property also provides a migration path for existing applications that rely on custom interface-based bindings that require explicit names.

For example,

```
--spring.cloud.stream.function.bindings.uppercase-in-0=input`
```

In the preceding example you mapped and effectively renamed `uppercase-in-0` binding name to `input`. Now all configuration properties can refer to `input` binding name instead (e.g., `--spring.cloud.stream.bindings.input.destination=my-topic`).



While descriptive binding names may enhance the readability aspect of the configuration, they also create another level of misdirection by mapping an implicit binding name to an explicit binding name. And since all subsequent configuration properties will use the explicit binding name you must always refer to this 'bindings' property to correlate which function it actually corresponds to. We believe that for most cases (with the exception of [Functional Composition](#)) it may be an overkill, so, it is our recommendation to avoid using it all together, especially since not using it provides a clear path between binder destination and binding name, such as `spring.cloud.stream.bindings.uppercase-in-0.destination=sample-topic`, where you are clearly correlating the input of `uppercase` function to `sample-topic` destination.

For more on properties and other configuration options please see [Configuration Options](#) section.

Annotation-based binding names (legacy)

In previous versions of spring-cloud-stream *binding* names and in fact implementations, derived from the `@EnableBinding` annotation which typically would take one or more interface classes as parameters. The parameters are referred to as *bindings*, and they contain methods representing *bindable components*.

For compliance with legacy style applications we still support this annotation-based programming model and you can get more information about it in [Annotation-based support \(legacy\)](#) section (sub-section of the [Programming Model](#) section).

Spring Cloud Stream already provides *binding* interfaces for typical message exchange contracts,

which include:

- **Sink:** Identifies the contract for the message consumer by providing the destination from which the message is consumed.
- **Source:** Identifies the contract for the message producer by providing the destination to which the produced message is sent.
- **Processor:** Encapsulates both the sink and the source contracts by exposing two destinations that allow consumption and production of messages.

```
public interface Sink {  
  
    String INPUT = "input";  
  
    @Input(Sink.INPUT)  
    SubscribableChannel input();  
}
```

```
public interface Source {  
  
    String OUTPUT = "output";  
  
    @Output(Source.OUTPUT)  
    MessageChannel output();  
}
```

```
public interface Processor extends Source, Sink {}
```

And you can define your own interfaces as well

```
public interface MyBinding {  
  
    String FOO = "foo";  
  
    @Output(MyBinding.FOO)  
    MessageChannel foo();  
}
```



The reason why `@EnableBinding` and binding interfaces are not required with functional programming model is because they could be derived from the type of functional interface itself. For example, *Processor* = *Function*, *Source* = *Supplier* and so on.

Pollable Destination Binding

While the previously described bindings support event-based message consumption, sometimes you need more control, such as rate of consumption.

Starting with version 2.0, you can now bind a pollable consumer:

The following example shows how to bind a pollable consumer:

```
public interface PolledBarista {  
  
    @Input  
    PollableMessageSource orders();  
    . . .  
}
```

In this case, an implementation of `PollableMessageSource` is bound to the `orders` “channel”. See [Using Polled Consumers](#) for more details.

20.9.3. Producing and Consuming Messages

You can write a Spring Cloud Stream application by simply writing functions and exposing them as `@Bean`'s. You can also use Spring Integration annotations based configuration or Spring Cloud Stream annotation based configuration, although starting with `spring-cloud-stream 3.x` we recommend using functional implementations.

Spring Cloud Function support

Overview

Since Spring Cloud Stream v2.1, another alternative for defining *stream handlers* and *sources* is to use build-in support for [Spring Cloud Function](#) where they can be expressed as beans of type `java.util.function.[Supplier/Function/Consumer]`.

To specify which functional bean to bind to the external destination(s) exposed by the bindings, you must provide `spring.cloud.function.definition` property.



In the event you only have single bean of type `java.util.function.[Supplier/Function/Consumer]`, you can skip the `spring.cloud.function.definition` property, since such functional bean will be auto-discovered. However, it is considered best practice to use such property to avoid any confusion.

Here is the example of the application exposing message handler as `java.util.function.Function` effectively supporting *pass-thru* semantics by acting as consumer and producer of data.


```

@SpringBootApplication
public class MyFunctionBootTest {

    public static void main(String[] args) {
        SpringApplication.run(MyFunctionBootTest.class);
    }

    @Bean
    public Function<String, String> toUpperCase() {
        return s -> s.toUpperCase();
    }
}

```

In the preceding example, we define a bean of type `java.util.function.Function` called `toUpperCase` to be acting as message handler whose 'input' and 'output' must be bound to the external destinations exposed by the provided destination binder. By default the 'input' and 'output' binding names will be `toUpperCase-in-0` and `toUpperCase-out-0`. Please see [Functional binding names](#) section for details on naming convention used to establish binding names.

Below are the examples of simple functional applications to support other semantics:

Here is the example of a *source* semantics exposed as `java.util.function.Supplier`

```

@SpringBootApplication
public static class SourceFromSupplier {

    @Bean
    public Supplier<Date> date() {
        return () -> new Date(12345L);
    }
}

```

Here is the example of a *sink* semantics exposed as `java.util.function.Consumer`

```

@SpringBootApplication
public static class SinkFromConsumer {

    @Bean
    public Consumer<String> sink() {
        return System.out::println;
    }
}

```

Suppliers (Sources)

`Function` and `Consumer` are pretty straightforward when it comes to how their invocation is triggered. They are triggered based on data (events) sent to the destination they are bound to. In

other words, they are classic event-driven components.

However, `Supplier` is in its own category when it comes to triggering. Since it is, by definition, the source (the origin) of the data, it does not subscribe to any in-bound destination and, therefore, has to be triggered by some other mechanism(s). There is also a question of `Supplier` implementation, which could be *imperative* or *reactive* and which directly relates to the triggering of such suppliers.

Consider the following sample:

```
@SpringBootApplication
public static class SupplierConfiguration {

    @Bean
    public Supplier<String> stringSupplier() {
        return () -> "Hello from Supplier";
    }
}
```

The preceding `Supplier` bean produces a string whenever its `get()` method is invoked. However, who invokes this method and how often? The framework provides a default polling mechanism (answering the question of "Who?") that will trigger the invocation of the supplier and by default it will do so every second (answering the question of "How often?"). In other words, the above configuration produces a single message every second and each message is sent to an `output` destination that is exposed by the binder. To learn how to customize the polling mechanism, see [Polling Configuration Properties](#) section.

Consider a different sample:

```
@SpringBootApplication
public static class SupplierConfiguration {

    @Bean
    public Supplier<Flux<String>> stringSupplier() {
        return () -> Flux.fromStream(Stream.generate(new Supplier<String>() {
            @Override
            public String get() {
                try {
                    Thread.sleep(1000);
                    return "Hello from Supplier";
                } catch (Exception e) {
                    // ignore
                }
            }
        })).subscribeOn(Schedulers.elastic()).share();
    }
}
```

The preceding `Supplier` bean adopts the reactive programming style. Typically, and unlike the

imperative supplier, it should be triggered only once, given that the invocation of its `get()` method produces (supplies) the continuous stream of messages and not an individual message.

The framework recognizes the difference in the programming style and guarantees that such a supplier is triggered only once.

However, imagine the use case where you want to poll some data source and return a finite stream of data representing the result set. The reactive programming style is a perfect mechanism for such a Supplier. However, given the finite nature of the produced stream, such Supplier still needs to be invoked periodically.

Consider the following sample, which emulates such use case by producing a finite stream of data:

```
@SpringBootApplication
public static class SupplierConfiguration {

    @PollableBean
    public Supplier<Flux<String>> stringSupplier() {
        return () -> Flux.just("hello", "bye");
    }
}
```

The bean itself is annotated with `PollableBean` annotation (sub-set of `@Bean`), thus signaling to the framework that although the implementation of such a supplier is reactive, it still needs to be polled.



There is a `splittable` attribute defined in `PollableBean` which signals to the post processors of this annotation that the result produced by the annotated component has to be split and is set to `true` by default. It means that the framework will split the returning sending out each item as an individual message. If this is not the desired behavior you can set it to `false` at which point such supplier will simply return the produced Flux without splitting it.

Consumer (Reactive)

Reactive `Consumer` is a little bit special because it has a void return type, leaving framework with no reference to subscribe to. Most likely you will not need to write `Consumer<Flux<?>>`, and instead write it as a `Function<Flux<?>, Mono<Void>>` invoking `then` operator as the last operator on your stream.

For example:

```
public Function<Flux<?>, Mono<Void>>`consumer() {
    return flux -> flux.map(..).filter(..).then();
}
```

But if you do need to write an explicit `Consumer<Flux<?>>`, remember to subscribe to the incoming Flux.

Polling Configuration Properties

The following properties are exposed by `org.springframework.cloud.stream.config.DefaultPollerProperties` and are prefixed with `spring.cloud.stream.poller:`

fixedDelay

Fixed delay for default poller in milliseconds.

Default: 1000L.

maxMessagesPerPoll

Maximum messages for each polling event of the default poller.

Default: 1L.

For example `--spring.cloud.stream.poller.fixed-delay=2000` sets the poller interval to poll every two seconds.

Foreign event-driven sources

There are cases where the actual source of data may be coming from the external (foreign) system that is not a binder. For example, the source of the data may be a classic REST endpoint. How do we bridge such source with the functional mechanism used by spring-cloud-stream?

Spring Cloud Stream provides two mechanisms, so let's look at them in more details

Here, for both samples we'll use a standard MVC endpoint method called `delegateToSupplier` bound to the root web context, delegating incoming requests to stream via two different mechanisms - imperative (via `StreamBridge`) and reactive (via `EmitterProcessor`).

Using `StreamBridge`

```

@SpringBootApplication
@Controller
public class WebSourceApplication {

    public static void main(String[] args) {
        SpringApplication.run(WebSourceApplication.class, "--spring.cloud.stream.source=toStream");
    }

    @Autowired
    private StreamBridge streamBridge;

    @RequestMapping
    @ResponseStatus(HttpStatus.ACCEPTED)
    public void delegateToSupplier(@RequestBody String body) {
        System.out.println("Sending " + body);
        streamBridge.send("toStream-out-0", body);
    }
}

```

Here we autowire a `StreamBridge` bean which allows us to send data to an output binding effectively bridging non-stream application with spring-cloud-stream. Note that preceding example does not have any source functions defined (e.g., Supplier bean) leaving the framework with no trigger to create source bindings, which would be typical for cases where configuration contains function beans. So to trigger the creation of source binding we use `spring.cloud.stream.source` property where you can declare the name of your sources. The provided name will be used as a trigger to create a source binding. So in the preceding example the name of the output binding will be `toStream-out-0` which is consistent with the binding naming convention used by functions (see [Binding and Binding names](#)). You can use `;` to signify multiple sources (e.g., `--spring.cloud.stream.source=foo;bar`)

Also, note that `streamBridge.send(..)` method takes an `Object` for data. This means you can send POJO or `Message` to it and it will go through the same routine when sending output as if it was from any Function or Supplier providing the same level of consistency as with functions. This means the output type conversion, partitioning etc are honored as if it was from the output produced by functions.

StreamBridge and Dynamic Destinations

`StreamBridge` can also be used for cases when output destination(s) are not known ahead of time similar to the use cases described in [Routing FROM Consumer](#) section.

Let's look at the example

```

@SpringBootApplication
@Controller
public class WebSourceApplication {

    public static void main(String[] args) {
        SpringApplication.run(WebSourceApplication.class, args);
    }

    @Autowired
    private StreamBridge streamBridge;

    @RequestMapping
    @ResponseStatus(HttpStatus.ACCEPTED)
    public void delegateToSupplier(@RequestBody String body) {
        System.out.println("Sending " + body);
        streamBridge.send("myDestiniation", body);
    }
}

```

As you can see the preceding example is very similar to the previous one with the exception of explicit binding instruction provided via `spring.cloud.stream.source` property (which is not provided). Here we're sending data to `myDestiniation` name which does not exist as a binding. Therefore such name will be treated as dynamic destination as described in [Routing FROM Consumer](#) section.

Using reactor API

```

@SpringBootApplication
@Controller
public class WebSourceApplication {

    public static void main(String[] args) {
        SpringApplication.run(WebSourceApplication.class);
    }

    EmitterProcessor<String> processor = EmitterProcessor.create();

    @RequestMapping
    @ResponseStatus(HttpStatus.ACCEPTED)
    public void delegateToSupplier(@RequestBody String body) {
        processor.onNext(body);
    }

    @Bean
    public Supplier<Flux<String>> supplier() {
        return () -> processor;
    }
}

```

Here we declare a `Supplier` bean which returns `Flux` of `Strings`. This example uses `EmitterProcessor` from the reactor API (see [Reactive Functions support](#) for more details) to effectively provide a bridge between the actual event source (rest endpoint in this case) and spring-cloud-stream. All you need to do is define a `Supplier<Flux<your-type>>` and return the `EmitterProcessor` while feeding the incoming data via `EmitterProcessor#onNext(data)` operation.

You can now send message to spring-cloud-stream source as

```
curl -H "Content-Type: text/plain" -X POST -d "hello from the other side"
http://localhost:8080/
```

And while this example demonstrates bridging web endpoint with the Supplier of data that will be fed into spring-cloud-stream framework, the approach can be used with other type of foreign sources.

Reactive Functions support

Since *Spring Cloud Function* is build on top of [Project Reactor](#) there isn't much you need to do to benefit from reactive programming model while implementing `Supplier`, `Function` or `Consumer`.

For example:

```
@SpringBootApplication
public static class SinkFromConsumer {

    @Bean
    public Function<Flux<String>, Flux<String>> reactiveUpperCase() {
        return flux -> flux.map(val -> val.toUpperCase());
    }
}
```

Functional Composition

Using functional programming model you can also benefit from functional composition where you can dynamically compose complex handlers from a set of simple functions. As an example let's add the following function bean to the application defined above

```
@Bean
public Function<String, String> wrapInQuotes() {
    return s -> "\"" + s + "\"";
}
```

and modify the `spring.cloud.function.definition` property to reflect your intention to compose a new function from both 'toUpperCase' and 'wrapInQuotes'. To do so Spring Cloud Function relies on `|` (pipe) symbol. So, to finish our example our property will now look like this:

```
--spring.cloud.function.definition=toUpperCase|wrapInQuotes
```



One of the great benefits of functional composition support provided by *Spring Cloud Function* is the fact that you can compose *reactive* and *imperative* functions.

The result of a composition is a single function which, as you may guess, could have a very long and rather cryptic name (e.g., `foo|bar|baz|xyz. . .`) presenting a great deal of inconvenience when it comes to other configuration properties. This is where *descriptive binding names* feature described in [Functional binding names](#) section can help.

For example, if we want to give our `toUpperCase|wrapInQuotes` a more descriptive name we can do so with the following property `spring.cloud.stream.function.bindings.toUpperCase|wrapInQuotes=quotedUpperCase` allowing other configuration properties to refer to that binding name (e.g., `spring.cloud.stream.bindings.quotedUpperCase.destination=myDestination`).

Functions with multiple input and output arguments

Starting with version 3.0 `spring-cloud-stream` provides support for functions that have multiple inputs and/or multiple outputs (return values). What does this actually mean and what type of use cases it is targeting?

- *Big Data: Imagine the source of data you're dealing with is highly un-organized and contains various types of data elements (e.g., orders, transactions etc) and you effectively need to sort it out.*
- *Data aggregation: Another use case may require you to merge data elements from 2+ incoming _streams.*

The above describes just a few use cases where you may need to use a single function to accept and/or produce multiple *streams* of data. And that is the type of use cases we are targeting here.

Also, note a slightly different emphasis on the concept of *streams* here. The assumption is that such functions are only valuable if they are given access to the actual streams of data (not the individual elements). So for that we are relying on abstractions provided by [Project Reactor](#) (i.e., `Flux` and `Mono`) which is already available on the classpath as part of the dependencies brought in by `spring-cloud-functions`.

Another important aspect is representation of multiple input and outputs. While java provides variety of different abstractions to represent *multiple of something* those abstractions are *a) unbounded, b) lack arity* and *c) lack type information* which are all important in this context. As an example, let's look at `Collection` or an array which only allows us to describe *multiple* of a single type or up-cast everything to an `Object`, affecting transparent type conversion feature of `spring-cloud-stream` and so on.

So to accommodate all these requirements the initial support is relying on the signature which utilizes another abstraction provided by *Project Reactor* - `Tuples`. However, we are working on allowing a more flexible signatures.



Please refer to [Binding and Binding names](#) section to understand the naming convention used to establish *binding names* used by such application.

Let's look at the few samples:

```
@SpringBootApplication
public class SampleApplication {

    @Bean
    public Function<Tuple2<Flux<String>, Flux<Integer>>, Flux<String>> gather() {
        return tuple -> {
            Flux<String> stringStream = tuple.getT1();
            Flux<String> intStream = tuple.getT2().map(i -> String.valueOf(i));
            return Flux.merge(stringStream, intStream);
        };
    }
}
```

The above example demonstrates function which takes two inputs (first of type `String` and second of type `Integer`) and produces a single output of type `String`.

So, for the above example the two input bindings will be `gather-in-0` and `gather-in-1` and for consistency the output binding also follows the same convention and is named `gather-out-0`.

Knowing that will allow you to set binding specific properties the same way you did with `@StreamListener`. For example, the following will override content-type for `gather-in-0` binding:

```
--spring.cloud.stream.bindings.gather-in-0.content-type=text/plain
```

```

@SpringBootApplication
public class SampleApplication {

    @Bean
    public static Function<Flux<Integer>, Tuple2<Flux<String>, Flux<String>>>
scatter() {
    return flux -> {
        Flux<Integer> connectedFlux = flux.publish().autoConnect(2);
        UnicastProcessor even = UnicastProcessor.create();
        UnicastProcessor odd = UnicastProcessor.create();
        Flux<Integer> evenFlux = connectedFlux.filter(number -> number % 2 ==
0).doOnNext(number -> even.onNext("EVEN: " + number));
        Flux<Integer> oddFlux = connectedFlux.filter(number -> number % 2 !=
0).doOnNext(number -> odd.onNext("ODD: " + number));

        return Tuples.of(Flux.from(even).doOnSubscribe(x -> evenFlux.subscribe()),
Flux.from(odd).doOnSubscribe(x -> oddFlux.subscribe()));
    };
}
}

```

The above example is somewhat of a the opposite from the previous sample and demonstrates function which takes single input of type `Integer` and produces two outputs (both of type `String`).

So, for the above example the input binding is `gather-in-0` and the output bindings are `gather-out-0` and `gather-out-1`.

And you test it with the following code:

```

@Test
public void testSingleInputMultiOutput() {
    try (ConfigurableApplicationContext context = new SpringApplicationBuilder(
        TestChannelBinderConfiguration.getCompleteConfiguration(
            SampleApplication.class))
        .run("--spring.cloud.function.definition=scatter")) {
        context.getBean(InputDestination.class);

        InputDestination inputDestination = context.getBean(InputDestination.class);
        OutputDestination outputDestination =
context.getBean(OutputDestination.class);

        for (int i = 0; i < 10; i++) {

inputDestination.send(MessageBuilder.withPayload(String.valueOf(i).getBytes()).build()
);
        }

        int counter = 0;
        for (int i = 0; i < 5; i++) {
            Message<byte[]> even = outputDestination.receive(0, 0);
            assertThat(even.getPayload()).isEqualTo(("EVEN: " +
String.valueOf(counter++)).getBytes());
            Message<byte[]> odd = outputDestination.receive(0, 1);
            assertThat(odd.getPayload()).isEqualTo(("ODD: " +
String.valueOf(counter++)).getBytes());
        }
    }
}

```

Multiple functions in a single application

There may also be a need for grouping several message handlers in a single application. You would do so by defining several functions.

```

@SpringBootApplication
public class SampleApplication {

    @Bean
    public Function<String, String> uppercase() {
        return value -> value.toUpperCase();
    }

    @Bean
    public Function<String, String> reverse() {
        return value -> new StringBuilder(value).reverse().toString();
    }
}

```

In the above example we have configuration which defines two functions `uppercase` and `reverse`. So first, as mentioned before, we need to notice that there is a conflict (more than one function) and therefore we need to resolve it by providing `spring.cloud.function.definition` property pointing to the actual function we want to bind. Except here we will use `;` delimiter to point to both functions (see test case below).



As with functions with multiple inputs/outputs, please refer to [Binding and Binding names](#) section to understand the naming convention used to establish *binding names* used by such application.

And you test it with the following code:

```
@Test
public void testMultipleFunctions() {
    try (ConfigurableApplicationContext context = new SpringApplicationBuilder(
        TestChannelBinderConfiguration.getCompleteConfiguration(
            ReactiveFunctionConfiguration.class))
        .run("--
spring.cloud.function.definition=uppercase;reverse")) {
        context.getBean(InputDestination.class);

        InputDestination inputDestination = context.getBean(InputDestination.class);
        OutputDestination outputDestination =
context.getBean(OutputDestination.class);

        Message<byte[]> inputMessage =
MessageBuilder.withPayload("Hello".getBytes()).build();
        inputDestination.send(inputMessage, "uppercase-in-0");
        inputDestination.send(inputMessage, "reverse-in-0");

        Message<byte[]> outputMessage = outputDestination.receive(0, "uppercase-out-
0");
        assertThat(outputMessage.getPayload()).isEqualTo("HELLO".getBytes());

        outputMessage = outputDestination.receive(0, "uppercase-out-1");
        assertThat(outputMessage.getPayload()).isEqualTo("olleH".getBytes());
    }
}
```

Batch Consumers

When using a `MessageChannelBinder` that supports batch listeners, and the feature is enabled for the consumer binding, you can set `spring.cloud.stream.bindings.<binding-name>.consumer.batch-mode` to `true` to enable the entire batch of messages to be passed to the function in a `List`.

```
@Bean
public Function<List<Person>, Person> findFirstPerson() {
    return persons -> persons.get(0);
}
```

Spring Integration flow as functions

When you implement a function, you may have complex requirements that fit the category of [Enterprise Integration Patterns](#) (EIP). These are best handled by using a framework such as [Spring Integration](#) (SI), which is a reference implementation of EIP.

Thankfully SI already provides support for exposing integration flows as functions via [Integration flow as gateway](#). Consider the following sample:

```
@SpringBootApplication
public class FunctionSampleSpringIntegrationApplication {

    public static void main(String[] args) {
        SpringApplication.run(FunctionSampleSpringIntegrationApplication.class, args);
    }

    @Bean
    public IntegrationFlow uppercaseFlow() {
        return IntegrationFlows.from(MessageFunction.class, "uppercase")
            .<String, String>transform(String::toUpperCase)
            .logAndReply(LoggingHandler.Level.WARN);
    }

    public interface MessageFunction extends Function<Message<String>,
        Message<String>> {

    }
}
```

For those who are familiar with SI you can see we define a bean of type `IntegrationFlow` where we declare an integration flow that we want to expose as a `Function<String, String>` (using SI DSL) called `uppercase`. The `MessageFunction` interface lets us explicitly declare the type of the inputs and outputs for proper type conversion. See [Content Type Negotiation](#) section for more on type conversion.

To receive raw input you can use `from(Function.class, ...)`.

The resulting function is bound to the input and output destinations exposed by the target binder.



Please refer to [Binding and Binding names](#) section to understand the naming convention used to establish *binding names* used by such application.

For more details on interoperability of Spring Integration and Spring Cloud Stream specifically

around functional programming model you may find [this post](#) very interesting, as it dives a bit deeper into various patterns you can apply by merging the best of Spring Integration and Spring Cloud Stream/Functions.

Annotation-based support (legacy)

As mentioned earlier you can also use Spring Integration annotations based configuration or Spring Cloud Stream annotation based configuration.

Spring Integration Support

Spring Cloud Stream is built on the concepts and patterns defined by [Enterprise Integration Patterns](#) and relies in its internal implementation on an already established and popular implementation of Enterprise Integration Patterns within the Spring portfolio of projects: [Spring Integration](#) framework.

So its only natural for it to support the foundation, semantics, and configuration options that are already established by Spring Integration

For example, you can attach the output channel of a [Source](#) to a [MessageSource](#) and use the familiar [@InboundChannelAdapter](#) annotation, as follows:

```
@EnableBinding(Source.class)
public class TimerSource {

    @Bean
    @InboundChannelAdapter(value = Source.OUTPUT, poller = @Poller(fixedDelay = "10",
maxMessagesPerPoll = "1"))
    public MessageSource<String> timerMessageSource() {
        return () -> new GenericMessage<>("Hello Spring Cloud Stream");
    }
}
```

Similarly, you can use [@Transformer](#) or [@ServiceActivator](#) while providing an implementation of a message handler method for a *Processor* binding contract, as shown in the following example:

```
@EnableBinding(Processor.class)
public class TransformProcessor {
    @Transformer(inputChannel = Processor.INPUT, outputChannel = Processor.OUTPUT)
    public Object transform(String message) {
        return message.toUpperCase();
    }
}
```



While this may be skipping ahead a bit, it is important to understand that, when you consume from the same binding using `@StreamListener` annotation, a pub-sub model is used. Each method annotated with `@StreamListener` receives its own copy of a message, and each one has its own consumer group. However, if you consume from the same binding by using one of the Spring Integration annotation (such as `@Aggregator`, `@Transformer`, or `@ServiceActivator`), those consume in a competing model. No individual consumer group is created for each subscription.

Using `@StreamListener` Annotation

Complementary to its Spring Integration support, Spring Cloud Stream provides its own `@StreamListener` annotation, modeled after other Spring Messaging annotations (`@MessageMapping`, `@JmsListener`, `@RabbitListener`, and others) and provides conveniences, such as content-based routing and others.

```
@EnableBinding(Sink.class)
public class VoteHandler {

    @Autowired
    VotingService votingService;

    @StreamListener(Sink.INPUT)
    public void handle(Vote vote) {
        votingService.record(vote);
    }
}
```

As with other Spring Messaging methods, method arguments can be annotated with `@Payload`, `@Headers`, and `@Header`.

For methods that return data, you must use the `@SendTo` annotation to specify the output binding destination for data returned by the method, as shown in the following example:

```
@EnableBinding(Processor.class)
public class TransformProcessor {

    @Autowired
    VotingService votingService;

    @StreamListener(Processor.INPUT)
    @SendTo(Processor.OUTPUT)
    public VoteResult handle(Vote vote) {
        return votingService.record(vote);
    }
}
```

Similar to Spring MVC you can also benefit from JSR-303/309 compliant validation by annotating

your arguments with `@Valid`.

```
@StreamListener(Processor.INPUT)
@SendTo(Processor.OUTPUT)
public VoteResult handle(@Valid Vote vote) {
    return votingService.record(vote);
}
```

In the above example the `Vote` object and its individual fields will be validated according to the rules set by you (e.g., `@NotBlank`, `@Min/@Max` etc.).



Spring Cloud Stream does NOT provide a default `org.springframework.validation.Validator` to avoid potential conflicts with validators provided by other frameworks that may be part of your application (e.g., MVC), therefore you may need to provide your own validator by configuring a bean of type `org.springframework.validation.Validator`.

Using `@StreamListener` for Content-based routing

Spring Cloud Stream supports dispatching messages to multiple handler methods annotated with `@StreamListener` based on conditions.

In order to be eligible to support conditional dispatching, a method must satisfy the follow conditions:

- It must not return a value.
- It must be an individual message handling method (reactive API methods are not supported).

The condition is specified by a SpEL expression in the `condition` argument of the annotation and is evaluated for each message. All the handlers that match the condition are invoked in the same thread, and no assumption must be made about the order in which the invocations take place.

In the following example of a `@StreamListener` with dispatching conditions, all the messages bearing a header `type` with the value `bogey` are dispatched to the `receiveBogey` method, and all the messages bearing a header `type` with the value `bacall` are dispatched to the `receiveBacall` method.


```

@EnableBinding(Sink.class)
@EnableAutoConfiguration
public static class TestPojoWithAnnotatedArguments {

    @StreamListener(target = Sink.INPUT, condition = "headers['type']=='bogey'")
    public void receiveBogey(@Payload BogeyPojo bogeyPojo) {
        // handle the message
    }

    @StreamListener(target = Sink.INPUT, condition = "headers['type']=='bacall'")
    public void receiveBacall(@Payload BacallPojo bacallPojo) {
        // handle the message
    }
}

```

Content Type Negotiation in the Context of **condition**

It is important to understand some of the mechanics behind content-based routing using the **condition** argument of **@StreamListener**, especially in the context of the type of the message as a whole. It may also help if you familiarize yourself with the [Content Type Negotiation](#) before you proceed.

Consider the following scenario:

```

@EnableBinding(Sink.class)
@EnableAutoConfiguration
public static class CatsAndDogs {

    @StreamListener(target = Sink.INPUT, condition =
"payload.class.simpleName=='Dog'")
    public void bark(Dog dog) {
        // handle the message
    }

    @StreamListener(target = Sink.INPUT, condition =
"payload.class.simpleName=='Cat'")
    public void purr(Cat cat) {
        // handle the message
    }
}

```

The preceding code is perfectly valid. It compiles and deploys without any issues, yet it never produces the result you expect.

That is because you are testing something that does not yet exist in a state you expect. That is because the payload of the message is not yet converted from the wire format (**byte[]**) to the desired type. In other words, it has not yet gone through the type conversion process described in the [Content Type Negotiation](#).

So, unless you use a SpEL expression that evaluates raw data (for example, the value of the first byte in the byte array), use message header-based expressions (such as `condition = "headers['type']=='dog'"`).



At the moment, dispatching through `@StreamListener` conditions is supported only for channel-based binders (not for reactive programming) support.

Using Polled Consumers

Overview

When using polled consumers, you poll the `PollableMessageSource` on demand. Consider the following example of a polled consumer:

```
public interface PolledConsumer {

    @Input
    PollableMessageSource destIn();

    @Output
    MessageChannel destOut();

}
```

Given the polled consumer in the preceding example, you might use it as follows:

```
@Bean
public ApplicationRunner poller(PollableMessageSource destIn, MessageChannel destOut)
{
    return args -> {
        while (someCondition()) {
            try {
                if (!destIn.poll(m -> {
                    String newPayload = ((String) m.getPayload()).toUpperCase();
                    destOut.send(new GenericMessage<>(newPayload));
                })) {
                    Thread.sleep(1000);
                }
            }
            catch (Exception e) {
                // handle failure
            }
        }
    };
}
```

A less manual and more Spring-like alternative would be to configure a scheduled task bean. For example,

```

@Scheduled(fixedDelay = 5_000)
public void poll() {
    System.out.println("Polling...");
    this.source.poll(m -> {
        System.out.println(m.getPayload());

    }, new ParameterizedTypeReference<Foo>() { });
}

```

The `PollableMessageSource.poll()` method takes a `MessageHandler` argument (often a lambda expression, as shown here). It returns `true` if the message was received and successfully processed.

As with message-driven consumers, if the `MessageHandler` throws an exception, messages are published to error channels, as discussed in [Error Handling](#).

Normally, the `poll()` method acknowledges the message when the `MessageHandler` exits. If the method exits abnormally, the message is rejected (not re-queued), but see [Handling Errors](#). You can override that behavior by taking responsibility for the acknowledgment, as shown in the following example:

```

@Bean
public ApplicationRunner poller(PollableMessageSource dest1In, MessageChannel
dest2Out) {
    return args -> {
        while (someCondition()) {
            if (!dest1In.poll(m -> {
                StaticMessageHeaderAccessor.getAcknowledgmentCallback(m).noAutoAck();
                // e.g. hand off to another thread which can perform the ack
                // or acknowledge(Status.REQUEUE)

            })) {
                Thread.sleep(1000);
            }
        }
    };
}

```



You must `ack` (or `nack`) the message at some point, to avoid resource leaks.



Some messaging systems (such as Apache Kafka) maintain a simple offset in a log. If a delivery fails and is re-queued with `StaticMessageHeaderAccessor.getAcknowledgmentCallback(m).acknowledge(Status.REQUEUE);`, any later successfully ack'd messages are redelivered.

There is also an overloaded `poll` method, for which the definition is as follows:

```
poll(MessageHandler handler, ParameterizedTypeReference<?> type)
```

The `type` is a conversion hint that allows the incoming message payload to be converted, as shown in the following example:

```
boolean result = pollableSource.poll(received -> {  
    Map<String, Foo> payload = (Map<String, Foo>) received.getPayload();  
    ...  
}, new ParameterizedTypeReference<Map<String, Foo>>() {});
```

Handling Errors

By default, an error channel is configured for the pollable source; if the callback throws an exception, an `ErrorMessage` is sent to the error channel (`<destination>.<group>.errors`); this error channel is also bridged to the global Spring Integration `errorChannel`.

You can subscribe to either error channel with a `@ServiceActivator` to handle errors; without a subscription, the error will simply be logged and the message will be acknowledged as successful. If the error channel service activator throws an exception, the message will be rejected (by default) and won't be redelivered. If the service activator throws a `RequeueCurrentMessageException`, the message will be requeued at the broker and will be again retrieved on a subsequent poll.

If the listener throws a `RequeueCurrentMessageException` directly, the message will be requeued, as discussed above, and will not be sent to the error channels.

20.9.4. Event Routing

Event Routing, in the context of Spring Cloud Stream, is the ability to either *a) route events to a particular even subscriber* or *b) route event produced by an event subscriber to a particular destination*. Here we'll refer to it as route 'TO' and route 'FROM'.

Routing TO Consumer

Routing can be achieved by relying on `RoutingFunction` available in Spring Cloud Function 3.0. All you need to do is enable it via `--spring.cloud.stream.function.routing.enabled=true` application property or provide `spring.cloud.function.routing-expression` property. Once enabled `RoutingFunction` will be bound to input destination receiving all the messages and route them to other functions based on the provided instruction.



For the purposes of binding the name of the routing destination is `functionRouter-in-0` (see `RoutingFunction.FUNCTION_NAME` and binding naming convention [Functional binding names](#)).

Instruction could be provided with individual messages as well as application properties.

Here are couple of samples:

Using message headers

```
@SpringBootApplication
public class SampleApplication {

    public static void main(String[] args) {
        SpringApplication.run(SampleApplication.class,
            "--spring.cloud.stream.function.routing.enabled=true");
    }

    @Bean
    public Consumer<String> even() {
        return value -> {
            System.out.println("EVEN: " + value);
        };
    }

    @Bean
    public Consumer<String> odd() {
        return value -> {
            System.out.println("ODD: " + value);
        };
    }
}
```

By sending message to the `functionRouter-in-0` destination exposed by the binder (i.e., rabbit, kafka), such message will be routed to the appropriate ('even' or 'odd') Consumer.

By default `RoutingFunction` will look for `spring.cloud.function.definition` or `spring.cloud.function.routing-expression` (for more dynamic scenarios with SpEL) header and if it is found its value will be treated as routing instruction.

For example, setting `spring.cloud.function.routing-expression` header to value `T(java.lang.System).currentTimeMillis() % 2 == 0 ? 'even' : 'odd'` will end up semi-randomly routing request to either `odd` or `even` functions. Also, for SpEL, the *root object* of the evaluation context is `Message` so you can do evaluation on individual headers (or message) as well `...routing-expression=headers['type']`

Using application properties

The `spring.cloud.function.routing-expression` and/or `spring.cloud.function.definition` can be passed as application properties (e.g., `spring.cloud.function.routing-expression=headers['type']`).

```

@SpringBootApplication
public class RoutingStreamApplication {

    public static void main(String[] args) {
        SpringApplication.run(RoutingStreamApplication.class,
            "--spring.cloud.function.routing-expression="
            + "T(java.lang.System).nanoTime() % 2 == 0 ? 'even' : 'odd'");
    }
    @Bean
    public Consumer<Integer> even() {
        return value -> System.out.println("EVEN: " + value);
    }

    @Bean
    public Consumer<Integer> odd() {
        return value -> System.out.println("ODD: " + value);
    }
}

```



Passing instructions via application properties is especially important for reactive functions since given that reactive function is only invoked once to pass the Publisher, so access to the individual items is limited.

Routing FROM Consumer

Aside from static destinations, Spring Cloud Stream lets applications send messages to dynamically bound destinations. This is useful, for example, when the target destination needs to be determined at runtime. Applications can do so in one of two ways

BinderAwareChannelResolver

The `BinderAwareChannelResolver` is a special bean registered automatically by the framework. You can autowire this bean into your application and use it to resolve output destination at runtime

The 'spring.cloud.stream.dynamicDestinations' property can be used for restricting the dynamic destination names to a known set (whitelisting). If this property is not set, any destination can be bound dynamically.

The following example demonstrates one of the common scenarios where REST controller uses a path variable to determine target destination:

```

@SpringBootApplication
@Controller
public class SourceWithDynamicDestination {

    @Autowired
    private BinderAwareChannelResolver resolver;

    @RequestMapping(value="/{target}")
    @ResponseStatus(HttpStatus.ACCEPTED)
    public void send(@RequestBody String body, @PathVariable("target") String target){
        resolver.resolveDestination(target).send(new GenericMessage<String>(body));
    }
}

```

Now consider what happens when we start the application on the default port (8080) and make the following requests with CURL:

```

curl -H "Content-Type: application/json" -X POST -d "customer-1"
http://localhost:8080/customers

curl -H "Content-Type: application/json" -X POST -d "order-1"
http://localhost:8080/orders

```

The destinations, 'customers' and 'orders', are created in the broker (in the exchange for Rabbit or in the topic for Kafka) with names of 'customers' and 'orders', and the data is published to the appropriate destinations.

spring.cloud.stream.sendto.destination

You can also delegate to the framework to dynamically resolve the output destination by specifying `spring.cloud.stream.sendto.destination` header set to the name of the destination to be resolved.

Consider the following example:

```

@SpringBootApplication
@Controller
public class SourceWithDynamicDestination {

    @Bean
    public Function<String, Message<String>> destinationAsPayload() {
        return value -> {
            return MessageBuilder.withPayload(value)
                .setHeader("spring.cloud.stream.sendto.destination", value).build();};
    }
}

```

Albeit trivial you can clearly see in this example, our output is a Message with

`spring.cloud.stream.sendto.destination` header set to the value of the input argument. The framework will consult this header and will attempt to create or discover destination with that name and send output to it.

If destination names are known in advance, you can configure the producer properties as with any other destination. Alternatively, if you register a `NewDestinationBindingCallback<>` bean, it is invoked just before the binding is created. The callback takes the generic type of the extended producer properties used by the binder. It has one method:

```
void configure(String destinationName, MessageChannel channel, ProducerProperties
producerProperties,
    T extendedProducerProperties);
```

The following example shows how to use the RabbitMQ binder:

```
@Bean
public NewDestinationBindingCallback<RabbitProducerProperties> dynamicConfigurer() {
    return (name, channel, props, extended) -> {
        props.setRequiredGroups("bindThisQueue");
        extended.setQueueNameGroupOnly(true);
        extended.setAutoBindDlq(true);
        extended.setDeadLetterQueueName("myDLQ");
    };
}
```



If you need to support dynamic destinations with multiple binder types, use `Object` for the generic type and cast the `extended` argument as needed.

Also, please see [Using StreamBridge](#) section to see how yet another option (`StreamBridge`) can be utilized for similar cases.

20.9.5. Error Handling

In this section we'll explain the general idea behind error handling mechanisms provided by the framework. We'll be using Rabbit binder as an example, since individual binders define different set of properties for certain supported mechanisms specific to underlying broker capabilities (such as Kafka binder).

Errors happen, and Spring Cloud Stream provides several flexible mechanisms to deal with them. Note that the techniques are dependent on binder implementation and the capability of the underlying messaging middleware.

Whenever there is an exception during message processing, the framework will make several attempts at re-trying the same message (3 by default). For that, the framework uses [Spring Retry](#) library (for imperative functions and standard message handlers) and `retryBackoff` capabilities of the reactive API (for reactive functions).

Whenever handler (function) throws an exception, it is propagated back to the binder, and the binder subsequently propagates the error back to the messaging system. Depending on the capabilities of the messaging system such system may *drop* the message, *re-queue* the message for re-processing or *send the failed message to DLQ*. Both Rabbit and Kafka support these concepts. However, other binders may not, so refer to your individual binder's documentation for details on supported error-handling options.

Drop Failed Messages

By default, if no additional system-level configuration is provided, the messaging system drops the failed message. While acceptable in some cases, for most cases, it is not, and we need some recovery mechanism to avoid message loss.

DLQ - Dead Letter Queue

Perhaps the most common mechanism, DLQ allows failed messages to be sent to a special destination: - *Dead Letter Queue*.

When configured, failed messages are sent to this destination for subsequent re-processing or auditing and reconciliation.

Consider the following example:

```
@SpringBootApplication
public class SimpleStreamApplication {

    public static void main(String[] args) throws Exception {
        SpringApplication.run(SimpleStreamApplication.class,
            "--spring.cloud.function.definition=uppercase",
            "--spring.cloud.stream.bindings.uppercase-in-0.destination=uppercase",
            "--spring.cloud.stream.bindings.uppercase-in-0.group=myGroup",
            "--spring.cloud.stream.rabbit.bindings.uppercase-in-0.consumer.auto-bind-
dlq=true"
        );
    }

    @Bean
    public Function<Person, Person> uppercase() {
        return personIn -> {
            throw new RuntimeException("intentional");
        };
    }
}
```

As a reminder, in this example `uppercase-in-0` segment of the property corresponds to the name of the input destination binding. The `consumer` segment indicates that it is a consumer property.



When using DLQ, at least `group` property must be provided for proper naming of the DLQ destination. However `group` often used together with `destination` property, as in our example.

Aside from some standard properties we also set the `auto-bind-dlq` to instruct the binder to create and configure DLQ destination for `uppercase-in-0` binding which corresponds to `uppercase` destination (see corresponding property), which results in an additional Rabbit queue named `uppercase.myGroup.dlq` (see Kafka documentation for Kafka specific DLQ properties).

Once configured, all failed messages are routed to this destination preserving the original message for further actions.

And you can see that the error message contains more information relevant to the original error, as follows:

```
. . . . .
x-exception-stacktrace: org.springframework.messaging.MessageHandlingException: nested
exception is
    org.springframework.messaging.MessagingException: has an error,
failedMessage=GenericMessage [payload=byte[15],
    headers={amqp_receivedDeliveryMode=NON_PERSISTENT,
amqp_receivedRoutingKey=input.hello, amqp_deliveryTag=1,
    deliveryAttempt=3, amqp_consumerQueue=input.hello, amqp_redelivered=false,
id=a15231e6-3f80-677b-5ad7-d4b1e61e486e,
    amqp_consumerTag=amq.ctag-skBFapilvtZhDsn0k3ZmQg, contentType=application/json,
timestamp=1522327846136}]
    at
org.spring...integ...han...MethodInvokingMessageProcessor.processMessage(MethodInvokin
gMessageProcessor.java:107)
    at. . . . .
Payload: blah
```

You can also facilitate immediate dispatch to DLQ (without re-tries) by setting `max-attempts` to '1'. For example,

```
--spring.cloud.stream.bindings.uppercase-in-0.consumer.max-attempts=1
```

Retry Template and retryBackoff

In this section we cover configuration properties relevant to configuration of retry capabilities. Given that we use two different mechanisms for imperative and reactive handlers (`RetryTemplate` and `retryBackoff`), properties that corresponds to both will be identified as such.

The `RetryTemplate` is part of the `Spring Retry` library. While it is out of scope of this document to cover all of the capabilities of the `RetryTemplate`, we will mention the following consumer properties that are specifically related to the `RetryTemplate`:

maxAttempts

The number of attempts to process the message.

Default: 3. - Applies to 'retryBackoff'

backOffInitialInterval

The backoff initial interval on retry.

Default 1000 milliseconds. - Applies to 'retryBackoff'

backOffMaxInterval

The maximum backoff interval.

Default 10000 milliseconds. - Applies to 'retryBackoff'

backOffMultiplier

The backoff multiplier.

Default 2.0.

defaultRetryable

Whether exceptions thrown by the listener that are not listed in the `retryableExceptions` are retryable.

Default: `true`.

retryableExceptions

A map of Throwable class names in the key and a boolean in the value. Specify those exceptions (and subclasses) that will or won't be retried. Also see `defaultRetriable`. Example: `spring.cloud.stream.bindings.input.consumer.retryable-exceptions.java.lang.IllegalStateException=false`.

Default: empty.

While the preceding settings are sufficient for majority of the customization requirements, they may not satisfy certain complex requirements at, which point you may want to provide your own instance of the `RetryTemplate`. To do so configure it as a bean in your application configuration. The application provided instance will override the one provided by the framework. Also, to avoid conflicts you must qualify the instance of the `RetryTemplate` you want to be used by the binder as `@StreamRetryTemplate`. For example,

```
@StreamRetryTemplate
public RetryTemplate myRetryTemplate() {
    return new RetryTemplate();
}
```

As you can see from the above example you don't need to annotate it with `@Bean` since `@StreamRetryTemplate` is a qualified `@Bean`.

If you need to be more precise with your `RetryTemplate`, you can specify the bean by name in your `ConsumerProperties` to associate the specific retry bean per binding.

```
spring.cloud.stream.bindings.<foo>.consumer.retry-template-name=<your-retry-template-bean-name>
```

20.10. Binders

Spring Cloud Stream provides a Binder abstraction for use in connecting to physical destinations at the external middleware. This section provides information about the main concepts behind the Binder SPI, its main components, and implementation-specific details.

20.10.1. Producers and Consumers

The following image shows the general relationship of producers and consumers:

[producers consumers] | <https://raw.githubusercontent.com/spring-cloud/spring-cloud->

Figure 9. Producers and Consumers

A producer is any component that sends messages to a binding destination. The binding destination can be bound to an external message broker with a `Binder` implementation for that broker. When invoking the `bindProducer()` method, the first parameter is the name of the destination within the broker, the second parameter is the instance of local destination to which the producer sends messages, and the third parameter contains properties (such as a partition key expression) to be used within the adapter that is created for that binding destination.

A consumer is any component that receives messages from the binding destination. As with a producer, the consumer can be bound to an external message broker. When invoking the `bindConsumer()` method, the first parameter is the destination name, and a second parameter provides the name of a logical group of consumers. Each group that is represented by consumer bindings for a given destination receives a copy of each message that a producer sends to that destination (that is, it follows normal publish-subscribe semantics). If there are multiple consumer instances bound with the same group name, then messages are load-balanced across those consumer instances so that each message sent by a producer is consumed by only a single consumer instance within each group (that is, it follows normal queueing semantics).

20.10.2. Binder SPI

The Binder SPI consists of a number of interfaces, out-of-the box utility classes, and discovery strategies that provide a pluggable mechanism for connecting to external middleware.

The key point of the SPI is the `Binder` interface, which is a strategy for connecting inputs and outputs to external middleware. The following listing shows the definition of the `Binder` interface:

```
public interface Binder<T, C extends ConsumerProperties, P extends ProducerProperties>
{
    Binding<T> bindConsumer(String bindingName, String group, T inboundBindTarget, C
consumerProperties);

    Binding<T> bindProducer(String bindingName, T outboundBindTarget, P
producerProperties);
}
```

The interface is parameterized, offering a number of extension points:

- Input and output bind targets.
- Extended consumer and producer properties, allowing specific Binder implementations to add supplemental properties that can be supported in a type-safe manner.

A typical binder implementation consists of the following:

- A class that implements the `Binder` interface;
- A Spring `@Configuration` class that creates a bean of type `Binder` along with the middleware connection infrastructure.

- A `META-INF/spring.binders` file found on the classpath containing one or more binder definitions, as shown in the following example:

```
kafka:\norg.springframework.cloud.stream.binder.kafka.config.KafkaBinderConfiguration
```



As it was mentioned earlier Binder abstraction is also one of the extension points of the framework. So if you can't find a suitable binder in the preceding list you can implement your own binder on top of Spring Cloud Stream. In the [How to create a Spring Cloud Stream Binder from scratch](#) post a community member documents in details, with an example, a set of steps necessary to implement a custom binder. The steps are also highlighted in the [Implementing Custom Binders](#) section.

20.10.3. Binder Detection

Spring Cloud Stream relies on implementations of the Binder SPI to perform the task of connecting (binding) user code to message brokers. Each Binder implementation typically connects to one type of messaging system.

Classpath Detection

By default, Spring Cloud Stream relies on Spring Boot's auto-configuration to configure the binding process. If a single Binder implementation is found on the classpath, Spring Cloud Stream automatically uses it. For example, a Spring Cloud Stream project that aims to bind only to RabbitMQ can add the following dependency:

```
<dependency>\n  <groupId>org.springframework.cloud</groupId>\n  <artifactId>spring-cloud-stream-binder-rabbit</artifactId>\n</dependency>
```

For the specific Maven coordinates of other binder dependencies, see the documentation of that binder implementation.

20.10.4. Multiple Binders on the Classpath

When multiple binders are present on the classpath, the application must indicate which binder is to be used for each destination binding. Each binder configuration contains a `META-INF/spring.binders` file, which is a simple properties file, as shown in the following example:

```
rabbit:\norg.springframework.cloud.stream.binder.rabbit.config.RabbitServiceAutoConfiguration
```

Similar files exist for the other provided binder implementations (such as Kafka), and custom

binder implementations are expected to provide them as well. The key represents an identifying name for the binder implementation, whereas the value is a comma-separated list of configuration classes that each contain one and only one bean definition of type `org.springframework.cloud.stream.binder.Binder`.

Binder selection can either be performed globally, using the `spring.cloud.stream.defaultBinder` property (for example, `spring.cloud.stream.defaultBinder=rabbit`) or individually, by configuring the binder on each binding. For instance, a processor application (that has bindings named `input` and `output` for read and write respectively) that reads from Kafka and writes to RabbitMQ can specify the following configuration:

```
spring.cloud.stream.bindings.input.binder=kafka
spring.cloud.stream.bindings.output.binder=rabbit
```

20.10.5. Connecting to Multiple Systems

By default, binders share the application's Spring Boot auto-configuration, so that one instance of each binder found on the classpath is created. If your application should connect to more than one broker of the same type, you can specify multiple binder configurations, each with different environment settings.



Turning on explicit binder configuration disables the default binder configuration process altogether. If you do so, all binders in use must be included in the configuration. Frameworks that intend to use Spring Cloud Stream transparently may create binder configurations that can be referenced by name, but they do not affect the default binder configuration. In order to do so, a binder configuration may have its `defaultCandidate` flag set to `false` (for example, `spring.cloud.stream.binders.<configurationName>.defaultCandidate=false`). This denotes a configuration that exists independently of the default binder configuration process.

The following example shows a typical configuration for a processor application that connects to two RabbitMQ broker instances:

```
spring:
  cloud:
    stream:
      bindings:
        input:
          destination: thing1
          binder: rabbit1
        output:
          destination: thing2
          binder: rabbit2
      binders:
        rabbit1:
          type: rabbit
          environment:
            spring:
              rabbitmq:
                host: <host1>
        rabbit2:
          type: rabbit
          environment:
            spring:
              rabbitmq:
                host: <host2>
```



The `environment` property of the particular binder can also be used for any Spring Boot property, including this `spring.main.sources` which can be useful for adding additional configurations for the particular binders, e.g. overriding auto-configured beans.

For example;

```
environment:
  spring:
    main:
      sources: com.acme.config.MyCustomBinderConfiguration
```

To activate a specific profile for the particular binder environment, you should use a `spring.profiles.active` property:

```
environment:
  spring:
    profiles:
      active: myBinderProfile
```


20.10.6. Binding visualization and control

Since version 2.0, Spring Cloud Stream supports visualization and control of the Bindings through Actuator endpoints.

Starting with version 2.0 actuator and web are optional, you must first add one of the web dependencies as well as add the actuator dependency manually. The following example shows how to add the dependency for the Web framework:

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-web</artifactId>
</dependency>
```

The following example shows how to add the dependency for the WebFlux framework:

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-webflux</artifactId>
</dependency>
```

You can add the Actuator dependency as follows:

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
```



To run Spring Cloud Stream 2.0 apps in Cloud Foundry, you must add `spring-boot-starter-web` and `spring-boot-starter-actuator` to the classpath. Otherwise, the application will not start due to health check failures.

You must also enable the `bindings` actuator endpoints by setting the following property: `--management.endpoints.web.exposure.include=bindings`.

Once those prerequisites are satisfied, you should see the following in the logs when application start:

```
: Mapped "{[/actuator/bindings/{name}],methods=[POST]. . .
: Mapped "{[/actuator/bindings],methods=[GET]. . .
: Mapped "{[/actuator/bindings/{name}],methods=[GET]. . .
```

To visualize the current bindings, access the following URL: `<host>:<port>/actuator/bindings`

Alternative, to see a single binding, access one of the URLs similar to the following: `<code><a href="http://<host>:<port>/actuator/bindings/<bindingName>"`

```
class="bare">&lt;host&gt;;&lt;port&gt;/actuator/bindings/&lt;bindingName&gt;</a>;</code>
```

You can also stop, start, pause, and resume individual bindings by posting to the same URL while providing a `state` argument as JSON, as shown in the following examples:

```
curl -d '{"state":"STOPPED"}' -H "Content-Type: application/json" -X POST
http://<host>:<port>/actuator/bindings/myBindingName
curl -d '{"state":"STARTED"}' -H "Content-Type: application/json" -X POST
http://<host>:<port>/actuator/bindings/myBindingName
curl -d '{"state":"PAUSED"}' -H "Content-Type: application/json" -X POST
http://<host>:<port>/actuator/bindings/myBindingName
curl -d '{"state":"RESUMED"}' -H "Content-Type: application/json" -X POST
http://<host>:<port>/actuator/bindings/myBindingName
```



`PAUSED` and `RESUMED` work only when the corresponding binder and its underlying technology supports it. Otherwise, you see the warning message in the logs. Currently, only Kafka binder supports the `PAUSED` and `RESUMED` states.

20.10.7. Binder Configuration Properties

The following properties are available when customizing binder configurations. These properties exposed via `org.springframework.cloud.stream.config.BinderProperties`

They must be prefixed with `spring.cloud.stream.binders.<configurationName>`.

type

The binder type. It typically references one of the binders found on the classpath — in particular, a key in a `META-INF/spring.binders` file.

By default, it has the same value as the configuration name.

inheritEnvironment

Whether the configuration inherits the environment of the application itself.

Default: `true`.

environment

Root for a set of properties that can be used to customize the environment of the binder. When this property is set, the context in which the binder is being created is not a child of the application context. This setting allows for complete separation between the binder components and the application components.

Default: `empty`.

defaultCandidate

Whether the binder configuration is a candidate for being considered a default binder or can be used only when explicitly referenced. This setting allows adding binder configurations without interfering with the default processing.

Default: `true`.

20.10.8. Implementing Custom Binders

In order to implement a custom `Binder`, all you need is to:

- Add the required dependencies
- Provide a `ProvisioningProvider` implementation
- Provide a `MessageProducer` implementation
- Provide a `MessageHandler` implementation
- Provide a `Binder` implementation
- Create a `Binder Configuration`
- Define your binder in `META-INF/spring.binders`

Add the required dependencies

Add the `spring-cloud-stream` dependency to your project (*eg. for Maven*):

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-stream</artifactId>
  <version>${spring.cloud.stream.version}</version>
</dependency>
```

Provide a `ProvisioningProvider` implementation

The `ProvisioningProvider` is responsible for the provisioning of consumer and producer destinations, and is required to convert the logical destinations included in the `application.yml` or `application.properties` file in physical destination references.

Below an example of `ProvisioningProvider` implementation that simply trims the destinations provided via input/output bindings configuration:

```

public class FileMessageBinderProvisioner implements
ProvisioningProvider<ConsumerProperties, ProducerProperties> {

    @Override
    public ProducerDestination provisionProducerDestination(
        final String name,
        final ProducerProperties properties) {

        return new FileMessageDestination(name);
    }

    @Override
    public ConsumerDestination provisionConsumerDestination(
        final String name,
        final String group,
        final ConsumerProperties properties) {

        return new FileMessageDestination(name);
    }

    private class FileMessageDestination implements ProducerDestination,
ConsumerDestination {

        private final String destination;

        private FileMessageDestination(final String destination) {
            this.destination = destination;
        }

        @Override
        public String getName() {
            return destination.trim();
        }

        @Override
        public String getNameForPartition(int partition) {
            throw new UnsupportedOperationException("Partitioning is not implemented
for file messaging.");
        }

    }
}

```

Provide a MessageProducer implementation

The **MessageProducer** is responsible for consuming events and handling them as messages to the client application that is configured to consume such events.

Here is an example of MessageProducer implementation that extends the **MessageProducerSupport**

abstraction in order to poll on a file that matches the trimmed destination name and is located in the project path, while also archiving read messages and discarding consequent identical messages:

```
public class FileMessageProducer extends MessageProducerSupport {

    public static final String ARCHIVE = "archive.txt";
    private final ConsumerDestination destination;
    private String previousPayload;

    public FileMessageProducer(ConsumerDestination destination) {
        this.destination = destination;
    }

    @Override
    public void doStart() {
        receive();
    }

    private void receive() {
        ScheduledExecutorService executorService =
        Executors.newScheduledThreadPool(1);

        executorService.scheduleWithFixedDelay(() -> {
            String payload = getPayload();

            if(payload != null) {
                Message<String> receivedMessage =
                MessageBuilder.withPayload(payload).build();
                archiveMessage(payload);
                sendMessage(receivedMessage);
            }

        }, 0, 50, MILLISECONDS);
    }

    private String getPayload() {
        try {
            List<String> allLines =
            Files.readAllLines(Paths.get(destination.getName()));
            String currentPayload = allLines.get(allLines.size() - 1);

            if(!currentPayload.equals(previousPayload)) {
                previousPayload = currentPayload;
                return currentPayload;
            }
        } catch (IOException e) {
            throw new RuntimeException(e);
        }

        return null;
    }
}
```

```

    }

    private void archiveMessage(String payload) {
        try {
            Files.write(Paths.get(ARCHIVE), (payload + "\n").getBytes(), CREATE,
APPEND);
        } catch (IOException e) {
            throw new RuntimeException(e);
        }
    }
}

```



When implementing a custom binder, this step is not strictly mandatory as you could always resort to using an already existing `MessageProducer` implementation!

Provide a `MessageHandler` implementation

The `MessageHandler` provides the logic required to produce an event.

Here is an example of `MessageHandler` implementation:

```

public class FileMessageHandler implements MessageHandler{

    @Override
    public void handleMessage(Message<?> message) throws MessagingException {
        //write message to file
    }

}

```



When implementing a custom binder, this step is not strictly mandatory as you could always resort to using an already existing `MessageHandler` implementation!

Provide a `Binder` implementation

You are now able to provide your own implementation of the `Binder` abstraction. This can be easily done by:

- extending the `AbstractMessageChannelBinder` class
- specifying your `ProvisioningProvider` as a generic argument of the `AbstractMessageChannelBinder`
- overriding the `createProducerMessageHandler` and `createConsumerEndpoint` methods

eg.:

```

public class FileMessageBinder extends
AbstractMessageChannelBinder<ConsumerProperties, ProducerProperties,
FileMessageBinderProvisioner> {

    public FileMessageBinder(
        String[] headersToEmbed,
        FileMessageBinderProvisioner provisioningProvider) {

        super(headersToEmbed, provisioningProvider);
    }

    @Override
    protected MessageHandler createProducerMessageHandler(
        final ProducerDestination destination,
        final ProducerProperties producerProperties,
        final MessageChannel errorChannel) throws Exception {

        return message -> {
            String fileName = destination.getName();
            String payload = new String((byte[])message.getPayload()) + "\n";

            try {
                Files.write(Paths.get(fileName), payload.getBytes(), CREATE, APPEND);
            } catch (IOException e) {
                throw new RuntimeException(e);
            }
        };
    }

    @Override
    protected MessageProducer createConsumerEndpoint(
        final ConsumerDestination destination,
        final String group,
        final ConsumerProperties properties) throws Exception {

        return new FileMessageProducer(destination);
    }
}

```

Create a Binder Configuration

It is strictly required that you create a Spring Configuration to initialize the bean for your binder implementation (*and all other beans that you might need*):

```

@Configuration
public class FileMessageBinderConfiguration {

    @Bean
    @ConditionalOnMissingBean
    public FileMessageBinderProvisioner fileMessageBinderProvisioner() {
        return new FileMessageBinderProvisioner();
    }

    @Bean
    @ConditionalOnMissingBean
    public FileMessageBinder fileMessageBinder(FileMessageBinderProvisioner
fileMessageBinderProvisioner) {
        return new FileMessageBinder(null, fileMessageBinderProvisioner);
    }

}

```

Define your binder in META-INF/spring.binders

Finally, you must define your binder in a `META-INF/spring.binders` file on the classpath, specifying both the name of the binder and the full qualified name of your Binder Configuration class:

```

myFileBinder:\
com.example.springcloudstreamcustombinder.config.FileMessageBinderConfiguration

```

20.11. Configuration Options

Spring Cloud Stream supports general configuration options as well as configuration for bindings and binders. Some binders let additional binding properties support middleware-specific features.

Configuration options can be provided to Spring Cloud Stream applications through any mechanism supported by Spring Boot. This includes application arguments, environment variables, and YAML or `.properties` files.

20.11.1. Binding Service Properties

These properties are exposed via `org.springframework.cloud.stream.config.BindingServiceProperties`

spring.cloud.stream.instanceCount

The number of deployed instances of an application. Must be set for partitioning on the producer side. Must be set on the consumer side when using RabbitMQ and with Kafka if `autoRebalanceEnabled=false`.

Default: 1.

spring.cloud.stream.instanceIndex

The instance index of the application: A number from 0 to `instanceCount - 1`. Used for partitioning with RabbitMQ and with Kafka if `autoRebalanceEnabled=false`. Automatically set in Cloud Foundry to match the application's instance index.

spring.cloud.stream.dynamicDestinations

A list of destinations that can be bound dynamically (for example, in a dynamic routing scenario). If set, only listed destinations can be bound.

Default: empty (letting any destination be bound).

spring.cloud.stream.defaultBinder

The default binder to use, if multiple binders are configured. See [Multiple Binders on the Classpath](#).

Default: empty.

spring.cloud.stream.overrideCloudConnectors

This property is only applicable when the `cloud` profile is active and Spring Cloud Connectors are provided with the application. If the property is `false` (the default), the binder detects a suitable bound service (for example, a RabbitMQ service bound in Cloud Foundry for the RabbitMQ binder) and uses it for creating connections (usually through Spring Cloud Connectors). When set to `true`, this property instructs binders to completely ignore the bound services and rely on Spring Boot properties (for example, relying on the `spring.rabbitmq.*` properties provided in the environment for the RabbitMQ binder). The typical usage of this property is to be nested in a customized environment [when connecting to multiple systems](#).

Default: `false`.

spring.cloud.stream.bindingRetryInterval

The interval (in seconds) between retrying binding creation when, for example, the binder does not support late binding and the broker (for example, Apache Kafka) is down. Set it to zero to treat such conditions as fatal, preventing the application from starting.

Default: `30`

20.11.2. Binding Properties

Binding properties are supplied by using the format of `spring.cloud.stream.bindings.<bindingName>.<property>=<value>`. The `<bindingName>` represents the name of the binding being configured.

For example, for the following function

```
@Bean
public Function<String, String> uppercase() {
    return v -> v.toUpperCase();
}
```

there are two bindings named `uppercase-in-0` for input and `uppercase-out-0` for output. See [Binding and Binding names](#) for more details.

To avoid repetition, Spring Cloud Stream supports setting values for all bindings, in the format of `spring.cloud.stream.default.<property>=<value>` and `spring.cloud.stream.default.<producer|consumer>.<property>=<value>` for common binding properties.

When it comes to avoiding repetitions for extended binding properties, this format should be used - `spring.cloud.stream.<binder-type>.default.<producer|consumer>.<property>=<value>`.

Common Binding Properties

These properties are exposed via `org.springframework.cloud.stream.config.BindingProperties`

The following binding properties are available for both input and output bindings and must be prefixed with `spring.cloud.stream.bindings.<bindingName>`. (for example, `spring.cloud.stream.bindings.uppercase-in-0.destination=ticktock`).

Default values can be set by using the `spring.cloud.stream.default` prefix (for example `spring.cloud.stream.default.contentType=application/json``).

destination

The target destination of a binding on the bound middleware (for example, the RabbitMQ exchange or Kafka topic). If binding represents a consumer binding (input), it could be bound to multiple destinations, and the destination names can be specified as comma-separated `String` values. If not, the actual binding name is used instead. The default value of this property cannot be overridden.

group

The consumer group of the binding. Applies only to inbound bindings. See [Consumer Groups](#).

Default: `null` (indicating an anonymous consumer).

contentType

The content type of this binding. See [Content Type Negotiation](#).

Default: `application/json`.

binder

The binder used by this binding. See [Multiple Binders on the Classpath](#) for details.

Default: `null` (the default binder is used, if it exists).

Consumer Properties

These properties are exposed via `org.springframework.cloud.stream.binder.ConsumerProperties`

The following binding properties are available for input bindings only and must be prefixed with `spring.cloud.stream.bindings.<bindingName>.consumer`. (for example,

`spring.cloud.stream.bindings.input.consumer.concurrency=3`).

Default values can be set by using the `spring.cloud.stream.default.consumer` prefix (for example, `spring.cloud.stream.default.consumer.headerMode=none`).

autoStartup

Signals if this consumer needs to be started automatically

Default: `true`.

concurrency

The concurrency of the inbound consumer.

Default: `1`.

partitioned

Whether the consumer receives data from a partitioned producer.

Default: `false`.

headerMode

When set to `none`, disables header parsing on input. Effective only for messaging middleware that does not support message headers natively and requires header embedding. This option is useful when consuming data from non-Spring Cloud Stream applications when native headers are not supported. When set to `headers`, it uses the middleware's native header mechanism. When set to `embeddedHeaders`, it embeds headers into the message payload.

Default: depends on the binder implementation.

maxAttempts

If processing fails, the number of attempts to process the message (including the first). Set to `1` to disable retry.

Default: `3`.

backOffInitialInterval

The backoff initial interval on retry.

Default: `1000`.

backOffMaxInterval

The maximum backoff interval.

Default: `10000`.

backOffMultiplier

The backoff multiplier.

Default: `2.0`.

defaultRetryable

Whether exceptions thrown by the listener that are not listed in the `retryableExceptions` are retryable.

Default: `true`.

instanceCount

When set to a value greater than equal to zero, it allows customizing the instance count of this consumer (if different from `spring.cloud.stream.instanceCount`). When set to a negative value, it defaults to `spring.cloud.stream.instanceCount`. See [Instance Index and Instance Count](#) for more information.

Default: `-1`.

instanceIndex

When set to a value greater than equal to zero, it allows customizing the instance index of this consumer (if different from `spring.cloud.stream.instanceIndex`). When set to a negative value, it defaults to `spring.cloud.stream.instanceIndex`. Ignored if `instanceIndexList` is provided. See [Instance Index and Instance Count](#) for more information.

Default: `-1`.

instanceIndexList

Used with binders that do not support native partitioning (such as RabbitMQ); allows an application instance to consume from more than one partition.

Default: empty.

retryableExceptions

A map of Throwable class names in the key and a boolean in the value. Specify those exceptions (and subclasses) that will or won't be retried. Also see `defaultRetriable`. Example: `spring.cloud.stream.bindings.input.consumer.retryable-exceptions.java.lang.IllegalStateException=false`.

Default: empty.

useNativeDecoding

When set to `true`, the inbound message is deserialized directly by the client library, which must be configured correspondingly (for example, setting an appropriate Kafka producer value deserializer). When this configuration is being used, the inbound message unmarshalling is not based on the `contentType` of the binding. When native decoding is used, it is the responsibility of the producer to use an appropriate encoder (for example, the Kafka producer value serializer) to serialize the outbound message. Also, when native encoding and decoding is used, the `headerMode=embeddedHeaders` property is ignored and headers are not embedded in the message. See the producer property `useNativeEncoding`.

Default: `false`.

multiplex

When set to true, the underlying binder will natively multiplex destinations on the same input binding.

Default: `false`.

Advanced Consumer Configuration

For advanced configuration of the underlying message listener container for message-driven consumers, add a single `ListenerContainerCustomizer` bean to the application context. It will be invoked after the above properties have been applied and can be used to set additional properties. Similarly, for polled consumers, add a `MessageSourceCustomizer` bean.

The following is an example for the RabbitMQ binder:

```
@Bean
public ListenerContainerCustomizer<AbstractMessageListenerContainer>
containerCustomizer() {
    return (container, dest, group) -> container.setAdviceChain(advice1, advice2);
}

@Bean
public MessageSourceCustomizer<AmqpMessageSource> sourceCustomizer() {
    return (source, dest, group) ->
    source.setPropertiesConverter(customPropertiesConverter);
}
```

Producer Properties

These properties are exposed via `org.springframework.cloud.stream.binder.ProducerProperties`

The following binding properties are available for output bindings only and must be prefixed with `spring.cloud.stream.bindings.<bindingName>.producer.` (for example, `spring.cloud.stream.bindings.func-out-0.producer.partitionKeyExpression=payload.id`).

Default values can be set by using the prefix `spring.cloud.stream.default.producer` (for example, `spring.cloud.stream.default.producer.partitionKeyExpression=payload.id`).

autoStartup

Signals if this consumer needs to be started automatically

Default: `true`.

partitionKeyExpression

A SpEL expression that determines how to partition outbound data. If set, outbound data on this binding is partitioned. `partitionCount` must be set to a value greater than 1 to be effective. See [Partitioning Support](#).

Default: null.

partitionKeyExtractorName

The name of the bean that implements `PartitionKeyExtractorStrategy`. Used to extract a key used to compute the partition id (see 'partitionSelector*'). Mutually exclusive with 'partitionKeyExpression'.

Default: null.

partitionSelectorName

The name of the bean that implements `PartitionSelectorStrategy`. Used to determine partition id based on partition key (see 'partitionKeyExtractor*'). Mutually exclusive with 'partitionSelectorExpression'.

Default: null.

partitionSelectorExpression

A SpEL expression for customizing partition selection. If neither is set, the partition is selected as the `hashCode(key) % partitionCount`, where `key` is computed through either `partitionKeyExpression`.

Default: `null`.

partitionCount

The number of target partitions for the data, if partitioning is enabled. Must be set to a value greater than 1 if the producer is partitioned. On Kafka, it is interpreted as a hint. The larger of this and the partition count of the target topic is used instead.

Default: `1`.

requiredGroups

A comma-separated list of groups to which the producer must ensure message delivery even if they start after it has been created (for example, by pre-creating durable queues in RabbitMQ).

headerMode

When set to `none`, it disables header embedding on output. It is effective only for messaging middleware that does not support message headers natively and requires header embedding. This option is useful when producing data for non-Spring Cloud Stream applications when native headers are not supported. When set to `headers`, it uses the middleware's native header mechanism. When set to `embeddedHeaders`, it embeds headers into the message payload.

Default: Depends on the binder implementation.

useNativeEncoding

When set to `true`, the outbound message is serialized directly by the client library, which must be configured correspondingly (for example, setting an appropriate Kafka producer value serializer). When this configuration is being used, the outbound message marshalling is not based on the `contentType` of the binding. When native encoding is used, it is the responsibility of the consumer to use an appropriate decoder (for example, the Kafka consumer value deserializer) to deserialize the inbound message. Also, when native encoding and decoding is used, the `headerMode=embeddedHeaders` property is ignored and headers are not embedded in the

message. See the consumer property `useNativeDecoding`.

Default: `false`.

useNativeEncoding

When set to `true`, the outbound message is serialized directly by the client library, which must be configured correspondingly (for example, setting an appropriate Kafka producer value serializer). When this configuration is being used, the outbound message marshalling is not based on the `contentType` of the binding. When native encoding is used, it is the responsibility of the consumer to use an appropriate decoder (for example, the Kafka consumer value deserializer) to deserialize the inbound message. Also, when native encoding and decoding is used, the `headerMode=embeddedHeaders` property is ignored and headers are not embedded in the message. See the consumer property `useNativeDecoding`.

Default: `false`.

errorChannelEnabled

When set to true, if the binder supports asynchronous send results, send failures are sent to an error channel for the destination. See Error Handling for more information.

Default: `false`.

20.12. Content Type Negotiation

Data transformation is one of the core features of any message-driven microservice architecture. Given that, in Spring Cloud Stream, such data is represented as a Spring `Message`, a message may have to be transformed to a desired shape or size before reaching its destination. This is required for two reasons:

1. To convert the contents of the incoming message to match the signature of the application-provided handler.
2. To convert the contents of the outgoing message to the wire format.

The wire format is typically `byte[]` (that is true for the Kafka and Rabbit binders), but it is governed by the binder implementation.

In Spring Cloud Stream, message transformation is accomplished with an `org.springframework.messaging.converter.MessageConverter`.



As a supplement to the details to follow, you may also want to read the following [blog post](#).

20.12.1. Mechanics

To better understand the mechanics and the necessity behind content-type negotiation, we take a look at a very simple use case by using the following message handler as an example:

```
public Function<Person, Person> personFunction {..}
```



For simplicity, we assume that this is the only handler function in the application (we assume there is no internal pipeline).

The handler shown in the preceding example expects a `Person` object as an argument and produces a `String` type as an output. In order for the framework to succeed in passing the incoming `Message` as an argument to this handler, it has to somehow transform the payload of the `Message` type from the wire format to a `Person` type. In other words, the framework must locate and apply the appropriate `MessageConverter`. To accomplish that, the framework needs some instructions from the user. One of these instructions is already provided by the signature of the handler method itself (`Person` type). Consequently, in theory, that should be (and, in some cases, is) enough. However, for the majority of use cases, in order to select the appropriate `MessageConverter`, the framework needs an additional piece of information. That missing piece is `contentType`.

Spring Cloud Stream provides three mechanisms to define `contentType` (in order of precedence):

1. **HEADER:** The `contentType` can be communicated through the `Message` itself. By providing a `contentType` header, you declare the content type to use to locate and apply the appropriate `MessageConverter`.
2. **BINDING:** The `contentType` can be set per destination binding by setting the `spring.cloud.stream.bindings.input.content-type` property.



The `input` segment in the property name corresponds to the actual name of the destination (which is “input” in our case). This approach lets you declare, on a per-binding basis, the content type to use to locate and apply the appropriate `MessageConverter`.

3. **DEFAULT:** If `contentType` is not present in the `Message` header or the binding, the default `application/json` content type is used to locate and apply the appropriate `MessageConverter`.

As mentioned earlier, the preceding list also demonstrates the order of precedence in case of a tie. For example, a header-provided content type takes precedence over any other content type. The same applies for a content type set on a per-binding basis, which essentially lets you override the default content type. However, it also provides a sensible default (which was determined from community feedback).

Another reason for making `application/json` the default stems from the interoperability requirements driven by distributed microservices architectures, where producer and consumer not only run in different JVMs but can also run on different non-JVM platforms.

When the non-void handler method returns, if the return value is already a `Message`, that `Message` becomes the payload. However, when the return value is not a `Message`, the new `Message` is constructed with the return value as the payload while inheriting headers from the input `Message` minus the headers defined or filtered by `SpringIntegrationProperties.messageHandlerNotPropagatedHeaders`. By default, there is only one header set there: `contentType`. This means that the new `Message` does not have `contentType` header

set, thus ensuring that the `contentType` can evolve. You can always opt out of returning a `Message` from the handler method where you can inject any header you wish.

If there is an internal pipeline, the `Message` is sent to the next handler by going through the same process of conversion. However, if there is no internal pipeline or you have reached the end of it, the `Message` is sent back to the output destination.

Content Type versus Argument Type

As mentioned earlier, for the framework to select the appropriate `MessageConverter`, it requires argument type and, optionally, content type information. The logic for selecting the appropriate `MessageConverter` resides with the argument resolvers (`HandlerMethodArgumentResolvers`), which trigger right before the invocation of the user-defined handler method (which is when the actual argument type is known to the framework). If the argument type does not match the type of the current payload, the framework delegates to the stack of the pre-configured `MessageConverters` to see if any one of them can convert the payload. As you can see, the `Object fromMessage(Message<?> message, Class<?> targetClass)`; operation of the `MessageConverter` takes `targetClass` as one of its arguments. The framework also ensures that the provided `Message` always contains a `contentType` header. When no `contentType` header was already present, it injects either the per-binding `contentType` header or the default `contentType` header. The combination of `contentType` argument type is the mechanism by which framework determines if message can be converted to a target type. If no appropriate `MessageConverter` is found, an exception is thrown, which you can handle by adding a custom `MessageConverter` (see [User-defined Message Converters](#)).

But what if the payload type matches the target type declared by the handler method? In this case, there is nothing to convert, and the payload is passed unmodified. While this sounds pretty straightforward and logical, keep in mind handler methods that take a `Message<?>` or `Object` as an argument. By declaring the target type to be `Object` (which is an `instanceof` everything in Java), you essentially forfeit the conversion process.



Do not expect `Message` to be converted into some other type based only on the `contentType`. Remember that the `contentType` is complementary to the target type. If you wish, you can provide a hint, which `MessageConverter` may or may not take into consideration.

Message Converters

`MessageConverters` define two methods:

```
Object fromMessage(Message<?> message, Class<?> targetClass);  
  
Message<?> toMessage(Object payload, @Nullable MessageHeaders headers);
```

It is important to understand the contract of these methods and their usage, specifically in the context of Spring Cloud Stream.

The `fromMessage` method converts an incoming `Message` to an argument type. The payload of the `Message` could be any type, and it is up to the actual implementation of the `MessageConverter` to

support multiple types. For example, some JSON converter may support the payload type as `byte[]`, `String`, and others. This is important when the application contains an internal pipeline (that is, `input → handler1 → handler2 → . . . → output`) and the output of the upstream handler results in a `Message` which may not be in the initial wire format.

However, the `toMessage` method has a more strict contract and must always convert `Message` to the wire format: `byte[]`.

So, for all intents and purposes (and especially when implementing your own converter) you regard the two methods as having the following signatures:

```
Object fromMessage(Message<?> message, Class<?> targetClass);

Message<byte[]> toMessage(Object payload, @Nullable MessageHeaders headers);
```

20.12.2. Provided MessageConverters

As mentioned earlier, the framework already provides a stack of `MessageConverters` to handle most common use cases. The following list describes the provided `MessageConverters`, in order of precedence (the first `MessageConverter` that works is used):

1. `ApplicationJsonMessageMarshallingConverter`: Variation of the `org.springframework.messaging.converter.MappingJackson2MessageConverter`. Supports conversion of the payload of the `Message` to/from POJO for cases when `contentType` is `application/json` (DEFAULT).
2. `ByteArrayMessageConverter`: Supports conversion of the payload of the `Message` from `byte[]` to `byte[]` for cases when `contentType` is `application/octet-stream`. It is essentially a pass through and exists primarily for backward compatibility.
3. `ObjectStringMessageConverter`: Supports conversion of any type to a `String` when `contentType` is `text/plain`. It invokes `Object`'s `toString()` method or, if the payload is `byte[]`, a new `String(byte[])`.
4. `JsonUnmarshallingConverter`: Similar to the `ApplicationJsonMessageMarshallingConverter`. It supports conversion of any type when `contentType` is `application/x-java-object`. It expects the actual type information to be embedded in the `contentType` as an attribute (for example, `application/x-java-object;type=foo.bar.Cat`).

When no appropriate converter is found, the framework throws an exception. When that happens, you should check your code and configuration and ensure you did not miss anything (that is, ensure that you provided a `contentType` by using a binding or a header). However, most likely, you found some uncommon case (such as a custom `contentType` perhaps) and the current stack of provided `MessageConverters` does not know how to convert. If that is the case, you can add custom `MessageConverter`. See [User-defined Message Converters](#).

20.12.3. User-defined Message Converters

Spring Cloud Stream exposes a mechanism to define and register additional `MessageConverters`. To use it, implement `org.springframework.messaging.converter.MessageConverter`, configure it as a

`@Bean`. It is then appended to the existing stack of `MessageConverter``s.



It is important to understand that custom `MessageConverter` implementations are added to the head of the existing stack. Consequently, custom `MessageConverter` implementations take precedence over the existing ones, which lets you override as well as add to the existing converters.

The following example shows how to create a message converter bean to support a new content type called `application/bar`:

```
@SpringBootApplication
public static class SinkApplication {

    ...

    @Bean
    public MessageConverter customMessageConverter() {
        return new MyCustomMessageConverter();
    }
}

public class MyCustomMessageConverter extends AbstractMessageConverter {

    public MyCustomMessageConverter() {
        super(new MediaType("application", "bar"));
    }

    @Override
    protected boolean supports(Class<?> clazz) {
        return (Bar.class.equals(clazz));
    }

    @Override
    protected Object convertFromInternal(Message<?> message, Class<?> targetClass,
    Object conversionHint) {
        Object payload = message.getPayload();
        return (payload instanceof Bar ? payload : new Bar((byte[]) payload));
    }
}
```

Spring Cloud Stream also provides support for Avro-based converters and schema evolution. See [\[schema-evolution\]](#) for details.

[== Inter-Application Communication

Spring Cloud Stream enables communication between applications. Inter-application communication is a complex issue spanning several concerns, as described in the following topics:

- [Connecting Multiple Application Instances](#)

- [Instance Index and Instance Count](#)
- [Partitioning](#)

20.12.4. Connecting Multiple Application Instances

While Spring Cloud Stream makes it easy for individual Spring Boot applications to connect to messaging systems, the typical scenario for Spring Cloud Stream is the creation of multi-application pipelines, where microservice applications send data to each other. You can achieve this scenario by correlating the input and output destinations of “adjacent” applications.

Suppose a design calls for the Time Source application to send data to the Log Sink application. You could use a common destination named `ticktock` for bindings within both applications.

Time Source (that has the binding named `output`) would set the following property:

```
spring.cloud.stream.bindings.output.destination=ticktock
```

Log Sink (that has the binding named `input`) would set the following property:

```
spring.cloud.stream.bindings.input.destination=ticktock
```

20.12.5. Instance Index and Instance Count

When scaling up Spring Cloud Stream applications, each instance can receive information about how many other instances of the same application exist and what its own instance index is. Spring Cloud Stream does this through the `spring.cloud.stream.instanceCount` and `spring.cloud.stream.instanceIndex` properties. For example, if there are three instances of a HDFS sink application, all three instances have `spring.cloud.stream.instanceCount` set to `3`, and the individual applications have `spring.cloud.stream.instanceIndex` set to `0`, `1`, and `2`, respectively.

When Spring Cloud Stream applications are deployed through Spring Cloud Data Flow, these properties are configured automatically; when Spring Cloud Stream applications are launched independently, these properties must be set correctly. By default, `spring.cloud.stream.instanceCount` is `1`, and `spring.cloud.stream.instanceIndex` is `0`.

In a scaled-up scenario, correct configuration of these two properties is important for addressing partitioning behavior (see below) in general, and the two properties are always required by certain binders (for example, the Kafka binder) in order to ensure that data are split correctly across multiple consumer instances.

20.12.6. Partitioning

Partitioning in Spring Cloud Stream consists of two tasks:

- [Configuring Output Bindings for Partitioning](#)
- [Configuring Input Bindings for Partitioning](#)

Configuring Output Bindings for Partitioning

You can configure an output binding to send partitioned data by setting one and only one of its `partitionKeyExpression` or `partitionKeyExtractorName` properties, as well as its `partitionCount` property.

For example, the following is a valid and typical configuration:

```
spring.cloud.stream.bindings.func-out-0.producer.partitionKeyExpression=payload.id
spring.cloud.stream.bindings.func-out-0.producer.partitionCount=5
```

Based on that example configuration, data is sent to the target partition by using the following logic.

A partition key's value is calculated for each message sent to a partitioned output binding based on the `partitionKeyExpression`. The `partitionKeyExpression` is a SpEL expression that is evaluated against the outbound message for extracting the partitioning key.

If a SpEL expression is not sufficient for your needs, you can instead calculate the partition key value by providing an implementation of `org.springframework.cloud.stream.binder.PartitionKeyExtractorStrategy` and configuring it as a bean (by using the `@Bean` annotation). If you have more than one bean of type `org.springframework.cloud.stream.binder.PartitionKeyExtractorStrategy` available in the Application Context, you can further filter it by specifying its name with the `partitionKeyExtractorName` property, as shown in the following example:

```
--spring.cloud.stream.bindings.func-out
-0.producer.partitionKeyExtractorName=customPartitionKeyExtractor
--spring.cloud.stream.bindings.func-out-0.producer.partitionCount=5
. . .
@Bean
public CustomPartitionKeyExtractorClass customPartitionKeyExtractor() {
    return new CustomPartitionKeyExtractorClass();
}
```



In previous versions of Spring Cloud Stream, you could specify the implementation of `org.springframework.cloud.stream.binder.PartitionKeyExtractorStrategy` by setting the `spring.cloud.stream.bindings.output.producer.partitionKeyExtractorClass` property. Since version 3.0, this property is removed.

Once the message key is calculated, the partition selection process determines the target partition as a value between `0` and `partitionCount - 1`. The default calculation, applicable in most scenarios, is based on the following formula: `key.hashCode() % partitionCount`. This can be customized on the binding, either by setting a SpEL expression to be evaluated against the 'key' (through the `partitionSelectorExpression` property) or by configuring an implementation of `org.springframework.cloud.stream.binder.PartitionSelectorStrategy` as a bean (by using the `@Bean` annotation). Similar to the `PartitionKeyExtractorStrategy`, you can further filter it by using the

`spring.cloud.stream.bindings.output.producer.partitionSelectorName` property when more than one bean of this type is available in the Application Context, as shown in the following example:

```
--spring.cloud.stream.bindings.func-out
-0.producer.partitionSelectorName=customPartitionSelector
. . .
@Bean
public CustomPartitionSelectorClass customPartitionSelector() {
    return new CustomPartitionSelectorClass();
}
```



In previous versions of Spring Cloud Stream you could specify the implementation of `org.springframework.cloud.stream.binder.PartitionSelectorStrategy` by setting the `spring.cloud.stream.bindings.output.producer.partitionSelectorClass` property. Since version 3.0, this property is removed.

Configuring Input Bindings for Partitioning

An input binding (with the binding name `uppercase-in-0`) is configured to receive partitioned data by setting its `partitioned` property, as well as the `instanceIndex` and `instanceCount` properties on the application itself, as shown in the following example:

```
spring.cloud.stream.bindings.uppercase-in-0.consumer.partitioned=true
spring.cloud.stream.instanceIndex=3
spring.cloud.stream.instanceCount=5
```

The `instanceCount` value represents the total number of application instances between which the data should be partitioned. The `instanceIndex` must be a unique value across the multiple instances, with a value between `0` and `instanceCount - 1`. The instance index helps each application instance to identify the unique partition(s) from which it receives data. It is required by binders using technology that does not support partitioning natively. For example, with RabbitMQ, there is a queue for each partition, with the queue name containing the instance index. With Kafka, if `autoRebalanceEnabled` is `true` (default), Kafka takes care of distributing partitions across instances, and these properties are not required. If `autoRebalanceEnabled` is set to false, the `instanceCount` and `instanceIndex` are used by the binder to determine which partition(s) the instance subscribes to (you must have at least as many partitions as there are instances). The binder allocates the partitions instead of Kafka. This might be useful if you want messages for a particular partition to always go to the same instance. When a binder configuration requires them, it is important to set both values correctly in order to ensure that all of the data is consumed and that the application instances receive mutually exclusive datasets.

While a scenario in which using multiple instances for partitioned data processing may be complex to set up in a standalone case, Spring Cloud Dataflow can simplify the process significantly by populating both the input and output values correctly and by letting you rely on the runtime infrastructure to provide information about the instance index and instance count.

20.13. Testing

Spring Cloud Stream provides support for testing your microservice applications without connecting to a messaging system.

20.13.1. Spring Integration Test Binder

The old test binder defined in `spring-cloud-stream-test-support` module was specifically designed to facilitate *unit testing* of the actual messaging components and thus bypasses some of the core functionality of the binder API.

While such light-weight approach is sufficient for a lot of cases, it usually requires additional *integration testing* with real binders (e.g., Rabbit, Kafka etc). So we are effectively deprecating it.

To begin bridging the gap between *unit* and *integration* testing we've developed a new test binder which uses [Spring Integration](#) framework as an in-JVM Message Broker essentially giving you the best of both worlds - a real binder without the networking.

Test Binder configuration

To enable Spring Integration Test Binder all you need is:

- Add required dependencies
- Remove the dependency for `spring-cloud-stream-test-support`

Add required dependencies

Below is the example of the required Maven POM entries which could be easily retrofitted into Gradle.

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-stream</artifactId>
  <version>${spring.cloud.stream.version}</version>
  <type>test-jar</type>
  <scope>test</scope>
  <classifier>test-binder</classifier>
</dependency>
```

Remove the dependency for `spring-cloud-stream-test-support`

To avoid conflicts with the existing test binder you must remove the following entry

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-stream-test-support</artifactId>
  <scope>test</scope>
</dependency>
```

Test Binder usage

Now you can test your microservice as a simple unit test

```
@SpringBootTest
@RunWith(SpringRunner.class)
public class SampleStreamTests {

    @Autowired
    private InputDestination input;

    @Autowired
    private OutputDestination output;

    @Test
    public void testEmptyConfiguration() {
        this.input.send(new GenericMessage<byte[]>("hello".getBytes()));
        assertThat(output.receive().getPayload()).isEqualTo("HELLO".getBytes());
    }

    @SpringBootApplication
    @Import(TestChannelBinderConfiguration.class)
    public static class SampleConfiguration {
        @Bean
        public Function<String, String> uppercase() {
            return v -> v.toUpperCase();
        }
    }
}
```

And if you need more control or want to test several configurations in the same test suite you can also do the following:


```

@EnableAutoConfiguration
public static class MyTestConfiguration {
    @Bean
    public Function<String, String> uppercase() {
        return v -> v.toUpperCase();
    }
}

...

@Test
public void sampleTest() {
    try (ConfigurableApplicationContext context = new SpringApplicationBuilder(
        TestChannelBinderConfiguration.getCompleteConfiguration(
            MyTestConfiguration.class))
        .run("--spring.cloud.function.definition=uppercase")) {
        InputDestination source = context.getBean(InputDestination.class);
        OutputDestination target = context.getBean(OutputDestination.class);
        source.send(new GenericMessage<byte[]>("hello".getBytes()));
        assertThat(target.receive().getPayload()).isEqualTo("HELLO".getBytes());
    }
}

```

For cases where you have multiple bindings and/or multiple inputs and outputs, or simply want to be explicit about names of the destination you are sending to or receiving from, the `send()` and `receive()` methods of `InputDestination` and `OutputDestination` are overridden to allow you to provide the name of the input and output destination.

Consider the following sample:

```

@EnableAutoConfiguration
public static class SampleFunctionConfiguration {

    @Bean
    public Function<String, String> uppercase() {
        return value -> value.toUpperCase();
    }

    @Bean
    public Function<String, String> reverse() {
        return value -> new StringBuilder(value).reverse().toString();
    }
}

```

and the actual test

```

@Test
public void testMultipleFunctions() {
    try (ConfigurableApplicationContext context = new SpringApplicationBuilder(
        TestChannelBinderConfiguration.getCompleteConfiguration(
            SampleFunctionConfiguration.class))
        .run("--
spring.cloud.function.definition=uppercase;reverse")) {
        context.getBean(InputDestination.class);

        InputDestination inputDestination = context.getBean(InputDestination.class);
        OutputDestination outputDestination =
context.getBean(OutputDestination.class);

        Message<byte[]> inputMessage =
MessageBuilder.withPayload("Hello".getBytes()).build();
        inputDestination.send(inputMessage, "uppercase-in-0");
        inputDestination.send(inputMessage, "uppercase-in-0");

        Message<byte[]> outputMessage = outputDestination.receive(0, "uppercase-out-
0");
        assertThat(outputMessage.getPayload()).isEqualTo("HELLO".getBytes());

        outputMessage = outputDestination.receive(0, "uppercase-out-0");
        assertThat(outputMessage.getPayload()).isEqualTo("olleH".getBytes());
    }
}

```

For cases where you have additional mapping properties such as `destination` you should use those names. For example, consider a different version of the preceding test where we explicitly map inputs and outputs of the `uppercase` function to `myInput` and `myOutput` binding names:

```

@Test
public void testMultipleFunctions() {
    try (ConfigurableApplicationContext context = new SpringApplicationBuilder(
        TestChannelBinderConfiguration.getCompleteConfiguration(
            SampleFunctionConfiguration.class))
        .run(
            "--spring.cloud.function.definition=uppercase;reverse",
            "--spring.cloud.stream.bindings.uppercase-in-
0.destination=myInput",
            "--spring.cloud.stream.bindings.uppercase-out-
0.destination=myOutput",
        )) {
        context.getBean(InputDestination.class);

        InputDestination inputDestination = context.getBean(InputDestination.class);
        OutputDestination outputDestination =
context.getBean(OutputDestination.class);

        Message<byte[]> inputMessage =
MessageBuilder.withPayload("Hello".getBytes()).build();
        inputDestination.send(inputMessage, "myInput");
        inputDestination.send(inputMessage, "myInput");

        Message<byte[]> outputMessage = outputDestination.receive(0, "myOutput");
        assertThat(outputMessage.getPayload().isEqualTo("HELLO".getBytes());

        outputMessage = outputDestination.receive(0, "myOutput");
        assertThat(outputMessage.getPayload().isEqualTo("olleH".getBytes());
    }
}

```

You can also use this binder with legacy annotation-based configuration:

```

@SpringBootApplication
@EnableBinding(Processor.class)
public class LegacyStreamApplication {

    @StreamListener(Processor.INPUT)
    @SendTo(Processor.OUTPUT)
    public String echo(String value) {
        return value;
    }
}

...

@Test
public void sampleTest() {
    try (ConfigurableApplicationContext context = new SpringApplicationBuilder(
        TestChannelBinderConfiguration.getCompleteConfiguration(
            LegacyStreamApplication.class)).run()) {
        InputDestination source = context.getBean(InputDestination.class);
        OutputDestination target = context.getBean(OutputDestination.class);
        source.send(new GenericMessage<byte[]>("hello".getBytes()));
        assertThat(target.receive().getPayload()).isEqualTo("hello".getBytes());
    }
}

```

In the above you simply create an `ApplicationContext` with your configuration (your application) while additionally supplying `TestChannelBinderConfiguration` provided by the framework. Then you access `InputDestination` and `OutputDestination` beans to send/receive messages. In the context of this binder `InputDestination` and `OutputDestination` emulate remote destinations such as Rabbit *exchange/queue* or Kafka *topic*.

In the future we plan to simplify the API.



In its current state Spring Integration Test Binder only supports the three bindings provided by the framework (Source, Processor, Sink) specifically to promote light-weight microservices architectures rather than general purpose messaging applications.

Test Binder and `PollableMessageSource`

Spring Integration Test Binder also allows you to write tests when working with `PollableMessageSource` (see [Using Polled Consumers](#) for more details).

The important thing that needs to be understood though is that polling is not event-driven, and that `PollableMessageSource` is a strategy which exposes operation to produce (poll for) a `Message` (singular). How often you poll or how many threads you use or where you're polling from (message queue or file system) is entirely up to you; In other words it is your responsibility to configure Poller or Threads or the actual source of `Message`. Luckily Spring has plenty of abstractions to configure exactly that.

Let's look at the example:

```
@Test
public void samplePollingTest() {
    ApplicationContext context = new
    SpringApplication(builder(SamplePolledConfiguration.class)
        .web(WebApplicationType.NONE)
        .run("--spring.jmx.enabled=false"));
    OutputDestination destination = context.getBean(OutputDestination.class);
    System.out.println("Message 1: " + new
    String(destination.receive().getPayload()));
    System.out.println("Message 2: " + new
    String(destination.receive().getPayload()));
    System.out.println("Message 3: " + new
    String(destination.receive().getPayload()));
}

@EnableBinding(SamplePolledConfiguration.PolledConsumer.class)
@Import(TestChannelBinderConfiguration.class)
@EnableAutoConfiguration
public static class SamplePolledConfiguration {
    @Bean
    public ApplicationRunner poller(PollableMessageSource polledMessageSource,
    MessageChannel output, TaskExecutor taskScheduler) {
        return args -> {
            taskScheduler.execute(() -> {
                for (int i = 0; i < 3; i++) {
                    try {
                        if (!polledMessageSource.poll(m -> {
                            String newPayload = ((String)
m.getPayload()).toUpperCase();
                            output.send(new GenericMessage<>(newPayload));
                        }))) {
                            Thread.sleep(2000);
                        }
                    }
                    catch (Exception e) {
                        // handle failure
                    }
                }
            });
        };
    }

    public static interface PolledConsumer extends Source {
        @Input
        PollableMessageSource pollableSource();
    }
}
```

The above (very rudimentary) example will produce 3 messages in 2 second intervals sending them to the output destination of `Source` which this binder sends to `OutputDestination` where we retrieve them (for any assertions). Currently it prints the following:

```
Message 1: POLLED DATA
Message 2: POLLED DATA
Message 3: POLLED DATA
```

As you can see the data is the same. That is because this binder defines a default implementation of the actual `MessageSource` - the source from which the Messages are polled using `poll()` operation. While sufficient for most testing scenarios, there are cases where you may want to define your own `MessageSource`. To do so simply configure a bean of type `MessageSource` in your test configuration providing your own implementation of Message sourcing.

Here is the example:

```
@Bean
public MessageSource<?> source() {
    return () -> new GenericMessage<>("My Own Data " + UUID.randomUUID());
}
```

rendering the following output;

```
Message 1: MY OWN DATA 1C180A91-E79F-494F-ABF4-BA3F993710DA
Message 2: MY OWN DATA D8F3A477-5547-41B4-9434-E69DA7616FEE
Message 3: MY OWN DATA 20BF2E64-7FF4-4CB6-A823-4053D30B5C74
```



DO NOT name this bean `messageSource` as it is going to be in conflict with the bean of the same name (different type) provided by Spring Boot for unrelated reasons.

20.14. Health Indicator

Spring Cloud Stream provides a health indicator for binders. It is registered under the name `binders` and can be enabled or disabled by setting the `management.health.binders.enabled` property.

To enable health check you first need to enable both "web" and "actuator" by including its dependencies (see [Binding visualization and control](#))

If `management.health.binders.enabled` is not set explicitly by the application, then `management.health.defaults.enabled` is matched as `true` and the binder health indicators are enabled. If you want to disable health indicator completely, then you have to set `management.health.binders.enabled` to `false`.

You can use Spring Boot actuator health endpoint to access the health indicator - `/actuator/health`. By default, you will only receive the top level application status when you hit the above endpoint. In order to receive the full details from the binder specific health indicators, you need to include

the property `management.endpoint.health.show-details` with the value `ALWAYS` in your application.

Health indicators are binder-specific and certain binder implementations may not necessarily provide a health indicator.

If you want to completely disable all health indicators available out of the box and instead provide your own health indicators, you can do so by setting property `management.health.binders.enabled` to `false` and then provide your own `HealthIndicator` beans in your application. In this case, the health indicator infrastructure from Spring Boot will still pick up these custom beans. Even if you are not disabling the binder health indicators, you can still enhance the health checks by providing your own `HealthIndicator` beans in addition to the out of the box health checks.

When you have multiple binders in the same application, health indicators are enabled by default unless the application turns them off by setting `management.health.binders.enabled` to `false`. In this case, if the user wants to disable health check for a subset of the binders, then that should be done by setting `management.health.binders.enabled` to `false` in the multi binder configurations's environment. See [Connecting to Multiple Systems](#) for details on how environment specific properties can be provided.

If there are multiple binders present in the classpath but not all of them are used in the application, this may cause some issues in the context of health indicators. There may be implementation specific details as to how the health checks are performed. For example, a Kafka binder may decide the status as `DOWN` if there are no destinations registered by the binder. For this reason, if you include a binder in the classpath, it is advised to use that binder by providing at least one binding (for E.g. through `EnableBinding`). If you don't have any bindings to provide for this binder, then that is an indication that you don't need to include that binder in the classpath.

Lets take a concrete situation. Imagine you have both Kafka and Kafka Streams binders present in the classpath, but only use the Kafka Streams binder in the application code, i.e. only provide bindings using the Kafka Streams binder. Since Kafka binder is not used and it has specific checks to see if any destinations are registered, the binder health heck will fail. The top level application health check status will be reported as `DOWN`. In this situation, you can simply remove the dependency for kafka binder from your application since you are not using it.

20.15. Samples

For Spring Cloud Stream samples, see the [spring-cloud-stream-samples](#) repository on GitHub.

20.15.1. Deploying Stream Applications on CloudFoundry

On CloudFoundry, services are usually exposed through a special environment variable called `VCAP_SERVICES`.

When configuring your binder connections, you can use the values from an environment variable as explained on the [dataflow Cloud Foundry Server](#) docs.

20.16. Binder Implementations

The following is the list of available binder implementations

- [RabbitMQ](#)
- [Apache Kafka](#)
- [Amazon Kinesis](#)
- [Google PubSub](#) (*partner maintained*)
- [Solace PubSub+](#) (*partner maintained*)
- [Azure Event Hubs](#) (*partner maintained*)
- [Apache RocketMQ](#) (*partner maintained*)

As it was mentioned earlier Binder abstraction is also one of the extension points of the framework. So if you can't find a suitable binder in the preceding list you can implement your own binder on top of Spring Cloud Stream. In the [How to create a Spring Cloud Stream Binder from scratch](#) post a community member documents in details, with an example, a set of steps necessary to implement a custom binder. The steps are also highlighted in the [Implementing Custom Binders](#) section.

Chapter 21. Spring Cloud Stream RabbitMQ Binder Reference Guide

Sabby Anandan, Marius Bogoevici, Eric Bottard, Mark Fisher, Ilayaperumal Gopinathan, Gunnar Hillert, Mark Pollack, Patrick Peralta, Glenn Renfro, Thomas Risberg, Dave Syer, David Turanski, Janne Valkealahti, Benjamin Klein, Gary Russell, Jay Bryant :doctype: book :toc: :toclevels: 4 :source-highlighter: prettify :numbered: :icons: font :hide-uri-scheme: :spring-cloud-stream-binder-rabbit-repo: snapshot :github-tag: master :spring-cloud-stream-binder-rabbit-docs-version: current :spring-cloud-stream-binder-rabbit-docs: docs.spring.io/spring-cloud-stream-binder-rabbit/docs/{spring-cloud-stream-binder-rabbit-docs-version}/reference :spring-cloud-stream-binder-rabbit-docs-current: docs.spring.io/spring-cloud-stream-binder-rabbit/docs/current-SNAPSHOT/reference/html/ :github-repo: spring-cloud/spring-cloud-stream-binder-rabbit :github-raw: raw.githubusercontent.com/spring-cloud/spring-cloud-stream-binder-rabbit/master :github-code: github.com/spring-cloud/spring-cloud-stream-binder-rabbit/tree/master :github-wiki: github.com/spring-cloud/spring-cloud-stream-binder-rabbit/wiki :github-master-code: github.com/spring-cloud/spring-cloud-stream-binder-rabbit/tree/master :sc-ext: java

{spring-cloud-stream-version}

Chapter 22. Reference Guide

This guide describes the RabbitMQ implementation of the Spring Cloud Stream Binder. It contains information about its design, usage and configuration options, as well as information on how the Stream Cloud Stream concepts map into RabbitMQ specific constructs.

22.1. Usage

To use the RabbitMQ binder, you can add it to your Spring Cloud Stream application, by using the following Maven coordinates:

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-stream-binder-rabbit</artifactId>
</dependency>
```

Alternatively, you can use the Spring Cloud Stream RabbitMQ Starter, as follows:

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-stream-rabbit</artifactId>
</dependency>
```

22.2. RabbitMQ Binder Overview

The following simplified diagram shows how the RabbitMQ binder operates:

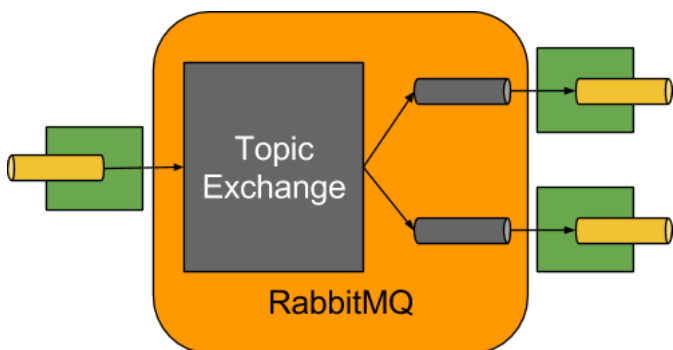


Figure 10. RabbitMQ Binder

By default, the RabbitMQ Binder implementation maps each destination to a `TopicExchange`. For each consumer group, a `Queue` is bound to that `TopicExchange`. Each consumer instance has a corresponding RabbitMQ `Consumer` instance for its group's `Queue`. For partitioned producers and consumers, the queues are suffixed with the partition index and use the partition index as the routing key. For anonymous consumers (those with no `group` property), an auto-delete queue (with a randomized unique name) is used.

By using the optional `autoBindDlq` option, you can configure the binder to create and configure

dead-letter queues (DLQs) (and a dead-letter exchange [DLX](#), as well as routing infrastructure). By default, the dead letter queue has the name of the destination, appended with `.dlq`. If retry is enabled (`maxAttempts > 1`), failed messages are delivered to the DLQ after retries are exhausted. If retry is disabled (`maxAttempts = 1`), you should set `requeueRejected` to `false` (the default) so that failed messages are routed to the DLQ, instead of being re-queued. In addition, `republishToDlq` causes the binder to publish a failed message to the DLQ (instead of rejecting it). This feature lets additional information (such as the stack trace in the `x-exception-stacktrace` header) be added to the message in headers. See the `frameMaxHeadroom` [property](#) for information about truncated stack traces. This option does not need retry enabled. You can republish a failed message after just one attempt. Starting with version 1.2, you can configure the delivery mode of republished messages. See the `republishDeliveryMode` [property](#).

If the stream listener throws an `ImmediateAcknowledgeAmqpException`, the DLQ is bypassed and the message simply discarded. Starting with version 2.1, this is true regardless of the setting of `republishToDlq`; previously it was only the case when `republishToDlq` was `false`.



Setting `requeueRejected` to `true` (with `republishToDlq=false`) causes the message to be re-queued and redelivered continually, which is likely not what you want unless the reason for the failure is transient. In general, you should enable retry within the binder by setting `maxAttempts` to greater than one or by setting `republishToDlq` to `true`.

See [RabbitMQ Binder Properties](#) for more information about these properties.

The framework does not provide any standard mechanism to consume dead-letter messages (or to re-route them back to the primary queue). Some options are described in [Dead-Letter Queue Processing](#).



When multiple RabbitMQ binders are used in a Spring Cloud Stream application, it is important to disable 'RabbitAutoConfiguration' to avoid the same configuration from `RabbitAutoConfiguration` being applied to the two binders. You can exclude the class by using the `@SpringBootApplication` annotation.

Starting with version 2.0, the `RabbitMessageChannelBinder` sets the `RabbitTemplate.userPublisherConnection` property to `true` so that the non-transactional producers avoid deadlocks on consumers, which can happen if cached connections are blocked because of a [memory alarm](#) on the broker.



Currently, a `multiplex` consumer (a single consumer listening to multiple queues) is only supported for message-driven consumers; polled consumers can only retrieve messages from a single queue.

22.3. Configuration Options

This section contains settings specific to the RabbitMQ Binder and bound channels.

For general binding configuration options and properties, see the [Spring Cloud Stream core documentation](#).

22.3.1. RabbitMQ Binder Properties

By default, the RabbitMQ binder uses Spring Boot's `ConnectionFactory`. Consequently, it supports all Spring Boot configuration options for RabbitMQ. (For reference, see the [Spring Boot documentation](#)). RabbitMQ configuration options use the `spring.rabbitmq` prefix.

In addition to Spring Boot options, the RabbitMQ binder supports the following properties:

`spring.cloud.stream.rabbit.binder.adminAddresses`

A comma-separated list of RabbitMQ management plugin URLs. Only used when `nodes` contains more than one entry. Each entry in this list must have a corresponding entry in `spring.rabbitmq.addresses`. Only needed if you use a RabbitMQ cluster and wish to consume from the node that hosts the queue. See [Queue Affinity and the LocalizedQueueConnectionFactory](#) for more information.

Default: empty.

`spring.cloud.stream.rabbit.binder.nodes`

A comma-separated list of RabbitMQ node names. When more than one entry, used to locate the server address where a queue is located. Each entry in this list must have a corresponding entry in `spring.rabbitmq.addresses`. Only needed if you use a RabbitMQ cluster and wish to consume from the node that hosts the queue. See [Queue Affinity and the LocalizedQueueConnectionFactory](#) for more information.

Default: empty.

`spring.cloud.stream.rabbit.binder.compressionLevel`

The compression level for compressed bindings. See `java.util.zip.Deflater`.

Default: 1 (BEST_LEVEL).

`spring.cloud.stream.binder.connection-name-prefix`

A connection name prefix used to name the connection(s) created by this binder. The name is this prefix followed by `#n`, where `n` increments each time a new connection is opened.

Default: none (Spring AMQP default).

22.3.2. RabbitMQ Consumer Properties

The following properties are available for Rabbit consumers only and must be prefixed with `spring.cloud.stream.rabbit.bindings.<channelName>.consumer..`

However if the same set of properties needs to be applied to most bindings, to avoid repetition, Spring Cloud Stream supports setting values for all channels, in the format of `spring.cloud.stream.rabbit.default.<property>=<value>`.

Also, keep in mind that binding specific property will override its equivalent in the default.

`acknowledgeMode`

The acknowledge mode.

Default: `AUTO`.

anonymousGroupPrefix

When the binding has no `group` property, an anonymous, auto-delete queue is bound to the destination exchange. The default naming strategy for such queues results in a queue named `anonymous.<base64 representation of a UUID>`. Set this property to change the prefix to something other than the default.

Default: `anonymous..`

autoBindDlq

Whether to automatically declare the DLQ and bind it to the binder DLX.

Default: `false`.

bindingRoutingKey

The routing key with which to bind the queue to the exchange (if `bindQueue` is `true`). Can be multiple keys - see `bindingRoutingKeyDelimiter`. For partitioned destinations, `-<instanceIndex>` is appended to each key.

Default: `#`.

bindingRoutingKeyDelimiter

When this is not null, 'bindingRoutingKey' is considered to be a list of keys delimited by this value; often a comma is used.

Default: `null`.

bindQueue

Whether to declare the queue and bind it to the destination exchange. Set it to `false` if you have set up your own infrastructure and have previously created and bound the queue.

Default: `true`.

consumerTagPrefix

Used to create the consumer tag(s); will be appended by `#n` where `n` increments for each consumer created. Example: `${spring.application.name}-${spring.cloud.stream.bindings.input.group}-${spring.cloud.stream.instance-index}`.

Default: none - the broker will generate random consumer tags.

containerType

Select the type of listener container to be used. See [Choosing a Container](#) in the Spring AMQP documentation for more information.

Default: `simple`

deadLetterQueueName

The name of the DLQ

Default: `prefix+destination.dlq`

deadLetterExchange

A DLX to assign to the queue. Relevant only if `autoBindDlq` is `true`.

Default: `'prefix+DLX'`

deadLetterExchangeType

The type of the DLX to assign to the queue. Relevant only if `autoBindDlq` is `true`.

Default: `'direct'`

deadLetterRoutingKey

A dead letter routing key to assign to the queue. Relevant only if `autoBindDlq` is `true`.

Default: `destination`

declareDlx

Whether to declare the dead letter exchange for the destination. Relevant only if `autoBindDlq` is `true`. Set to `false` if you have a pre-configured DLX.

Default: `true`.

declareExchange

Whether to declare the exchange for the destination.

Default: `true`.

delayedExchange

Whether to declare the exchange as a `Delayed Message Exchange`. Requires the delayed message exchange plugin on the broker. The `x-delayed-type` argument is set to the `exchangeType`.

Default: `false`.

dlqBindingArguments

Arguments applied when binding the dlq to the dead letter exchange; used with `headers deadLetterExchangeType` to specify headers to match on. For example `···dlqBindingArguments.x-match=any`, `···dlqBindingArguments.someHeader=someValue`.

Default: empty

dlqDeadLetterExchange

If a DLQ is declared, a DLX to assign to that queue.

Default: `none`

dlqDeadLetterRoutingKey

If a DLQ is declared, a dead letter routing key to assign to that queue.

Default: `none`

dlqExpires

How long before an unused dead letter queue is deleted (in milliseconds).

Default: `no expiration`

dlqLazy

Declare the dead letter queue with the `x-queue-mode=lazy` argument. See “[Lazy Queues](#)”. Consider using a policy instead of this setting, because using a policy allows changing the setting without deleting the queue.

Default: `false`.

dlqMaxLength

Maximum number of messages in the dead letter queue.

Default: `no limit`

dlqMaxLengthBytes

Maximum number of total bytes in the dead letter queue from all messages.

Default: `no limit`

dlqMaxPriority

Maximum priority of messages in the dead letter queue (0-255).

Default: `none`

dlqOverflowBehavior

Action to take when `dlqMaxLength` or `dlqMaxLengthBytes` is exceeded; currently `drop-head` or `reject-publish` but refer to the RabbitMQ documentation.

Default: `none`

dlqQuorum.deliveryLimit

When `quorum.enabled=true`, set a delivery limit after which the message is dropped or dead-lettered.

Default: none - broker default will apply.

dlqQuorum.enabled

When true, create a quorum dead letter queue instead of a classic queue.

Default: false

dlqQuorum.initialQuorumSize

When `quorum.enabled=true`, set the initial quorum size.

Default: none - broker default will apply.

dlqSingleActiveConsumer

Set to true to set the `x-single-active-consumer` queue property to true.

Default: `false`

dlqTtl

Default time to live to apply to the dead letter queue when declared (in milliseconds).

Default: `no limit`

durableSubscription

Whether the subscription should be durable. Only effective if `group` is also set.

Default: `true`.

exchangeAutoDelete

If `declareExchange` is true, whether the exchange should be auto-deleted (that is, removed after the last queue is removed).

Default: `true`.

exchangeDurable

If `declareExchange` is true, whether the exchange should be durable (that is, it survives broker restart).

Default: `true`.

exchangeType

The exchange type: `direct`, `fanout`, `headers` or `topic` for non-partitioned destinations and `direct`, `headers` or `topic` for partitioned destinations.

Default: `topic`.

exclusive

Whether to create an exclusive consumer. Concurrency should be 1 when this is `true`. Often used when strict ordering is required but enabling a hot standby instance to take over after a failure. See `recoveryInterval`, which controls how often a standby instance attempts to consume. Consider using `singleActiveConsumer` instead when using RabbitMQ 3.8 or later.

Default: `false`.

expires

How long before an unused queue is deleted (in milliseconds).

Default: `no expiration`

failedDeclarationRetryInterval

The interval (in milliseconds) between attempts to consume from a queue if it is missing.

Default: 5000

frameMaxHeadroom

The number of bytes to reserve for other headers when adding the stack trace to a DLQ message header. All headers must fit within the `frame_max` size configured on the broker. Stack traces can be large; if the size plus this property exceeds `frame_max` then the stack trace will be truncated. A WARN log will be written; consider increasing the `frame_max` or reducing the stack trace by catching the exception and throwing one with a smaller stack trace.

Default: 20000

headerPatterns

Patterns for headers to be mapped from inbound messages.

Default: `['*']` (all headers).

lazy

Declare the queue with the `x-queue-mode=lazy` argument. See “[Lazy Queues](#)”. Consider using a policy instead of this setting, because using a policy allows changing the setting without deleting the queue.

Default: `false`.

maxConcurrency

The maximum number of consumers. Not supported when the `containerType` is `direct`.

Default: `1`.

maxLength

The maximum number of messages in the queue.

Default: `no limit`

maxLengthBytes

The maximum number of total bytes in the queue from all messages.

Default: `no limit`

maxPriority

The maximum priority of messages in the queue (0-255).

Default: `none`

missingQueuesFatal

When the queue cannot be found, whether to treat the condition as fatal and stop the listener container. Defaults to `false` so that the container keeps trying to consume from the queue — for example, when using a cluster and the node hosting a non-HA queue is down.

Default: `false`

overflowBehavior

Action to take when `maxLength` or `maxLengthBytes` is exceeded; currently `drop-head` or `reject-`

`publish` but refer to the RabbitMQ documentation.

Default: `none`

prefetch

Prefetch count.

Default: `1`.

prefix

A prefix to be added to the name of the `destination` and queues.

Default: `""`.

queueBindingArguments

Arguments applied when binding the queue to the exchange; used with `headers exchangeType` to specify headers to match on. For example `...queueBindingArguments.x-match=any, ...queueBindingArguments.someHeader=someValue`.

Default: empty

queueDeclarationRetries

The number of times to retry consuming from a queue if it is missing. Relevant only when `missingQueuesFatal` is `true`. Otherwise, the container keeps retrying indefinitely. Not supported when the `containerType` is `direct`.

Default: `3`

queueNameGroupOnly

When true, consume from a queue with a name equal to the `group`. Otherwise the queue name is `destination.group`. This is useful, for example, when using Spring Cloud Stream to consume from an existing RabbitMQ queue.

Default: `false`.

quorum.deliveryLimit

When `quorum.enabled=true`, set a delivery limit after which the message is dropped or dead-lettered.

Default: `none` - broker default will apply.

quorum.enabled

When true, create a quorum queue instead of a classic queue.

Default: `false`

quorum.initialQuorumSize

When `quorum.enabled=true`, set the initial quorum size.

Default: `none` - broker default will apply.

recoveryInterval

The interval between connection recovery attempts, in milliseconds.

Default: `5000`.

requeueRejected

Whether delivery failures should be re-queued when retry is disabled or `republishToDlq` is `false`.

Default: `false`.

republishDeliveryMode

When `republishToDlq` is `true`, specifies the delivery mode of the republished message.

Default: `DeliveryMode.PERSISTENT`

republishToDlq

By default, messages that fail after retries are exhausted are rejected. If a dead-letter queue (DLQ) is configured, RabbitMQ routes the failed message (unchanged) to the DLQ. If set to `true`, the binder republishes failed messages to the DLQ with additional headers, including the exception message and stack trace from the cause of the final failure. Also see the [frameMaxHeadroom](#) property.

Default: `false`

singleActiveConsumer

Set to true to set the `x-single-active-consumer` queue property to true.

Default: `false`

transacted

Whether to use transacted channels.

Default: `false`.

ttl

Default time to live to apply to the queue when declared (in milliseconds).

Default: `no limit`

txSize

The number of deliveries between acks. Not supported when the `containerType` is `direct`.

Default: `1`.

22.3.3. Advanced Listener Container Configuration

To set listener container properties that are not exposed as binder or binding properties, add a single bean of type `ListenerContainerCustomizer` to the application context. The binder and binding properties will be set and then the customizer will be called. The customizer (`configure()` method) is provided with the queue name as well as the consumer group as arguments.

22.3.4. Advanced Queue/Exchange/Binding Configuration

From time to time, the RabbitMQ team add new features that are enabled by setting some argument when declaring, for example, a queue. Generally, such features are enabled in the binder by adding appropriate properties, but this may not be immediately available in a current version. Starting with version 3.0.1, you can now add `DeclarableCustomizer` bean(s) to the application context to modify a `Declarable` (Queue, Exchange or Binding) just before the declaration is performed. This allows you to add arguments that are not currently directly supported by the binder.

22.3.5. Receiving Batched Messages

Normally, if a producer binding has `batch-enabled=true` (see [Rabbit Producer Properties](#)), or a message is created by a `BatchingRabbitTemplate`, elements of the batch are returned as individual calls to the listener method. Starting with version 3.0, any such batch can be presented as a `List<?>` to the listener method if `spring.cloud.stream.bindings.<name>.consumer.batch-mode` is set to `true`.

22.3.6. Rabbit Producer Properties

The following properties are available for Rabbit producers only and must be prefixed with `spring.cloud.stream.rabbit.bindings.<channelName>.producer..`

However if the same set of properties needs to be applied to most bindings, to avoid repetition, Spring Cloud Stream supports setting values for all channels, in the format of `spring.cloud.stream.rabbit.default.<property>=<value>`.

Also, keep in mind that binding specific property will override its equivalent in the default.

autoBindDlq

Whether to automatically declare the DLQ and bind it to the binder DLX.

Default: `false`.

batchingEnabled

Whether to enable message batching by producers. Messages are batched into one message according to the following properties (described in the next three entries in this list): 'batchSize', `batchBufferLimit`, and `batchTimeout`. See [Batching](#) for more information. Also see [Receiving Batched Messages](#).

Default: `false`.

batchSize

The number of messages to buffer when batching is enabled.

Default: `100`.

batchBufferLimit

The maximum buffer size when batching is enabled.

Default: `10000`.

batchTimeout

The batch timeout when batching is enabled.

Default: `5000`.

bindingRoutingKey

The routing key with which to bind the queue to the exchange (if `bindQueue` is `true`). Can be multiple keys - see `bindingRoutingKeyDelimiter`. For partitioned destinations, `-n` is appended to each key. Only applies if `requiredGroups` are provided and then only to those groups.

Default: `#`.

bindingRoutingKeyDelimiter

When this is not null, 'bindingRoutingKey' is considered to be a list of keys delimited by this value; often a comma is used. Only applies if `requiredGroups` are provided and then only to those groups.

Default: `null`.

bindQueue

Whether to declare the queue and bind it to the destination exchange. Set it to `false` if you have set up your own infrastructure and have previously created and bound the queue. Only applies if `requiredGroups` are provided and then only to those groups.

Default: `true`.

compress

Whether data should be compressed when sent.

Default: `false`.

confirmAckChannel

When `errorChannelEnabled` is true, a channel to which to send positive delivery acknowledgments (aka publisher confirms). If the channel does not exist, a `DirectChannel` is registered with this name. The connection factory must be configured to enable publisher confirms.

Default: `nullChannel` (acks are discarded).

deadLetterQueueName

The name of the DLQ Only applies if `requiredGroups` are provided and then only to those groups.

Default: `prefix+destination.dlq`

deadLetterExchange

A DLX to assign to the queue. Relevant only when `autoBindDlq` is `true`. Applies only when `requiredGroups` are provided and then only to those groups.

Default: `'prefix+DLX'`

deadLetterExchangeType

The type of the DLX to assign to the queue. Relevant only if `autoBindDlq` is `true`. Applies only when `requiredGroups` are provided and then only to those groups.

Default: 'direct'

deadLetterRoutingKey

A dead letter routing key to assign to the queue. Relevant only when `autoBindDlq` is `true`. Applies only when `requiredGroups` are provided and then only to those groups.

Default: `destination`

declareDlx

Whether to declare the dead letter exchange for the destination. Relevant only if `autoBindDlq` is `true`. Set to `false` if you have a pre-configured DLX. Applies only when `requiredGroups` are provided and then only to those groups.

Default: `true`.

declareExchange

Whether to declare the exchange for the destination.

Default: `true`.

delayExpression

A SpEL expression to evaluate the delay to apply to the message (`x-delay` header). It has no effect if the exchange is not a delayed message exchange.

Default: No `x-delay` header is set.

delayedExchange

Whether to declare the exchange as a `Delayed Message Exchange`. Requires the delayed message exchange plugin on the broker. The `x-delayed-type` argument is set to the `exchangeType`.

Default: `false`.

deliveryMode

The delivery mode.

Default: `PERSISTENT`.

dlqBindingArguments

Arguments applied when binding the dlq to the dead letter exchange; used with headers `deadLetterExchangeType` to specify headers to match on. For example `...dlqBindingArguments.x-match=any`, `...dlqBindingArguments.someHeader=someValue`. Applies only when `requiredGroups` are provided and then only to those groups.

Default: empty

dlqDeadLetterExchange

When a DLQ is declared, a DLX to assign to that queue. Applies only if `requiredGroups` are provided and then only to those groups.

Default: `none`

dlqDeadLetterRoutingKey

When a DLQ is declared, a dead letter routing key to assign to that queue. Applies only when `requiredGroups` are provided and then only to those groups.

Default: `none`

dlqExpires

How long (in milliseconds) before an unused dead letter queue is deleted. Applies only when `requiredGroups` are provided and then only to those groups.

Default: `no expiration`

dlqLazy

Declare the dead letter queue with the `x-queue-mode=lazy` argument. See “[Lazy Queues](#)”. Consider using a policy instead of this setting, because using a policy allows changing the setting without deleting the queue. Applies only when `requiredGroups` are provided and then only to those groups.

dlqMaxLength

Maximum number of messages in the dead letter queue. Applies only if `requiredGroups` are provided and then only to those groups.

Default: `no limit`

dlqMaxLengthBytes

Maximum number of total bytes in the dead letter queue from all messages. Applies only when `requiredGroups` are provided and then only to those groups.

Default: `no limit`

dlqMaxPriority

Maximum priority of messages in the dead letter queue (0-255) Applies only when `requiredGroups` are provided and then only to those groups.

Default: `none`

dlqQuorum.deliveryLimit

When `quorum.enabled=true`, set a delivery limit after which the message is dropped or dead-lettered. Applies only when `requiredGroups` are provided and then only to those groups.

Default: `none` - broker default will apply.

dlqQuorum.enabled

When true, create a quorum dead letter queue instead of a classic queue. Applies only when `requiredGroups` are provided and then only to those groups.

Default: false

dlqQuorum.initialQuorumSize

When `quorum.enabled=true`, set the initial quorum size. Applies only when `requiredGroups` are provided and then only to those groups.

Default: none - broker default will apply.

dlqSingleActiveConsumer

Set to true to set the `x-single-active-consumer` queue property to true. Applies only when `requiredGroups` are provided and then only to those groups.

Default: false

dlqTtl

Default time (in milliseconds) to live to apply to the dead letter queue when declared. Applies only when `requiredGroups` are provided and then only to those groups.

Default: no limit

exchangeAutoDelete

If `declareExchange` is true, whether the exchange should be auto-delete (it is removed after the last queue is removed).

Default: true.

exchangeDurable

If `declareExchange` is true, whether the exchange should be durable (survives broker restart).

Default: true.

exchangeType

The exchange type: `direct`, `fanout`, `headers` or `topic` for non-partitioned destinations and `direct`, `headers` or `topic` for partitioned destinations.

Default: `topic`.

expires

How long (in milliseconds) before an unused queue is deleted. Applies only when `requiredGroups` are provided and then only to those groups.

Default: no expiration

headerPatterns

Patterns for headers to be mapped to outbound messages.

Default: `['*']` (all headers).

lazy

Declare the queue with the `x-queue-mode=lazy` argument. See “[Lazy Queues](#)”. Consider using a policy instead of this setting, because using a policy allows changing the setting without deleting the queue. Applies only when `requiredGroups` are provided and then only to those groups.

Default: `false`.

maxLength

Maximum number of messages in the queue. Applies only when `requiredGroups` are provided and then only to those groups.

Default: `no limit`

maxLengthBytes

Maximum number of total bytes in the queue from all messages. Only applies if `requiredGroups` are provided and then only to those groups.

Default: `no limit`

maxPriority

Maximum priority of messages in the queue (0-255). Only applies if `requiredGroups` are provided and then only to those groups.

Default: `none`

prefix

A prefix to be added to the name of the `destination` exchange.

Default: `""`.

queueBindingArguments

Arguments applied when binding the queue to the exchange; used with `headers exchangeType` to specify headers to match on. For example `...queueBindingArguments.x-match=any, ...queueBindingArguments.someHeader=someValue`. Applies only when `requiredGroups` are provided and then only to those groups.

Default: empty

queueNameGroupOnly

When `true`, consume from a queue with a name equal to the `group`. Otherwise the queue name is `destination.group`. This is useful, for example, when using Spring Cloud Stream to consume from an existing RabbitMQ queue. Applies only when `requiredGroups` are provided and then only to those groups.

Default: `false`.

quorum.deliveryLimit

When `quorum.enabled=true`, set a delivery limit after which the message is dropped or dead-lettered. Applies only when `requiredGroups` are provided and then only to those groups.

Default: none - broker default will apply.

quorum.enabled

When true, create a quorum queue instead of a classic queue. Applies only when `requiredGroups` are provided and then only to those groups.

Default: false

quorum.initialQuorumSize

When `quorum.enabled=true`, set the initial quorum size. Applies only when `requiredGroups` are provided and then only to those groups.

Default: none - broker default will apply.

routingKeyExpression

A SpEL expression to determine the routing key to use when publishing messages. For a fixed routing key, use a literal expression, such as `routingKeyExpression='my.routingKey'` in a properties file or `routingKeyExpression: 'my.routingKey'` in a YAML file.

Default: `destination` or `destination-<partition>` for partitioned destinations.

singleActiveConsumer

Set to true to set the `x-single-active-consumer` queue property to true. Applies only when `requiredGroups` are provided and then only to those groups.

Default: false

transacted

Whether to use transacted channels.

Default: false.

ttl

Default time (in milliseconds) to live to apply to the queue when declared. Applies only when `requiredGroups` are provided and then only to those groups.

Default: no limit



In the case of RabbitMQ, content type headers can be set by external applications. Spring Cloud Stream supports them as part of an extended internal protocol used for any type of transport — including transports, such as Kafka (prior to 0.11), that do not natively support headers.

22.4. Using Existing Queues/Exchanges

By default, the binder will automatically provision a topic exchange with the name being derived from the value of the destination binding property `<prefix><destination>`. The destination defaults to the binding name, if not provided. When binding a consumer, a queue will automatically be provisioned with the name `<prefix><destination>.<group>` (if a `group` binding property is specified),

or an anonymous, auto-delete queue when there is no `group`. The queue will be bound to the exchange with the "match-all" wildcard routing key (`#`) for a non-partitioned binding or `<destination>-<instanceIndex>` for a partitioned binding. The prefix is an empty `String` by default. If an output binding is specified with `requiredGroups`, a queue/binding will be provisioned for each group.

There are a number of rabbit-specific binding properties that allow you to modify this default behavior.

If you have an existing exchange/queue that you wish to use, you can completely disable automatic provisioning as follows, assuming the exchange is named `myExchange` and the queue is named `myQueue`:

- `spring.cloud.stream.bindings.<binding name>.destination=myExchange`
- `spring.cloud.stream.bindings.<binding name>.group=myQueue`
- `spring.cloud.stream.rabbit.bindings.<binding name>.consumer.bindQueue=false`
- `spring.cloud.stream.rabbit.bindings.<binding name>.consumer.declareExchange=false`
- `spring.cloud.stream.rabbit.bindings.<binding name>.consumer.queueNameGroupOnly=true`

If you want the binder to provision the queue/exchange, but you want to do it using something other than the defaults discussed here, use the following properties. Refer to the property documentation above for more information.

- `spring.cloud.stream.rabbit.bindings.<binding name>.consumer.bindingRoutingKey=myRoutingKey`
- `spring.cloud.stream.rabbit.bindings.<binding name>.consumer.exchangeType=<type>`
- `spring.cloud.stream.rabbit.bindings.<binding name>.producer.routingKeyExpression='myRoutingKey'`

There are similar properties used when declaring a dead-letter exchange/queue, when `autoBindDlq` is `true`.

22.5. Retry With the RabbitMQ Binder

When retry is enabled within the binder, the listener container thread is suspended for any back off periods that are configured. This might be important when strict ordering is required with a single consumer. However, for other use cases, it prevents other messages from being processed on that thread. An alternative to using binder retry is to set up dead lettering with time to live on the dead-letter queue (DLQ) as well as dead-letter configuration on the DLQ itself. See “[RabbitMQ Binder Properties](#)” for more information about the properties discussed here. You can use the following example configuration to enable this feature:

- Set `autoBindDlq` to `true`. The binder create a DLQ. Optionally, you can specify a name in `deadLetterQueueName`.
- Set `dlqTtl` to the back off time you want to wait between redeliveries.
- Set the `dlqDeadLetterExchange` to the default exchange. Expired messages from the DLQ are routed to the original queue, because the default `deadLetterRoutingKey` is the queue name (`destination.group`). Setting to the default exchange is achieved by setting the property with no value, as shown in the next example.

To force a message to be dead-lettered, either throw an `AmqpRejectAndDontRequeueException` or set `requeueRejected` to `true` (the default) and throw any exception.

The loop continues without end, which is fine for transient problems, but you may want to give up after some number of attempts. Fortunately, RabbitMQ provides the `x-death` header, which lets you determine how many cycles have occurred.

To acknowledge a message after giving up, throw an `ImmediateAcknowledgeAmqpException`.

22.5.1. Putting it All Together

The following configuration creates an exchange `myDestination` with queue `myDestination.consumerGroup` bound to a topic exchange with a wildcard routing key #:

```
---
spring.cloud.stream.bindings.input.destination=myDestination
spring.cloud.stream.bindings.input.group=consumerGroup
#disable binder retries
spring.cloud.stream.bindings.input.consumer.max-attempts=1
#dlx/dlq setup
spring.cloud.stream.rabbit.bindings.input.consumer.auto-bind-dlq=true
spring.cloud.stream.rabbit.bindings.input.consumer.dlq-ttl=5000
spring.cloud.stream.rabbit.bindings.input.consumer.dlq-dead-letter-exchange=
---
```

This configuration creates a DLQ bound to a direct exchange (DLX) with a routing key of `myDestination.consumerGroup`. When messages are rejected, they are routed to the DLQ. After 5 seconds, the message expires and is routed to the original queue by using the queue name as the routing key, as shown in the following example:

```
@SpringBootApplication
@EnableBinding(Sink.class)
public class XDeathApplication {

    public static void main(String[] args) {
        SpringApplication.run(XDeathApplication.class, args);
    }

    @StreamListener(Sink.INPUT)
    public void listen(String in, @Header(name = "x-death", required = false) Map<?,?>
death) {
        if (death != null && death.get("count").equals(3L)) {
            // giving up - don't send to DLX
            throw new ImmediateAcknowledgeAmqpException("Failed after 4 attempts");
        }
        throw new AmqpRejectAndDontRequeueException("failed");
    }
}
```

Notice that the count property in the `x-death` header is a `Long`.

22.6. Error Channels

Starting with version 1.3, the binder unconditionally sends exceptions to an error channel for each consumer destination and can also be configured to send async producer send failures to an error channel. See “[Error Handling](#)” for more information.

RabbitMQ has two types of send failures:

- Returned messages,
- Negatively acknowledged [Publisher Confirms](#).

The latter is rare. According to the RabbitMQ documentation “[A nack] will only be delivered if an internal error occurs in the Erlang process responsible for a queue.”.

As well as enabling producer error channels (as described in “[Error Handling](#)”), the RabbitMQ binder only sends messages to the channels if the connection factory is appropriately configured, as follows:

- `ccf.setPublisherConfirms(true);`
- `ccf.setPublisherReturns(true);`

When using Spring Boot configuration for the connection factory, set the following properties:

- `spring.rabbitmq.publisher-confirms`
- `spring.rabbitmq.publisher-returns`

The payload of the `ErrorMessage` for a returned message is a `ReturnedAmqpMessageException` with the following properties:

- `failedMessage`: The spring-messaging `Message<?>` that failed to be sent.
- `amqpMessage`: The raw spring-amqp `Message`.
- `replyCode`: An integer value indicating the reason for the failure (for example, 312 - No route).
- `replyText`: A text value indicating the reason for the failure (for example, `NO_ROUTE`).
- `exchange`: The exchange to which the message was published.
- `routingKey`: The routing key used when the message was published.

For negatively acknowledged confirmations, the payload is a `NackedAmqpMessageException` with the following properties:

- `failedMessage`: The spring-messaging `Message<?>` that failed to be sent.
- `nackReason`: A reason (if available—you may need to examine the broker logs for more information).

There is no automatic handling of these exceptions (such as sending to a [dead-letter queue](#)). You can consume these exceptions with your own Spring Integration flow.

22.7. Dead-Letter Queue Processing

Because you cannot anticipate how users would want to dispose of dead-lettered messages, the framework does not provide any standard mechanism to handle them. If the reason for the dead-lettering is transient, you may wish to route the messages back to the original queue. However, if the problem is a permanent issue, that could cause an infinite loop. The following Spring Boot application shows an example of how to route those messages back to the original queue but moves them to a third “parking lot” queue after three attempts. The second example uses the [RabbitMQ Delayed Message Exchange](#) to introduce a delay to the re-queued message. In this example, the delay increases for each attempt. These examples use a `@RabbitListener` to receive messages from the DLQ. You could also use `RabbitTemplate.receive()` in a batch process.

The examples assume the original destination is `so8400in` and the consumer group is `so8400`.

22.7.1. Non-Partitioned Destinations

The first two examples are for when the destination is **not** partitioned:

```

@SpringBootApplication
public class ReRouteDlqApplication {

    private static final String ORIGINAL_QUEUE = "so8400in.so8400";

    private static final String DLQ = ORIGINAL_QUEUE + ".dlq";

    private static final String PARKING_LOT = ORIGINAL_QUEUE + ".parkingLot";

    private static final String X_RETRIES_HEADER = "x-retries";

    public static void main(String[] args) throws Exception {
        ConfigurableApplicationContext context =
SpringApplication.run(ReRouteDlqApplication.class, args);
        System.out.println("Hit enter to terminate");
        System.in.read();
        context.close();
    }

    @Autowired
    private RabbitTemplate rabbitTemplate;

    @RabbitListener(queues = DLQ)
    public void rePublish(Message failedMessage) {
        Integer retriesHeader = (Integer)
failedMessage.getMessageProperties().getHeaders().get(X_RETRIES_HEADER);
        if (retriesHeader == null) {
            retriesHeader = Integer.valueOf(0);
        }
        if (retriesHeader < 3) {
            failedMessage.getMessageProperties().getHeaders().put(X_RETRIES_HEADER,
retriesHeader + 1);
            this.rabbitTemplate.send(ORIGINAL_QUEUE, failedMessage);
        }
        else {
            this.rabbitTemplate.send(PARKING_LOT, failedMessage);
        }
    }

    @Bean
    public Queue parkingLot() {
        return new Queue(PARKING_LOT);
    }
}

```

```

@SpringBootApplication
public class ReRouteDlqApplication {

```

```

private static final String ORIGINAL_QUEUE = "so8400in.so8400";

private static final String DLQ = ORIGINAL_QUEUE + ".dlq";

private static final String PARKING_LOT = ORIGINAL_QUEUE + ".parkingLot";

private static final String X_RETRIES_HEADER = "x-retries";

private static final String DELAY_EXCHANGE = "dlqReRouter";

public static void main(String[] args) throws Exception {
    ConfigurableApplicationContext context =
SpringApplication.run(ReRouteDlqApplication.class, args);
    System.out.println("Hit enter to terminate");
    System.in.read();
    context.close();
}

@Autowired
private RabbitTemplate rabbitTemplate;

@RabbitListener(queues = DLQ)
public void rePublish(Message failedMessage) {
    Map<String, Object> headers =
failedMessage.getMessageProperties().getHeaders();
    Integer retriesHeader = (Integer) headers.get(X_RETRIES_HEADER);
    if (retriesHeader == null) {
        retriesHeader = Integer.valueOf(0);
    }
    if (retriesHeader < 3) {
        headers.put(X_RETRIES_HEADER, retriesHeader + 1);
        headers.put("x-delay", 5000 * retriesHeader);
        this.rabbitTemplate.send(DELAY_EXCHANGE, ORIGINAL_QUEUE, failedMessage);
    }
    else {
        this.rabbitTemplate.send(PARKING_LOT, failedMessage);
    }
}

@Bean
public DirectExchange delayExchange() {
    DirectExchange exchange = new DirectExchange(DELAY_EXCHANGE);
    exchange.setDelayed(true);
    return exchange;
}

@Bean
public Binding bindOriginalToDelay() {
    return BindingBuilder.bind(new
Queue(ORIGINAL_QUEUE)).to(delayExchange()).with(ORIGINAL_QUEUE);
}

```



```

@Bean
public Queue parkingLot() {
    return new Queue(PARKING_LOT);
}
}

```

22.7.2. Partitioned Destinations

With partitioned destinations, there is one DLQ for all partitions. We determine the original queue from the headers.

`republishToDlq=false`

When `republishToDlq` is `false`, RabbitMQ publishes the message to the DLX/DLQ with an `x-death` header containing information about the original destination, as shown in the following example:

```

@SpringBootApplication
public class ReRouteDlqApplication {

    private static final String ORIGINAL_QUEUE = "so8400in.so8400";

    private static final String DLQ = ORIGINAL_QUEUE + ".dlq";

    private static final String PARKING_LOT = ORIGINAL_QUEUE + ".parkingLot";

    private static final String X_DEATH_HEADER = "x-death";

    private static final String X_RETRIES_HEADER = "x-retries";

    public static void main(String[] args) throws Exception {
        ConfigurableApplicationContext context =
        SpringApplication.run(ReRouteDlqApplication.class, args);
        System.out.println("Hit enter to terminate");
        System.in.read();
        context.close();
    }

    @Autowired
    private RabbitTemplate rabbitTemplate;

    @SuppressWarnings("unchecked")
    @RabbitListener(queues = DLQ)
    public void rePublish(Message failedMessage) {
        Map<String, Object> headers =
        failedMessage.getMessageProperties().getHeaders();
        Integer retriesHeader = (Integer) headers.get(X_RETRIES_HEADER);
        if (retriesHeader == null) {
            retriesHeader = Integer.valueOf(0);
        }
    }
}

```

```

    }
    if (retriesHeader < 3) {
        headers.put(X_RETRIES_HEADER, retriesHeader + 1);
        List<Map<String, ?>> xDeath = (List<Map<String, ?>>)
headers.get(X_DEATH_HEADER);
        String exchange = (String) xDeath.get(0).get("exchange");
        List<String> routingKeys = (List<String>) xDeath.get(0).get("routing-
keys");
        this.rabbitTemplate.send(exchange, routingKeys.get(0), failedMessage);
    }
    else {
        this.rabbitTemplate.send(PARKING_LOT, failedMessage);
    }
}

@Bean
public Queue parkingLot() {
    return new Queue(PARKING_LOT);
}
}

```

republishToDlq=true

When `republishToDlq` is `true`, the republishing recoverer adds the original exchange and routing key to headers, as shown in the following example:

```

@SpringBootApplication
public class ReRouteDlqApplication {

    private static final String ORIGINAL_QUEUE = "so8400in.so8400";

    private static final String DLQ = ORIGINAL_QUEUE + ".dlq";

    private static final String PARKING_LOT = ORIGINAL_QUEUE + ".parkingLot";

    private static final String X_RETRIES_HEADER = "x-retries";

    private static final String X_ORIGINAL_EXCHANGE_HEADER =
RepublishMessageRecoverer.X_ORIGINAL_EXCHANGE;

    private static final String X_ORIGINAL_ROUTING_KEY_HEADER =
RepublishMessageRecoverer.X_ORIGINAL_ROUTING_KEY;

    public static void main(String[] args) throws Exception {
        ConfigurableApplicationContext context =
SpringApplication.run(ReRouteDlqApplication.class, args);
        System.out.println("Hit enter to terminate");
        System.in.read();
        context.close();
    }
}

```

```

}

@Autowired
private RabbitTemplate rabbitTemplate;

@RabbitListener(queues = DLQ)
public void rePublish(Message failedMessage) {
    Map<String, Object> headers =
failedMessage.getMessageProperties().getHeaders();
    Integer retriesHeader = (Integer) headers.get(X_RETRIES_HEADER);
    if (retriesHeader == null) {
        retriesHeader = Integer.valueOf(0);
    }
    if (retriesHeader < 3) {
        headers.put(X_RETRIES_HEADER, retriesHeader + 1);
        String exchange = (String) headers.get(X_ORIGINAL_EXCHANGE_HEADER);
        String originalRoutingKey = (String)
headers.get(X_ORIGINAL_ROUTING_KEY_HEADER);
        this.rabbitTemplate.send(exchange, originalRoutingKey, failedMessage);
    }
    else {
        this.rabbitTemplate.send(PARKING_LOT, failedMessage);
    }
}

@Bean
public Queue parkingLot() {
    return new Queue(PARKING_LOT);
}
}

```

22.8. Partitioning with the RabbitMQ Binder

RabbitMQ does not support partitioning natively.

Sometimes, it is advantageous to send data to specific partitions—for example, when you want to strictly order message processing, all messages for a particular customer should go to the same partition.

The `RabbitMessageChannelBinder` provides partitioning by binding a queue for each partition to the destination exchange.

The following Java and YAML examples show how to configure the producer:

Producer

```
@SpringBootApplication
@EnableBinding(Source.class)
public class RabbitPartitionProducerApplication {

    private static final Random RANDOM = new Random(System.currentTimeMillis());

    private static final String[] data = new String[] {
        "abc1", "def1", "qux1",
        "abc2", "def2", "qux2",
        "abc3", "def3", "qux3",
        "abc4", "def4", "qux4",
    };

    public static void main(String[] args) {
        new SpringApplicationBuilder(RabbitPartitionProducerApplication.class)
            .web(false)
            .run(args);
    }

    @InboundChannelAdapter(channel = Source.OUTPUT, poller = @Poller(fixedRate =
"5000"))
    public Message<?> generate() {
        String value = data[RANDOM.nextInt(data.length)];
        System.out.println("Sending: " + value);
        return MessageBuilder.withPayload(value)
            .setHeader("partitionKey", value)
            .build();
    }
}
```

application.yml

```
spring:
  cloud:
    stream:
      bindings:
        output:
          destination: partitioned.destination
          producer:
            partitioned: true
            partition-key-expression: headers['partitionKey']
            partition-count: 2
            required-groups:
              - myGroup
```



The configuration in the preceding example uses the default partitioning (`key.hashCode() % partitionCount`). This may or may not provide a suitably balanced algorithm, depending on the key values. You can override this default by using the `partitionSelectorExpression` or `partitionSelectorClass` properties.

The `required-groups` property is required only if you need the consumer queues to be provisioned when the producer is deployed. Otherwise, any messages sent to a partition are lost until the corresponding consumer is deployed.

The following configuration provisions a topic exchange:

partitioned.destination	topic	
--------------------------------	-------	--

The following queues are bound to that exchange:

partitioned.destination.myGroup-0	
partitioned.destination.myGroup-1	

The following bindings associate the queues to the exchange:

▼ Bindings

This exchange

⇓

To	Routing key	Arguments	
partitioned.destination.myGroup-0	partitioned.destination-0		
partitioned.destination.myGroup-1	partitioned.destination-1		

The following Java and YAML examples continue the previous examples and show how to configure the consumer:

Consumer

```
@SpringBootApplication
@EnableBinding(Sink.class)
public class RabbitPartitionConsumerApplication {

    public static void main(String[] args) {
        new SpringApplicationBuilder(RabbitPartitionConsumerApplication.class)
            .web(false)
            .run(args);
    }

    @StreamListener(Sink.INPUT)
    public void listen(@Payload String in, @Header(AmqpHeaders.CONSUMER_QUEUE) String
queue) {
        System.out.println(in + " received from queue " + queue);
    }
}
```

application.yml

```
spring:
  cloud:
    stream:
      bindings:
        input:
          destination: partitioned.destination
          group: myGroup
          consumer:
            partitioned: true
            instance-index: 0
```



The `RabbitMessageChannelBinder` does not support dynamic scaling. There must be at least one consumer per partition. The consumer's `instanceIndex` is used to indicate which partition is consumed. Platforms such as Cloud Foundry can have only one instance with an `instanceIndex`.

Chapter 23. Appendices

Appendix B: Building

23.B.1. Basic Compile and Test

To build the source you will need to install JDK 1.8.

The build uses the Maven wrapper so you don't have to install a specific version of Maven. To enable the tests, you should have RabbitMQ server running on localhost and the default port (5672) before building.

The main build command is

```
$ ./mvnw clean install
```

You can also add '-DskipTests' if you like, to avoid running the tests.



You can also install Maven (>=3.3.3) yourself and run the `mvn` command in place of `./mvnw` in the examples below. If you do that you also might need to add `-P spring` if your local Maven settings do not contain repository declarations for spring pre-release artifacts.



Be aware that you might need to increase the amount of memory available to Maven by setting a `MAVEN_OPTS` environment variable with a value like `-Xmx512m -XX:MaxPermSize=128m`. We try to cover this in the `.mvn` configuration, so if you find you have to do it to make a build succeed, please raise a ticket to get the settings added to source control.

The projects that require middleware generally include a `docker-compose.yml`, so consider using [Docker Compose](#) to run the middleware servers in Docker containers.

23.B.2. Documentation

There is a "docs" profile that will generate documentation.

```
./mvnw clean package -Pdocs -DskipTests
```

The reference documentation can then be found in `docs/target/contents/reference`.

23.B.3. Working with the code

If you don't have an IDE preference we would recommend that you use [Spring Tools Suite](#) or [Eclipse](#) when working with the code. We use the [m2eclipse](#) eclipse plugin for maven support. Other IDEs and tools should also work without issue.

Importing into eclipse with m2eclipse

We recommend the [m2eclipse](#) eclipse plugin when working with eclipse. If you don't already have m2eclipse installed it is available from the "eclipse marketplace".

Unfortunately m2e does not yet support Maven 3.3, so once the projects are imported into Eclipse you will also need to tell m2eclipse to use the `.settings.xml` file for the projects. If you do not do this you may see many different errors related to the POMs in the projects. Open your Eclipse preferences, expand the Maven preferences, and select User Settings. In the User Settings field click Browse and navigate to the Spring Cloud project you imported selecting the `.settings.xml` file in that project. Click Apply and then OK to save the preference changes.



Alternatively you can copy the repository settings from `.settings.xml` into your own `~/.m2/settings.xml`.

Importing into eclipse without m2eclipse

If you prefer not to use m2eclipse you can generate eclipse project metadata using the following command:

```
$ ./mvnw eclipse:eclipse
```

The generated eclipse projects can be imported by selecting `import existing projects` from the `file` menu.

Appendix C: Contributing

Spring Cloud is released under the non-restrictive Apache 2.0 license, and follows a very standard Github development process, using Github tracker for issues and merging pull requests into master. If you want to contribute even something trivial please do not hesitate, but follow the guidelines below.

23.C.1. Sign the Contributor License Agreement

Before we accept a non-trivial patch or pull request we will need you to sign the [contributor's agreement](#). Signing the contributor's agreement does not grant anyone commit rights to the main repository, but it does mean that we can accept your contributions, and you will get an author credit if we do. Active contributors might be asked to join the core team, and given the ability to merge pull requests.

23.C.2. Code Conventions and Housekeeping

None of these is essential for a pull request, but they will all help. They can also be added after the original pull request but before a merge.

- Use the Spring Framework code format conventions. If you use Eclipse you can import formatter settings using the `eclipse-code-formatter.xml` file from the [Spring Cloud Build](#) project. If using IntelliJ, you can use the [Eclipse Code Formatter Plugin](#) to import the same file.

- Make sure all new `.java` files to have a simple Javadoc class comment with at least an `@author` tag identifying you, and preferably at least a paragraph on what the class is for.
- Add the ASF license header comment to all new `.java` files (copy from existing files in the project)
- Add yourself as an `@author` to the `.java` files that you modify substantially (more than cosmetic changes).
- Add some Javadocs and, if you change the namespace, some XSD doc elements.
- A few unit tests would help a lot as well — someone has to do it.
- If no-one else is using your branch, please rebase it against the current master (or other target branch in the main project).
- When writing a commit message please follow [these conventions](#), if you are fixing an existing issue please add `Fixes gh-XXXX` at the end of the commit message (where XXXX is the issue number).

Chapter 24. Spring Cloud Stream Kafka Binder Reference Guide

Sabby Anandan, Marius Bogoevici, Eric Bottard, Mark Fisher, Ilayaperumal Gopinathan, Gunnar Hillert, Mark Pollack, Patrick Peralta, Glenn Renfro, Thomas Risberg, Dave Syer, David Turanski, Janne Valkealahti, Benjamin Klein, Henryk Konsek, Gary Russell, Arnaud Jardiné, Soby Chacko

:doctype: book :toc: :toclevels: 4 :source-highlighter: prettify :numbered: :icons: font :hide-uri-scheme: :spring-cloud-stream-binder-kafka-repo: snapshot :github-tag: master :spring-cloud-stream-binder-kafka-docs-version: current :spring-cloud-stream-binder-kafka-docs: docs.spring.io/spring-cloud-stream-binder-kafka/docs/{spring-cloud-stream-binder-kafka-docs-version}/reference :spring-cloud-stream-binder-kafka-docs-current: docs.spring.io/spring-cloud-stream-binder-kafka/docs/current-SNAPSHOT/reference/html/ :github-repo: spring-cloud/spring-cloud-stream-binder-kafka :github-raw: raw.githubusercontent.com/spring-cloud/spring-cloud-stream-binder-kafka/master :github-code: github.com/spring-cloud/spring-cloud-stream-binder-kafka/tree/master :github-wiki: github.com/spring-cloud/spring-cloud-stream-binder-kafka/wiki :github-master-code: github.com/spring-cloud/spring-cloud-stream-binder-kafka/tree/master :sc-ext: java

{spring-cloud-stream-version}

Chapter 25. Reference Guide

This guide describes the Apache Kafka implementation of the Spring Cloud Stream Binder. It contains information about its design, usage, and configuration options, as well as information on how the Stream Cloud Stream concepts map onto Apache Kafka specific constructs. In addition, this guide explains the Kafka Streams binding capabilities of Spring Cloud Stream.

25.1. Apache Kafka Binder

25.1.1. Usage

To use Apache Kafka binder, you need to add `spring-cloud-stream-binder-kafka` as a dependency to your Spring Cloud Stream application, as shown in the following example for Maven:

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-stream-binder-kafka</artifactId>
</dependency>
```

Alternatively, you can also use the Spring Cloud Stream Kafka Starter, as shown in the following example for Maven:

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-stream-kafka</artifactId>
</dependency>
```

25.1.2. Overview

The following image shows a simplified diagram of how the Apache Kafka binder operates:

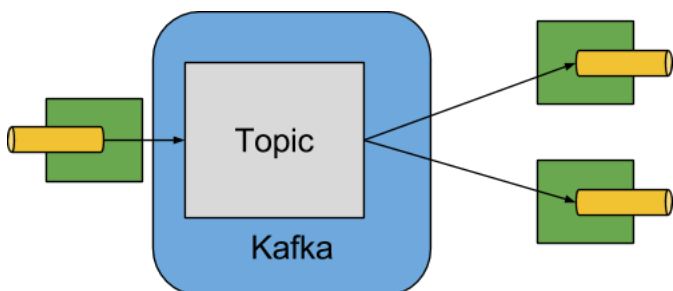


Figure 11. Kafka Binder

The Apache Kafka Binder implementation maps each destination to an Apache Kafka topic. The consumer group maps directly to the same Apache Kafka concept. Partitioning also maps directly to Apache Kafka partitions as well.

The binder currently uses the Apache Kafka `kafka-clients` version `2.3.1`. This client can communicate with older brokers (see the Kafka documentation), but certain features may not be

available. For example, with versions earlier than 0.11.x.x, native headers are not supported. Also, 0.11.x.x does not support the `autoAddPartitions` property.

25.1.3. Configuration Options

This section contains the configuration options used by the Apache Kafka binder.

For common configuration options and properties pertaining to binder, see the [core documentation](#).

Kafka Binder Properties

`spring.cloud.stream.kafka.binder.brokers`

A list of brokers to which the Kafka binder connects.

Default: `localhost`.

`spring.cloud.stream.kafka.binder.defaultBrokerPort`

`brokers` allows hosts specified with or without port information (for example, `host1,host2:port2`). This sets the default port when no port is configured in the broker list.

Default: `9092`.

`spring.cloud.stream.kafka.binder.configuration`

Key/Value map of client properties (both producers and consumer) passed to all clients created by the binder. Due to the fact that these properties are used by both producers and consumers, usage should be restricted to common properties—for example, security settings. Unknown Kafka producer or consumer properties provided through this configuration are filtered out and not allowed to propagate. Properties here supersede any properties set in boot.

Default: Empty map.

`spring.cloud.stream.kafka.binder.consumerProperties`

Key/Value map of arbitrary Kafka client consumer properties. In addition to support known Kafka consumer properties, unknown consumer properties are allowed here as well. Properties here supersede any properties set in boot and in the `configuration` property above.

Default: Empty map.

`spring.cloud.stream.kafka.binder.headers`

The list of custom headers that are transported by the binder. Only required when communicating with older applications ($\leq 1.3.x$) with a `kafka-clients` version $< 0.11.0.0$. Newer versions support headers natively.

Default: empty.

`spring.cloud.stream.kafka.binder.healthTimeout`

The time to wait to get partition information, in seconds. Health reports as down if this timer expires.

Default: 10.

spring.cloud.stream.kafka.binder.requiredAcks

The number of required acks on the broker. See the Kafka documentation for the producer `acks` property.

Default: 1.

spring.cloud.stream.kafka.binder.minPartitionCount

Effective only if `autoCreateTopics` or `autoAddPartitions` is set. The global minimum number of partitions that the binder configures on topics on which it produces or consumes data. It can be superseded by the `partitionCount` setting of the producer or by the value of `instanceCount * concurrency` settings of the producer (if either is larger).

Default: 1.

spring.cloud.stream.kafka.binder.producerProperties

Key/Value map of arbitrary Kafka client producer properties. In addition to support known Kafka producer properties, unknown producer properties are allowed here as well. Properties here supersede any properties set in boot and in the `configuration` property above.

Default: Empty map.

spring.cloud.stream.kafka.binder.replicationFactor

The replication factor of auto-created topics if `autoCreateTopics` is active. Can be overridden on each binding.

Default: 1.

spring.cloud.stream.kafka.binder.autoCreateTopics

If set to `true`, the binder creates new topics automatically. If set to `false`, the binder relies on the topics being already configured. In the latter case, if the topics do not exist, the binder fails to start.



This setting is independent of the `auto.create.topics.enable` setting of the broker and does not influence it. If the server is set to auto-create topics, they may be created as part of the metadata retrieval request, with default broker settings.

Default: `true`.

spring.cloud.stream.kafka.binder.autoAddPartitions

If set to `true`, the binder creates new partitions if required. If set to `false`, the binder relies on the partition size of the topic being already configured. If the partition count of the target topic is smaller than the expected value, the binder fails to start.

Default: `false`.

spring.cloud.stream.kafka.binder.transaction.transactionIdPrefix

Enables transactions in the binder. See `transaction.id` in the Kafka documentation and [Transactions](#) in the `spring-kafka` documentation. When transactions are enabled, individual `producer` properties are ignored and all producers use the `spring.cloud.stream.kafka.binder.transaction.producer.*` properties.

Default `null` (no transactions)

spring.cloud.stream.kafka.binder.transaction.producer.*

Global producer properties for producers in a transactional binder. See `spring.cloud.stream.kafka.binder.transaction.transactionIdPrefix` and [Kafka Producer Properties](#) and the general producer properties supported by all binders.

Default: See individual producer properties.

spring.cloud.stream.kafka.binder.headerMapperBeanName

The bean name of a `KafkaHeaderMapper` used for mapping `spring-messaging` headers to and from Kafka headers. Use this, for example, if you wish to customize the trusted packages in a `BinderHeaderMapper` bean that uses JSON deserialization for the headers. If this custom `BinderHeaderMapper` bean is not made available to the binder using this property, then the binder will look for a header mapper bean with the name `kafkaBinderHeaderMapper` that is of type `BinderHeaderMapper` before falling back to a default `BinderHeaderMapper` created by the binder.

Default: none.

Kafka Consumer Properties



To avoid repetition, Spring Cloud Stream supports setting values for all channels, in the format of `spring.cloud.stream.kafka.default.consumer.<property>=<value>`.

The following properties are available for Kafka consumers only and must be prefixed with `spring.cloud.stream.kafka.bindings.<channelName>.consumer..`

admin.configuration

Since version 2.1.1, this property is deprecated in favor of `topic.properties`, and support for it will be removed in a future version.

admin.replicas-assignment

Since version 2.1.1, this property is deprecated in favor of `topic.replicas-assignment`, and support for it will be removed in a future version.

admin.replication-factor

Since version 2.1.1, this property is deprecated in favor of `topic.replication-factor`, and support for it will be removed in a future version.

autoRebalanceEnabled

When `true`, topic partitions is automatically rebalanced between the members of a consumer group. When `false`, each consumer is assigned a fixed set of partitions based on `spring.cloud.stream.instanceCount` and `spring.cloud.stream.instanceIndex`. This requires both the `spring.cloud.stream.instanceCount` and `spring.cloud.stream.instanceIndex` properties to be

set appropriately on each launched instance. The value of the `spring.cloud.stream.instanceCount` property must typically be greater than 1 in this case.

Default: `true`.

ackEachRecord

When `autoCommitOffset` is `true`, this setting dictates whether to commit the offset after each record is processed. By default, offsets are committed after all records in the batch of records returned by `consumer.poll()` have been processed. The number of records returned by a poll can be controlled with the `max.poll.records` Kafka property, which is set through the consumer `configuration` property. Setting this to `true` may cause a degradation in performance, but doing so reduces the likelihood of redelivered records when a failure occurs. Also, see the binder `requiredAcks` property, which also affects the performance of committing offsets.

Default: `false`.

autoCommitOffset

Whether to autocommit offsets when a message has been processed. If set to `false`, a header with the key `kafka_acknowledgment` of the type `org.springframework.kafka.support.Acknowledgment` header is present in the inbound message. Applications may use this header for acknowledging messages. See the examples section for details. When this property is set to `false`, Kafka binder sets the `ack` mode to `org.springframework.kafka.listener.AbstractMessageListenerContainer.AckMode.MANUAL` and the application is responsible for acknowledging records. Also see `ackEachRecord`.

Default: `true`.

autoCommitOnError

Effective only if `autoCommitOffset` is set to `true`. If set to `false`, it suppresses auto-commits for messages that result in errors and commits only for successful messages. It allows a stream to automatically replay from the last successfully processed message, in case of persistent failures. If set to `true`, it always auto-commits (if auto-commit is enabled). If not set (the default), it effectively has the same value as `enableDlq`, auto-committing erroneous messages if they are sent to a DLQ and not committing them otherwise.

Default: not set.

resetOffsets

Whether to reset offsets on the consumer to the value provided by `startOffset`. Must be false if a `KafkaRebalanceListener` is provided; see [Using a KafkaRebalanceListener](#).

Default: `false`.

startOffset

The starting offset for new groups. Allowed values: `earliest` and `latest`. If the consumer group is set explicitly for the consumer 'binding' (through `spring.cloud.stream.bindings.<channelName>.group`), 'startOffset' is set to `earliest`. Otherwise, it is set to `latest` for the `anonymous` consumer group. Also see `resetOffsets` (earlier in this list).

Default: null (equivalent to `earliest`).

enableDlq

When set to true, it enables DLQ behavior for the consumer. By default, messages that result in errors are forwarded to a topic named `error.<destination>.<group>`. The DLQ topic name can be configurable by setting the `dlqName` property. This provides an alternative option to the more common Kafka replay scenario for the case when the number of errors is relatively small and replaying the entire original topic may be too cumbersome. See [Dead-Letter Topic Processing](#) for more information. Starting with version 2.0, messages sent to the DLQ topic are enhanced with the following headers: `x-original-topic`, `x-exception-message`, and `x-exception-stacktrace` as `byte[]`. By default, a failed record is sent to the same partition number in the DLQ topic as the original record. See [Dead-Letter Topic Partition Selection](#) for how to change that behavior. **Not allowed when `destinationIsPattern` is true.**

Default: `false`.

dlqPartitions

When `enableDlq` is true, and this property is not set, a dead letter topic with the same number of partitions as the primary topic(s) is created. Usually, dead-letter records are sent to the same partition in the dead-letter topic as the original record. This behavior can be changed; see [Dead-Letter Topic Partition Selection](#). If this property is set to `1` and there is no `DlqPartitionFunction` bean, all dead-letter records will be written to partition `0`. If this property is greater than `1`, you **MUST** provide a `DlqPartitionFunction` bean. Note that the actual partition count is affected by the binder's `minPartitionCount` property.

Default: `none`

configuration

Map with a key/value pair containing generic Kafka consumer properties. In addition to having Kafka consumer properties, other configuration properties can be passed here. For example some properties needed by the application such as `spring.cloud.stream.kafka.bindings.input.consumer.configuration.foo=bar`.

Default: Empty map.

dlqName

The name of the DLQ topic to receive the error messages.

Default: null (If not specified, messages that result in errors are forwarded to a topic named `error.<destination>.<group>`).

dlqProducerProperties

Using this, DLQ-specific producer properties can be set. All the properties available through kafka producer properties can be set through this property. When native decoding is enabled on the consumer (i.e., `useNativeDecoding: true`), the application must provide corresponding key/value serializers for DLQ. This must be provided in the form of `dlqProducerProperties.configuration.key.serializer` and `dlqProducerProperties.configuration.value.serializer`.

Default: Default Kafka producer properties.

standardHeaders

Indicates which standard headers are populated by the inbound channel adapter. Allowed values: `none`, `id`, `timestamp`, or `both`. Useful if using native deserialization and the first component to receive a message needs an `id` (such as an aggregator that is configured to use a JDBC message store).

Default: `none`

converterBeanName

The name of a bean that implements `RecordMessageConverter`. Used in the inbound channel adapter to replace the default `MessagingMessageConverter`.

Default: `null`

idleEventInterval

The interval, in milliseconds, between events indicating that no messages have recently been received. Use an `ApplicationListener<ListenerContainerIdleEvent>` to receive these events. See [Example: Pausing and Resuming the Consumer](#) for a usage example.

Default: `30000`

destinationIsPattern

When true, the destination is treated as a regular expression `Pattern` used to match topic names by the broker. When true, topics are not provisioned, and `enableDlq` is not allowed, because the binder does not know the topic names during the provisioning phase. Note, the time taken to detect new topics that match the pattern is controlled by the consumer property `metadata.max.age.ms`, which (at the time of writing) defaults to 300,000ms (5 minutes). This can be configured using the `configuration` property above.

Default: `false`

topic.properties

A `Map` of Kafka topic properties used when provisioning new topics—for example, `spring.cloud.stream.kafka.bindings.input.consumer.topic.properties.message.format.version=0.9.0.0`

Default: none.

topic.replicas-assignment

A `Map<Integer, List<Integer>>` of replica assignments, with the key being the partition and the value being the assignments. Used when provisioning new topics. See the `NewTopic` Javadocs in the `kafka-clients` jar.

Default: none.

topic.replication-factor

The replication factor to use when provisioning topics. Overrides the binder-wide setting. Ignored if `replicas-assignments` is present.

Default: none (the binder-wide default of 1 is used).

pollTimeout

Timeout used for polling in pollable consumers.

Default: 5 seconds.

transactionManager

Bean name of a `KafkaAwareTransactionManager` used to override the binder's transaction manager for this binding. Usually needed if you want to synchronize another transaction with the Kafka transaction, using the `ChainedKafkaTransactionManager`. To achieve exactly once consumption and production of records, the consumer and producer bindings must all be configured with the same transaction manager.

Default: none.

Consuming Batches

Starting with version 3.0, when `spring.cloud.stream.binding.<name>.consumer.batch-mode` is set to `true`, all of the records received by polling the Kafka `Consumer` will be presented as a `List<?>` to the listener method. Otherwise, the method will be called with one record at a time. The size of the batch is controlled by Kafka consumer properties `max.poll.records`, `min.fetch.bytes`, `fetch.max.wait.ms`; refer to the Kafka documentation for more information.



Retry within the binder is not supported when using batch mode, so `maxAttempts` will be overridden to 1. You can configure a `SeekToCurrentBatchErrorHandler` (using a `ListenerContainerCustomizer`) to achieve similar functionality to retry in the binder. You can also use a manual `AckMode` and call `Acknowledgment.nack(index, sleep)` to commit the offsets for a partial batch and have the remaining records redelivered. Refer to the [Spring for Apache Kafka documentation](#) for more information about these techniques.

Kafka Producer Properties



To avoid repetition, Spring Cloud Stream supports setting values for all channels, in the format of `spring.cloud.stream.kafka.default.producer.<property>=<value>`.

The following properties are available for Kafka producers only and must be prefixed with `spring.cloud.stream.kafka.bindings.<channelName>.producer..`

admin.configuration

Since version 2.1.1, this property is deprecated in favor of `topic.properties`, and support for it will be removed in a future version.

admin.replicas-assignment

Since version 2.1.1, this property is deprecated in favor of `topic.replicas-assignment`, and support for it will be removed in a future version.

admin.replication-factor

Since version 2.1.1, this property is deprecated in favor of `topic.replication-factor`, and support for it will be removed in a future version.

bufferSize

Upper limit, in bytes, of how much data the Kafka producer attempts to batch before sending.

Default: `16384`.

sync

Whether the producer is synchronous.

Default: `false`.

sendTimeoutExpression

A SpEL expression evaluated against the outgoing message used to evaluate the time to wait for ack when synchronous publish is enabled — for example, `headers['mySendTimeout']`. The value of the timeout is in milliseconds. With versions before 3.0, the payload could not be used unless native encoding was being used because, by the time this expression was evaluated, the payload was already in the form of a `byte[]`. Now, the expression is evaluated before the payload is converted.

Default: `none`.

batchTimeout

How long the producer waits to allow more messages to accumulate in the same batch before sending the messages. (Normally, the producer does not wait at all and simply sends all the messages that accumulated while the previous send was in progress.) A non-zero value may increase throughput at the expense of latency.

Default: `0`.

messageKeyExpression

A SpEL expression evaluated against the outgoing message used to populate the key of the produced Kafka message — for example, `headers['myKey']`. With versions before 3.0, the payload could not be used unless native encoding was being used because, by the time this expression was evaluated, the payload was already in the form of a `byte[]`. Now, the expression is evaluated before the payload is converted.

Default: `none`.

headerPatterns

A comma-delimited list of simple patterns to match Spring messaging headers to be mapped to the Kafka `Headers` in the `ProducerRecord`. Patterns can begin or end with the wildcard character (asterisk). Patterns can be negated by prefixing with `!`. Matching stops after the first match (positive or negative). For example `!ask,as*` will pass `ash` but not `ask`. `id` and `timestamp` are never mapped.

Default: `*` (all headers - except the `id` and `timestamp`)

configuration

Map with a key/value pair containing generic Kafka producer properties.

Default: Empty map.

topic.properties

A `Map` of Kafka topic properties used when provisioning new topics—for example, `spring.cloud.stream.kafka.bindings.output.producer.topic.properties.message.format.version=0.9.0.0`

topic.replicas-assignment

A `Map<Integer, List<Integer>>` of replica assignments, with the key being the partition and the value being the assignments. Used when provisioning new topics. See the `NewTopic` Javadocs in the `kafka-clients` jar.

Default: none.

topic.replication-factor

The replication factor to use when provisioning topics. Overrides the binder-wide setting. Ignored if `replicas-assignments` is present.

Default: none (the binder-wide default of 1 is used).

useTopicHeader

Set to `true` to override the default binding destination (topic name) with the value of the `KafkaHeaders.TOPIC` message header in the outbound message. If the header is not present, the default binding destination is used. Default: `false`.

recordMetadataChannel

The bean name of a `MessageChannel` to which successful send results should be sent; the bean must exist in the application context. The message sent to the channel is the sent message (after conversion, if any) with an additional header `KafkaHeaders.RECORD_METADATA`. The header contains a `RecordMetadata` object provided by the Kafka client; it includes the partition and offset where the record was written in the topic.

```
ResultMetadata meta = sendResultMsg.getHeaders().get(KafkaHeaders.RECORD_METADATA, RecordMetadata.class)
```

Failed sends go the producer error channel (if configured); see [Error Channels](#). Default: null

+



The Kafka binder uses the `partitionCount` setting of the producer as a hint to create a topic with the given partition count (in conjunction with the `minPartitionCount`, the maximum of the two being the value being used). Exercise caution when configuring both `minPartitionCount` for a binder and `partitionCount` for an application, as the larger value is used. If a topic already exists with a smaller partition count and `autoAddPartitions` is disabled (the default), the binder fails to start. If a topic already exists with a smaller partition count and `autoAddPartitions` is enabled, new partitions are added. If a topic already exists with a larger number of partitions than the maximum of (`minPartitionCount` or `partitionCount`), the existing partition count is used.

compression

Set the `compression.type` producer property. Supported values are `none`, `gzip`, `snappy` and `lz4`. If you override the `kafka-clients` jar to 2.1.0 (or later), as discussed in the [Spring for Apache Kafka documentation](#), and wish to use `zstd` compression, use `spring.cloud.stream.kafka.bindings.<binding-name>.producer.configuration.compression.type=zstd`.

Default: `none`.

transactionManager

Bean name of a `KafkaAwareTransactionManager` used to override the binder's transaction manager for this binding. Usually needed if you want to synchronize another transaction with the Kafka transaction, using the `ChainedKafkaTransactionManager`. To achieve exactly once consumption and production of records, the consumer and producer bindings must all be configured with the same transaction manager.

Default: `none`.

Usage examples

In this section, we show the use of the preceding properties for specific scenarios.

Example: Setting `autoCommitOffset` to `false` and Relying on Manual Acking

This example illustrates how one may manually acknowledge offsets in a consumer application.

This example requires that `spring.cloud.stream.kafka.bindings.input.consumer.autoCommitOffset` be set to `false`. Use the corresponding input channel name for your example.

```

@SpringBootApplication
@EnableBinding(Sink.class)
public class ManuallyAcknowdledgingConsumer {

    public static void main(String[] args) {
        SpringApplication.run(ManuallyAcknowdledgingConsumer.class, args);
    }

    @StreamListener(Sink.INPUT)
    public void process(Message<?> message) {
        Acknowledgment acknowledgment =
message.getHeaders().get(KafkaHeaders.ACKNOWLEDGMENT, Acknowledgment.class);
        if (acknowledgment != null) {
            System.out.println("Acknowledgment provided");
            acknowledgment.acknowledge();
        }
    }
}

```

Example: Security Configuration

Apache Kafka 0.9 supports secure connections between client and brokers. To take advantage of this feature, follow the guidelines in the [Apache Kafka Documentation](#) as well as the [Kafka 0.9 security guidelines from the Confluent documentation](#). Use the `spring.cloud.stream.kafka.binder.configuration` option to set security properties for all clients created by the binder.

For example, to set `security.protocol` to `SASL_SSL`, set the following property:

```
spring.cloud.stream.kafka.binder.configuration.security.protocol=SASL_SSL
```

All the other security properties can be set in a similar manner.

When using Kerberos, follow the instructions in the [reference documentation](#) for creating and referencing the JAAS configuration.

Spring Cloud Stream supports passing JAAS configuration information to the application by using a JAAS configuration file and using Spring Boot properties.

Using JAAS Configuration Files

The JAAS and (optionally) krb5 file locations can be set for Spring Cloud Stream applications by using system properties. The following example shows how to launch a Spring Cloud Stream application with SASL and Kerberos by using a JAAS configuration file:

```
java -Djava.security.auth.login.config=/path.to/kafka_client_jaas.conf -jar log.jar \  
--spring.cloud.stream.kafka.binder.brokers=secure.server:9092 \  
--spring.cloud.stream.bindings.input.destination=stream.ticktock \  
--spring.cloud.stream.kafka.binder.configuration.security.protocol=SASL_PLAINTEXT
```

Using Spring Boot Properties

As an alternative to having a JAAS configuration file, Spring Cloud Stream provides a mechanism for setting up the JAAS configuration for Spring Cloud Stream applications by using Spring Boot properties.

The following properties can be used to configure the login context of the Kafka client:

spring.cloud.stream.kafka.binder.jaas.loginModule

The login module name. Not necessary to be set in normal cases.

Default: `com.sun.security.auth.module.Krb5LoginModule`.

spring.cloud.stream.kafka.binder.jaas.controlFlag

The control flag of the login module.

Default: `required`.

spring.cloud.stream.kafka.binder.jaas.options

Map with a key/value pair containing the login module options.

Default: Empty map.

The following example shows how to launch a Spring Cloud Stream application with SASL and Kerberos by using Spring Boot configuration properties:

```
java --spring.cloud.stream.kafka.binder.brokers=secure.server:9092 \  
--spring.cloud.stream.bindings.input.destination=stream.ticktock \  
--spring.cloud.stream.kafka.binder.autoCreateTopics=false \  
--spring.cloud.stream.kafka.binder.configuration.security.protocol=SASL_PLAINTEXT \  
--spring.cloud.stream.kafka.binder.jaas.options.useKeyTab=true \  
--spring.cloud.stream.kafka.binder.jaas.options.storeKey=true \  
  
--spring.cloud.stream.kafka.binder.jaas.options.keyTab=/etc/security/keytabs/kafka_client.keytab \  
--spring.cloud.stream.kafka.binder.jaas.options.principal=kafka-client  
-1@EXAMPLE.COM
```

The preceding example represents the equivalent of the following JAAS file:

```
KafkaClient {
    com.sun.security.auth.module.Krb5LoginModule required
    useKeyTab=true
    storeKey=true
    keyTab="/etc/security/keytabs/kafka_client.keytab"
    principal="kafka-client-1@EXAMPLE.COM";
};
```

If the topics required already exist on the broker or will be created by an administrator, autocreation can be turned off and only client JAAS properties need to be sent.



Do not mix JAAS configuration files and Spring Boot properties in the same application. If the `-Djava.security.auth.login.config` system property is already present, Spring Cloud Stream ignores the Spring Boot properties.



Be careful when using the `autoCreateTopics` and `autoAddPartitions` with Kerberos. Usually, applications may use principals that do not have administrative rights in Kafka and Zookeeper. Consequently, relying on Spring Cloud Stream to create/modify topics may fail. In secure environments, we strongly recommend creating topics and managing ACLs administratively by using Kafka tooling.

Example: Pausing and Resuming the Consumer

If you wish to suspend consumption but not cause a partition rebalance, you can pause and resume the consumer. This is facilitated by adding the `Consumer` as a parameter to your `@StreamListener`. To resume, you need an `ApplicationListener` for `ListenerContainerIdleEvent` instances. The frequency at which events are published is controlled by the `idleEventInterval` property. Since the consumer is not thread-safe, you must call these methods on the calling thread.

The following simple application shows how to pause and resume:


```

@SpringBootApplication
@EnableBinding(Sink.class)
public class Application {

    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }

    @StreamListener(Sink.INPUT)
    public void in(String in, @Header(KafkaHeaders.CONSUMER) Consumer<?, ?> consumer)
    {
        System.out.println(in);
        consumer.pause(Collections.singleton(new TopicPartition("myTopic", 0)));
    }

    @Bean
    public ApplicationListener<ListenerContainerIdleEvent> idleListener() {
        return event -> {
            System.out.println(event);
            if (event.getConsumer().paused().size() > 0) {
                event.getConsumer().resume(event.getConsumer().paused());
            }
        };
    }
}

```

25.1.4. Transactional Binder

Enable transactions by setting `spring.cloud.stream.kafka.binder.transaction.transactionIdPrefix` to a non-empty value, e.g. `tx-`. When used in a processor application, the consumer starts the transaction; any records sent on the consumer thread participate in the same transaction. When the listener exits normally, the listener container will send the offset to the transaction and commit it. A common producer factory is used for all producer bindings configured using `spring.cloud.stream.kafka.binder.transaction.producer.*` properties; individual binding Kafka producer properties are ignored.



Normal binder retries (and dead lettering) are not supported with transactions because the retries will run in the original transaction, which may be rolled back and any published records will be rolled back too. When retries are enabled (the common property `maxAttempts` is greater than zero) the retry properties are used to configure a `DefaultAfterRollbackProcessor` to enable retries at the container level. Similarly, instead of publishing dead-letter records within the transaction, this functionality is moved to the listener container, again via the `DefaultAfterRollbackProcessor` which runs after the main transaction has rolled back.

If you wish to use transactions in a source application, or from some arbitrary thread for producer-

only transaction (e.g. `@Scheduled` method), you must get a reference to the transactional producer factory and define a `KafkaTransactionManager` bean using it.

```
@Bean
public PlatformTransactionManager transactionManager(BinderFactory binders,
    @Value("${unique.tx.id.per.instance}") String txId) {

    ProducerFactory<byte[], byte[]> pf = ((KafkaMessageChannelBinder)
binders.getBinder(null,
    MessageChannel.class)).getTransactionalProducerFactory();
    KafkaTransactionManager tm = new KafkaTransactionManager<>(pf);
    tm.setTransactionId(txId)
    return tm;
}
```

Notice that we get a reference to the binder using the `BinderFactory`; use `null` in the first argument when there is only one binder configured. If more than one binder is configured, use the binder name to get the reference. Once we have a reference to the binder, we can obtain a reference to the `ProducerFactory` and create a transaction manager.

Then you would use normal Spring transaction support, e.g. `TransactionTemplate` or `@Transactional`, for example:

```
public static class Sender {

    @Transactional
    public void doInTransaction(MessageChannel output, List<String> stuffToSend) {
        stuffToSend.forEach(stuff -> output.send(new GenericMessage<>(stuff)));
    }

}
```

If you wish to synchronize producer-only transactions with those from some other transaction manager, use a `ChainedTransactionManager`.



If you deploy multiple instances of your application, each instance needs a unique `transactionIdPrefix`.

25.1.5. Error Channels

Starting with version 1.3, the binder unconditionally sends exceptions to an error channel for each consumer destination and can also be configured to send async producer send failures to an error channel. See [Error Handling](#) for more information.

The payload of the `ErrorMessage` for a send failure is a `KafkaSendFailureException` with properties:

- `failedMessage`: The Spring Messaging `Message<?>` that failed to be sent.
- `record`: The raw `ProducerRecord` that was created from the `failedMessage`

There is no automatic handling of producer exceptions (such as sending to a [Dead-Letter queue](#)). You can consume these exceptions with your own Spring Integration flow.

25.1.6. Kafka Metrics

Kafka binder module exposes the following metrics:

`spring.cloud.stream.binder.kafka.offset`: This metric indicates how many messages have not been yet consumed from a given binder's topic by a given consumer group. The metrics provided are based on the Mircometer metrics library. The metric contains the consumer group information, topic and the actual lag in committed offset from the latest offset on the topic. This metric is particularly useful for providing auto-scaling feedback to a PaaS platform.

25.1.7. Tombstone Records (null record values)

When using compacted topics, a record with a `null` value (also called a tombstone record) represents the deletion of a key. To receive such messages in a `@StreamListener` method, the parameter must be marked as not required to receive a `null` value argument.

```
@StreamListener(Sink.INPUT)
public void in(@Header(KafkaHeaders.RECEIVED_MESSAGE_KEY) byte[] key,
              @Payload(required = false) Customer customer) {
    // customer is null if a tombstone record
    ...
}
```

25.1.8. Using a KafkaRebalanceListener

Applications may wish to seek topics/partitions to arbitrary offsets when the partitions are initially assigned, or perform other operations on the consumer. Starting with version 2.1, if you provide a single `KafkaRebalanceListener` bean in the application context, it will be wired into all Kafka consumer bindings.

```

public interface KafkaBindingRebalanceListener {

    /**
     * Invoked by the container before any pending offsets are committed.
     * @param bindingName the name of the binding.
     * @param consumer the consumer.
     * @param partitions the partitions.
     */
    default void onPartitionsRevokedBeforeCommit(String bindingName, Consumer<?,
?> consumer,
        Collection<TopicPartition> partitions) {

    }

    /**
     * Invoked by the container after any pending offsets are committed.
     * @param bindingName the name of the binding.
     * @param consumer the consumer.
     * @param partitions the partitions.
     */
    default void onPartitionsRevokedAfterCommit(String bindingName, Consumer<?, ?>
consumer, Collection<TopicPartition> partitions) {

    }

    /**
     * Invoked when partitions are initially assigned or after a rebalance.
     * Applications might only want to perform seek operations on an initial
assignment.
     * @param bindingName the name of the binding.
     * @param consumer the consumer.
     * @param partitions the partitions.
     * @param initial true if this is the initial assignment.
     */
    default void onPartitionsAssigned(String bindingName, Consumer<?, ?> consumer,
Collection<TopicPartition> partitions,
        boolean initial) {

    }

}

```

You cannot set the `resetOffsets` consumer property to `true` when you provide a rebalance listener.

25.1.9. Dead-Letter Topic Processing

Dead-Letter Topic Partition Selection

By default, records are published to the Dead-Letter topic using the same partition as the original record. This means the Dead-Letter topic must have at least as many partitions as the original record.

To change this behavior, add a `DlqPartitionFunction` implementation as a `@Bean` to the application context. Only one such bean can be present. The function is provided with the consumer group, the failed `ConsumerRecord` and the exception. For example, if you always want to route to partition 0, you might use:

```
@Bean
public DlqPartitionFunction partitionFunction() {
    return (group, record, ex) -> 0;
}
```



If you set a consumer binding's `dlqPartitions` property to 1 (and the binder's `minPartitionCount` is equal to 1), there is no need to supply a `DlqPartitionFunction`; the framework will always use partition 0. If you set a consumer binding's `dlqPartitions` property to a value greater than 1 (or the binder's `minPartitionCount` is greater than 1), you **must** provide a `DlqPartitionFunction` bean, even if the partition count is the same as the original topic's.

Handling Records in a Dead-Letter Topic

Because the framework cannot anticipate how users would want to dispose of dead-lettered messages, it does not provide any standard mechanism to handle them. If the reason for the dead-lettering is transient, you may wish to route the messages back to the original topic. However, if the problem is a permanent issue, that could cause an infinite loop. The sample Spring Boot application within this topic is an example of how to route those messages back to the original topic, but it moves them to a “parking lot” topic after three attempts. The application is another spring-cloud-stream application that reads from the dead-letter topic. It terminates when no messages are received for 5 seconds.

The examples assume the original destination is `so8400out` and the consumer group is `so8400`.

There are a couple of strategies to consider:

- Consider running the rerouting only when the main application is not running. Otherwise, the retries for transient errors are used up very quickly.
- Alternatively, use a two-stage approach: Use this application to route to a third topic and another to route from there back to the main topic.

The following code listings show the sample application:

application.properties

```
spring.cloud.stream.bindings.input.group=so8400replay
spring.cloud.stream.bindings.input.destination=error.so8400out.so8400

spring.cloud.stream.bindings.output.destination=so8400out

spring.cloud.stream.bindings.parkingLot.destination=so8400in.parkingLot

spring.cloud.stream.kafka.binder.configuration.auto.offset.reset=earliest

spring.cloud.stream.kafka.binder.headers=x-retries
```

Application

```
@SpringBootApplication
@EnableBinding(TwoOutputProcessor.class)
public class ReRouteDlqApplication implements CommandLineRunner {

    private static final String X_RETRIES_HEADER = "x-retries";

    public static void main(String[] args) {
        SpringApplication.run(ReRouteDlqApplication.class, args).close();
    }

    private final AtomicInteger processed = new AtomicInteger();

    @Autowired
    private MessageChannel parkingLot;

    @StreamListener(Processor.INPUT)
    @SendTo(Processor.OUTPUT)
    public Message<?> reRoute(Message<?> failed) {
        processed.incrementAndGet();
        Integer retries = failed.getHeaders().get(X_RETRIES_HEADER, Integer.class);
        if (retries == null) {
            System.out.println("First retry for " + failed);
            return MessageBuilder.fromMessage(failed)
                .setHeader(X_RETRIES_HEADER, new Integer(1))
                .setHeader(BinderHeaders.PARTITION_OVERRIDE,

failed.getHeaders().get(KafkaHeaders.RECEIVED_PARTITION_ID))
                .build();
        }
        else if (retries.intValue() < 3) {
            System.out.println("Another retry for " + failed);
            return MessageBuilder.fromMessage(failed)
                .setHeader(X_RETRIES_HEADER, new Integer(retries.intValue() + 1))
                .setHeader(BinderHeaders.PARTITION_OVERRIDE,
```

```

failed.getHeaders().get(KafkaHeaders.RECEIVED_PARTITION_ID))
    .build();
    }
    else {
        System.out.println("Retries exhausted for " + failed);
        parkingLot.send(MessageBuilder.fromMessage(failed)
            .setHeader(BinderHeaders.PARTITION_OVERRIDE,

failed.getHeaders().get(KafkaHeaders.RECEIVED_PARTITION_ID))
    .build());
    }
    return null;
}

@Override
public void run(String... args) throws Exception {
    while (true) {
        int count = this.processed.get();
        Thread.sleep(5000);
        if (count == this.processed.get()) {
            System.out.println("Idle, terminating");
            return;
        }
    }
}

public interface TwoOutputProcessor extends Processor {

    @Output("parkingLot")
    MessageChannel parkingLot();

}
}

```

25.1.10. Partitioning with the Kafka Binder

Apache Kafka supports topic partitioning natively.

Sometimes it is advantageous to send data to specific partitions — for example, when you want to strictly order message processing (all messages for a particular customer should go to the same partition).

The following example shows how to configure the producer and consumer side:

```

@SpringBootApplication
@EnableBinding(Source.class)
public class KafkaPartitionProducerApplication {

    private static final Random RANDOM = new Random(System.currentTimeMillis());

    private static final String[] data = new String[] {
        "foo1", "bar1", "qux1",
        "foo2", "bar2", "qux2",
        "foo3", "bar3", "qux3",
        "foo4", "bar4", "qux4",
    };

    public static void main(String[] args) {
        new SpringApplicationBuilder(KafkaPartitionProducerApplication.class)
            .web(false)
            .run(args);
    }

    @InboundChannelAdapter(channel = Source.OUTPUT, poller = @Poller(fixedRate =
"5000"))
    public Message<?> generate() {
        String value = data[RANDOM.nextInt(data.length)];
        System.out.println("Sending: " + value);
        return MessageBuilder.withPayload(value)
            .setHeader("partitionKey", value)
            .build();
    }
}

```

application.yml

```

spring:
  cloud:
    stream:
      bindings:
        output:
          destination: partitioned.topic
          producer:
            partition-key-expression: headers['partitionKey']
            partition-count: 12

```



The topic must be provisioned to have enough partitions to achieve the desired concurrency for all consumer groups. The above configuration supports up to 12 consumer instances (6 if their **concurrency** is 2, 4 if their concurrency is 3, and so on). It is generally best to “over-provision” the partitions to allow for future increases in consumers or concurrency.



The preceding configuration uses the default partitioning (`key.hashCode() % partitionCount`). This may or may not provide a suitably balanced algorithm, depending on the key values. You can override this default by using the `partitionSelectorExpression` or `partitionSelectorClass` properties.

Since partitions are natively handled by Kafka, no special configuration is needed on the consumer side. Kafka allocates partitions across the instances.

The following Spring Boot application listens to a Kafka stream and prints (to the console) the partition ID to which each message goes:

```
@SpringBootApplication
@EnableBinding(Sink.class)
public class KafkaPartitionConsumerApplication {

    public static void main(String[] args) {
        new SpringApplicationBuilder(KafkaPartitionConsumerApplication.class)
            .web(false)
            .run(args);
    }

    @StreamListener(Sink.INPUT)
    public void listen(@Payload String in, @Header(KafkaHeaders.RECEIVED_PARTITION_ID)
        int partition) {
        System.out.println(in + " received from partition " + partition);
    }
}
```

application.yml

```
spring:
  cloud:
    stream:
      bindings:
        input:
          destination: partitioned.topic
          group: myGroup
```

You can add instances as needed. Kafka rebalances the partition allocations. If the instance count (or `instance count * concurrency`) exceeds the number of partitions, some consumers are idle.

25.2. Kafka Streams Binder

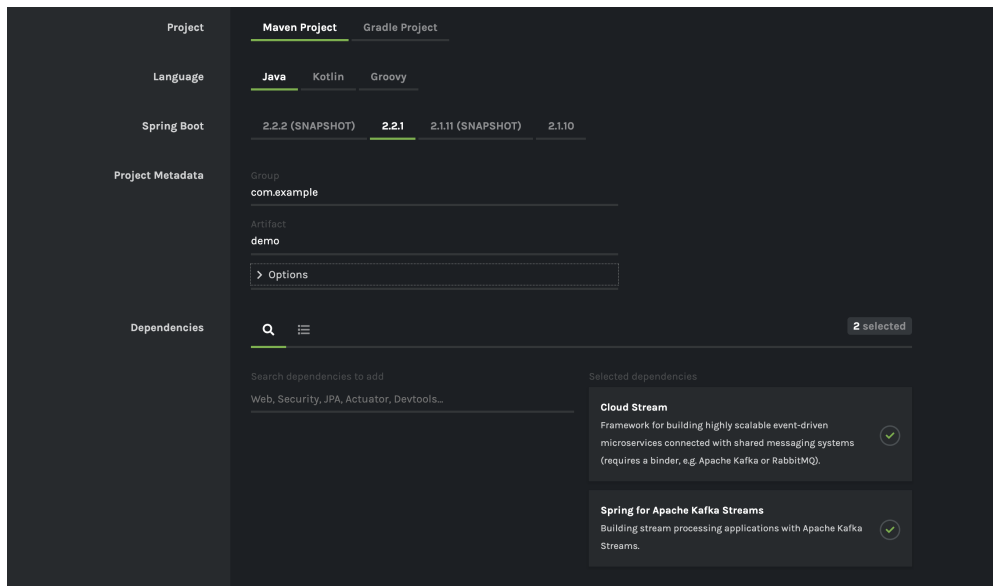
25.2.1. Usage

For using the Kafka Streams binder, you just need to add it to your Spring Cloud Stream application,

using the following maven coordinates:

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-stream-binder-kafka-streams</artifactId>
</dependency>
```

A quick way to bootstrap a new project for Kafka Streams binder is to use [Spring Initializr](#) and then select "Cloud Streams" and "Spring for Kafka Streams" as shown below



25.2.2. Overview

Spring Cloud Stream includes a binder implementation designed explicitly for [Apache Kafka Streams](#) binding. With this native integration, a Spring Cloud Stream "processor" application can directly use the [Apache Kafka Streams](#) APIs in the core business logic.

Kafka Streams binder implementation builds on the foundations provided by the [Spring for Apache Kafka](#) project.

Kafka Streams binder provides binding capabilities for the three major types in Kafka Streams - [KStream](#), [KTable](#) and [GlobalKTable](#).

Kafka Streams applications typically follow a model in which the records are read from an inbound topic, apply business logic, and then write the transformed records to an outbound topic. Alternatively, a Processor application with no outbound destination can be defined as well.

In the following sections, we are going to look at the details of Spring Cloud Stream's integration with Kafka Streams.

25.2.3. Programming Model

When using the programming model provided by Kafka Streams binder, both the high-level [Streams DSL](#) and a mix of both the higher level and the lower level [Processor-API](#) can be used as options. When mixing both higher and lower level APIs, this is usually achieved by invoking

`transform` or `process` API methods on `KStream`.

Functional Style

Starting with Spring Cloud Stream 3.0.0, Kafka Streams binder allows the applications to be designed and developed using the functional programming style that is available in Java 8. This means that the applications can be concisely represented as a lambda expression of types `java.util.function.Function` or `java.util.function.Consumer`.

Let's take a very basic example.

```
@SpringBootApplication
public class SimpleConsumerApplication {

    @Bean
    public java.util.function.Consumer<KStream<Object, String>> process() {

        return input ->
            input.foreach((key, value) -> {
                System.out.println("Key: " + key + " Value: " + value);
            });
    }
}
```

Albeit simple, this is a complete standalone Spring Boot application that is leveraging Kafka Streams for stream processing. This is a consumer application with no outbound binding and only a single inbound binding. The application consumes data and it simply logs the information from the `KStream` key and value on the standard output. The application contains the `SpringBootApplication` annotation and a method that is marked as `Bean`. The bean method is of type `java.util.function.Consumer` which is parameterized with `KStream`. Then in the implementation, we are returning a Consumer object that is essentially a lambda expression. Inside the lambda expression, the code for processing the data is provided.

In this application, there is a single input binding that is of type `KStream`. The binder creates this binding for the application with a name `process-in-0`, i.e. the name of the function bean name followed by a dash character (-) and the literal `in` followed by another dash and then the ordinal position of the parameter. You use this binding name to set other properties such as destination. For example, `spring.cloud.stream.bindings.process-in-0.destination=my-topic`.



If the destination property is not set on the binding, a topic is created with the same name as the binding (if there are sufficient privileges for the application) or that topic is expected to be already available.

Once built as a uber-jar (e.g., `kstream-consumer-app.jar`), you can run the above example like the following.

```
java -jar kstream-consumer-app.jar --spring.cloud.stream.bindings.process-in-0.destination=my-topic
```

Here is another example, where it is a full processor with both input and output bindings. This is the classic word-count example in which the application receives data from a topic, the number of occurrences for each word is then computed in a tumbling time-window.

```
@SpringBootApplication
public class WordCountProcessorApplication {

    @Bean
    public Function<KStream<Object, String>, KStream<?, WordCount>> process() {

        return input -> input
            .flatMapValues(value ->
                Arrays.asList(value.toLowerCase().split("\\W+")))
            .map((key, value) -> new KeyValue<>(value, value))
            .groupByKey(Serialized.with(Serdes.String(), Serdes.String()))
            .windowedBy(TimeWindows.of(5000))
            .count(Materialized.as("word-counts-state-store"))
            .toStream()
            .map((key, value) -> new KeyValue<>(key.key(), new
                WordCount(key.key(), value,
                    new Date(key.window().start()), new
                    Date(key.window().end()))));
    }

    public static void main(String[] args) {
        SpringApplication.run(WordCountProcessorApplication.class, args);
    }
}
```

Here again, this is a complete Spring Boot application. The difference here from the first application is that the bean method is of type `java.util.function.Function`. The first parameterized type for the `Function` is for the input `KStream` and the second one is for the output. In the method body, a lambda expression is provided that is of type `Function` and as implementation, the actual business logic is given. Similar to the previously discussed Consumer based application, the input binding here is named as `process-in-0` by default. For the output, the binding name is automatically also set to `process-out-0`.

Once built as an uber-jar (e.g., `wordcount-processor.jar`), you can run the above example like the following.

```
java -jar wordcount-processor.jar --spring.cloud.stream.bindings.process-in-0.destination=words --spring.cloud.stream.bindings.process-out-0.destination=counts
```

This application will consume messages from the Kafka topic `words` and the computed results are

published to an output topic `counts`.

Spring Cloud Stream will ensure that the messages from both the incoming and outgoing topics are automatically bound as `KStream` objects. As a developer, you can exclusively focus on the business aspects of the code, i.e. writing the logic required in the processor. Setting up Kafka Streams specific configuration required by the Kafka Streams infrastructure is automatically handled by the framework.

The two examples we saw above have a single `KStream` input binding. In both cases, the bindings received the records from a single topic. If you want to multiplex multiple topics into a single `KStream` binding, you can provide comma separated Kafka topics as destinations below.

```
spring.cloud.stream.bindings.process-in-0.destination=topic-1,topic-2,topic-3
```

In addition, you can also provide topic patterns as destinations if you want to match topics against a regular expression.

```
spring.cloud.stream.bindings.process-in-0.destination=input.*
```

Multiple Input Bindings

Many non-trivial Kafka Streams applications often consume data from more than one topic through multiple bindings. For instance, one topic is consumed as `KStream` and another as `KTable` or `GlobalKTable`. There are many reasons why an application might want to receive data as a table type. Think of a use-case where the underlying topic is populated through a change data capture (CDC) mechanism from a database or perhaps the application only cares about the latest updates for downstream processing. If the application specifies that the data needs to be bound as `KTable` or `GlobalKTable`, then Kafka Streams binder will properly bind the destination to a `KTable` or `GlobalKTable` and make them available for the application to operate upon. We will look at a few different scenarios how multiple input bindings are handled in the Kafka Streams binder.

BiFunction in Kafka Streams Binder

Here is an example where we have two inputs and an output. In this case, the application can leverage on `java.util.function.BiFunction`.

```

@Bean
public BiFunction<KStream<String, Long>, KTable<String, String>, KStream<String,
Long>> process() {
    return (userClicksStream, userRegionsTable) -> (userClicksStream
        .leftJoin(userRegionsTable, (clicks, region) -> new
RegionWithClicks(region == null ?
            "UNKNOWN" : region, clicks),
            Joined.with(Serdes.String(), Serdes.Long(), null))
        .map((user, regionWithClicks) -> new
KeyValue<>(regionWithClicks.getRegion(),
            regionWithClicks.getClicks()))
        .groupByKey(Grouped.with(Serdes.String(), Serdes.Long()))
        .reduce(Long::sum)
        .toStream());
}

```

Here again, the basic theme is the same as in the previous examples, but here we have two inputs. Java's `BiFunction` support is used to bind the inputs to the desired destinations. The default binding names generated by the binder for the inputs are `process-in-0` and `process-in-1` respectively. The default output binding is `process-out-0`. In this example, the first parameter of `BiFunction` is bound as a `KStream` for the first input and the second parameter is bound as a `KTable` for the second input.

BiConsumer in Kafka Streams Binder

If there are two inputs, but no outputs, in that case we can use `java.util.function.BiConsumer` as shown below.

```

@Bean
public BiConsumer<KStream<String, Long>, KTable<String, String>> process() {
    return (userClicksStream, userRegionsTable) -> {}
}

```

Beyond two inputs

What if you have more than two inputs? There are situations in which you need more than two inputs. In that case, the binder allows you to chain partial functions. In functional programming jargon, this technique is generally known as currying. With the functional programming support added as part of Java 8, Java now enables you to write curried functions. Spring Cloud Stream Kafka Streams binder can make use of this feature to enable multiple input bindings.

Let's see an example.

```

@Bean
public Function<KStream<Long, Order>,
    Function<GlobalKTable<Long, Customer>,
        Function<GlobalKTable<Long, Product>, KStream<Long, EnrichedOrder>>>>
enrichOrder() {

    return orders -> (
        customers -> (
            products -> (
                orders.join(customers,
                    (orderId, order) -> order.getCustomerId(),
                    (order, customer) -> new CustomerOrder(customer,
order))

                    .join(products,
                        (orderId, customerOrder) -> customerOrder
                            .productId(),
                        (customerOrder, product) -> {
                            EnrichedOrder enrichedOrder = new
EnrichedOrder();
                                enrichedOrder.setProduct(product);

enrichedOrder.setCustomer(customerOrder.customer);

enrichedOrder.setOrder(customerOrder.order);
                                return enrichedOrder;
                            })
                )
            )
        )
    );
}

```

Let's look at the details of the binding model presented above. In this model, we have 3 partially applied functions on the inbound. Let's call them as $f(x)$, $f(y)$ and $f(z)$. If we expand these functions in the sense of true mathematical functions, it will look like these: $f(x) \rightarrow (fy) \rightarrow f(z) \rightarrow KStream<Long, EnrichedOrder>$. The x variable stands for $KStream<Long, Order>$, the y variable stands for $GlobalKTable<Long, Customer>$ and the z variable stands for $GlobalKTable<Long, Product>$. The first function $f(x)$ has the first input binding of the application ($KStream<Long, Order>$) and its output is the function, $f(y)$. The function $f(y)$ has the second input binding for the application ($GlobalKTable<Long, Customer>$) and its output is yet another function, $f(z)$. The input for the function $f(z)$ is the third input for the application ($GlobalKTable<Long, Product>$) and its output is $KStream<Long, EnrichedOrder>$ which is the final output binding for the application. The input from the three partial functions which are $KStream$, $GlobalKTable$, $GlobalKTable$ respectively are available for you in the method body for implementing the business logic as part of the lambda expression.

Input bindings are named as `enrichOrder-in-0`, `enrichOrder-in-1` and `enrichOrder-in-2` respectively. Output binding is named as `enrichOrder-out-0`.

With curried functions, you can virtually have any number of inputs. However, keep in mind that, anything more than a smaller number of inputs and partially applied functions for them as above

in Java might lead to unreadable code. Therefore if your Kafka Streams application requires more than a reasonably smaller number of input bindings and you want to use this functional model, then you may want to rethink your design and decompose the application appropriately.

Multiple Output Bindings

Kafka Streams allows to write outbound data into multiple topics. This feature is known as branching in Kafka Streams. When using multiple output bindings, you need to provide an array of `KStream` (`KStream[]`) as the outbound return type.

Here is an example:

```
@Bean
public Function<KStream<Object, String>, KStream<?, WordCount>[]> process() {

    Predicate<Object, WordCount> isEnglish = (k, v) -> v.word.equals("english");
    Predicate<Object, WordCount> isFrench = (k, v) -> v.word.equals("french");
    Predicate<Object, WordCount> isSpanish = (k, v) -> v.word.equals("spanish");

    return input -> input
        .flatMapValues(value -> Arrays.asList(value.toLowerCase().split("\\W+")))
        .groupBy((key, value) -> value)
        .windowedBy(TimeWindows.of(5000))
        .count(Materialized.as("WordCounts-branch"))
        .toStream()
        .map((key, value) -> new KeyValue<>(null, new WordCount(key.key(), value,
            new Date(key.window().start()), new Date(key.window().end()))))
        .branch(isEnglish, isFrench, isSpanish);
}
```

The programming model remains the same, however the outbound parameterized type is `KStream[]`. The default output binding names are `process-out-0`, `process-out-1`, `process-out-2` respectively. The reason why the binder generates three output bindings is because it detects the length of the returned `KStream` array.

Summary of Function based Programming Styles for Kafka Streams

In summary, the following table shows the various options that can be used in the functional paradigm.

Number of Inputs	Number of Outputs	Component to use
1	0	java.util.function.Consumer
2	0	java.util.function.BiConsumer
1	1..n	java.util.function.Function
2	1..n	java.util.function.BiFunction
>= 3	0..n	Use curried functions

- In the case of more than one output in this table, the type simply becomes `KStream[]`.

Imperative programming model.

Although the functional programming model outlined above is the preferred approach, you can still use the classic `StreamListener` based approach if you prefer.

Here are some examples.

Following is the equivalent of the Word count example using `StreamListener`.

```
@SpringBootApplication
@EnableBinding(KafkaStreamsProcessor.class)
public class WordCountProcessorApplication {

    @StreamListener("input")
    @SendTo("output")
    public KStream<?, WordCount> process(KStream<?, String> input) {
        return input
            .flatMapValues(value ->
Arrays.asList(value.toLowerCase().split("\\W+")))
            .groupBy((key, value) -> value)
            .windowedBy(TimeWindows.of(5000))
            .count(Materialized.as("WordCounts-multi"))
            .toStream()
            .map((key, value) -> new KeyValue<>(null, new WordCount(key.key(),
value, new Date(key.window().start()), new Date(key.window().end()))));
    }

    public static void main(String[] args) {
        SpringApplication.run(WordCountProcessorApplication.class, args);
    }
}
```

As you can see, this is a bit more verbose since you need to provide `EnableBinding` and the other extra annotations like `StreamListener` and `SendTo` to make it a complete application. `EnableBinding` is where you specify your binding interface that contains your bindings. In this case, we are using the stock `KafkaStreamsProcessor` binding interface that has the following contracts.

```
public interface KafkaStreamsProcessor {

    @Input("input")
    KStream<?, ?> input();

    @Output("output")
    KStream<?, ?> output();

}
```

Binder will create bindings for the input `KStream` and output `KStream` since you are using a binding

interface that contains those declarations.

In addition to the obvious differences in the programming model offered in the functional style, one particular thing that needs to be mentioned here is that the binding names are what you specify in the binding interface. For example, in the above application, since we are using `KafkaStreamsProcessor`, the binding names are `input` and `output`. Binding properties need to use those names. For instance `spring.cloud.stream.bindings.input.destination`, `spring.cloud.stream.bindings.output.destination` etc. Keep in mind that this is fundamentally different from the functional style since there the binder generates binding names for the application. This is because the application does not provide any binding interfaces in the functional model using `EnableBinding`.

Here is another example of a sink where we have two inputs.

```
@EnableBinding(KStreamKTableBinding.class)
.....
.....
@StreamListener
public void process(@Input("inputStream") KStream<String, PlayEvent> playEvents,
                   @Input("inputTable") KTable<Long, Song> songTable) {
    ....
    ....
}

interface KStreamKTableBinding {

    @Input("inputStream")
    KStream<?, ?> inputStream();

    @Input("inputTable")
    KTable<?, ?> inputTable();
}
```

Following is the `StreamListener` equivalent of the same `BiFunction` based processor that we saw above.

```

@EnableBinding(KStreamKTableBinding.class)
....
....

@StreamListener
@SendTo("output")
public KStream<String, Long> process(@Input("input") KStream<String, Long>
userClicksStream,
                                @Input("inputTable") KTable<String, String>
userRegionsTable) {
....
....
}

interface KStreamKTableBinding extends KafkaStreamsProcessor {

    @Input("inputX")
    KTable<?, ?> inputTable();
}

```

Finally, here is the `StreamListener` equivalent of the application with three inputs and curried functions.

```

@EnableBinding(CustomGlobalKTableProcessor.class)
...
...
@StreamListener
@SendTo("output")
public KStream<Long, EnrichedOrder> process(
    @Input("input-1") KStream<Long, Order> ordersStream,
    @Input("input-2") GlobalKTable<Long, Customer> customers,
    @Input("input-3") GlobalKTable<Long, Product> products) {

    KStream<Long, CustomerOrder> customerOrdersStream = ordersStream.join(
        customers, (orderId, order) -> order.getCustomerId(),
        (order, customer) -> new CustomerOrder(customer, order));

    return customerOrdersStream.join(products,
        (orderId, customerOrder) -> customerOrder.productId(),
        (customerOrder, product) -> {
            EnrichedOrder enrichedOrder = new EnrichedOrder();
            enrichedOrder.setProduct(product);
            enrichedOrder.setCustomer(customerOrder.customer);
            enrichedOrder.setOrder(customerOrder.order);
            return enrichedOrder;
        });
}

interface CustomGlobalKTableProcessor {

    @Input("input-1")
    KStream<?, ?> input1();

    @Input("input-2")
    GlobalKTable<?, ?> input2();

    @Input("input-3")
    GlobalKTable<?, ?> input3();

    @Output("output")
    KStream<?, ?> output();
}

```

You might notice that the above two examples are even more verbose since in addition to provide `EnableBinding`, you also need to write your own custom binding interface as well. Using the functional model, you can avoid all those ceremonial details.

Before we move on from looking at the general programming model offered by Kafka Streams binder, here is the `StreamListener` version of multiple output bindings.

```

@EnableBinding(KStreamProcessorWithBranches.class)
public static class WordCountProcessorApplication {

    @Autowired
    private TimeWindows timeWindows;

    @StreamListener("input")
    @SendTo({"output1", "output2", "output3"})
    public KStream<?, WordCount>[] process(KStream<Object, String> input) {

        Predicate<Object, WordCount> isEnglish = (k, v) ->
v.word.equals("english");
        Predicate<Object, WordCount> isFrench = (k, v) ->
v.word.equals("french");
        Predicate<Object, WordCount> isSpanish = (k, v) ->
v.word.equals("spanish");

        return input
            .flatMapValues(value ->
Arrays.asList(value.toLowerCase().split("\\W+")))
            .groupBy((key, value) -> value)
            .windowedBy(timeWindows)
            .count(Materialized.as("WordCounts-1"))
            .toStream()
            .map((key, value) -> new KeyValue<>(null, new WordCount(key.key(),
value, new Date(key.window().start()), new Date(key.window().end()))))
            .branch(isEnglish, isFrench, isSpanish);
    }

    interface KStreamProcessorWithBranches {

        @Input("input")
        KStream<?, ?> input();

        @Output("output1")
        KStream<?, ?> output1();

        @Output("output2")
        KStream<?, ?> output2();

        @Output("output3")
        KStream<?, ?> output3();
    }
}

```

To recap, we have reviewed the various programming model choices when using the Kafka Streams binder.

The binder provides binding capabilities for `KStream`, `KTable` and `GlobalKTable` on the input. `KTable` and `GlobalKTable` bindings are only available on the input. Binder supports both input and output

bindings for `KStream`.

The upshot of the programming model of Kafka Streams binder is that the binder provides you the flexibility of going with a fully functional programming model or using the `StreamListener` based imperative approach.

25.2.4. Ancillaries to the programming model

Multiple Kafka Streams processors within a single application

Binder allows to have multiple Kafka Streams processors within a single Spring Cloud Stream application. You can have an application as below.

```
@Bean
public java.util.function.Function<KStream<Object, String>, KStream<Object, String>>
process() {
    ...
}

@Bean
public java.util.function.Consumer<KStream<Object, String>> anotherProcess() {
    ...
}

@Bean
public java.util.function.BiFunction<KStream<Object, String>, KTable<Integer, String>,
KStream<Object, String>> yetAnotherProcess() {
    ...
}
```

In this case, the binder will create 3 separate Kafka Streams objects with different application ID's (more on this below). However, if you have more than one processor in the application, you have to tell Spring Cloud Stream, which functions need to be activated. Here is how you activate the functions.

```
spring.cloud.stream.function.definition: process;anotherProcess;yetAnotherProcess
```

If you want certain functions to be not activated right away, you can remove that from this list.

This is also true when you have a single Kafka Streams processor and other types of `Function` beans in the same application that is handled through a different binder (for e.g., a function bean that is based on the regular Kafka Message Channel binder)

Kafka Streams Application ID

Application id is a mandatory property that you need to provide for a Kafka Streams application. Spring Cloud Stream Kafka Streams binder allows you to configure this application id in multiple ways.

If you only have one single processor or `StreamListener` in the application, then you can set this at

the binder level using the following property:

```
spring.cloud.stream.kafka.streams.binder.applicationId.
```

As a convenience, if you only have a single processor, you can also use `spring.application.name` as the property to delegate the application id.

If you have multiple Kafka Streams processors in the application, then you need to set the application id per processor. In the case of the functional model, you can attach it to each function as a property.

For e.g. imagine that you have the following functions.

```
@Bean
public java.util.function.Consumer<KStream<Object, String>> process() {
    ...
}
```

and

```
@Bean
public java.util.function.Consumer<KStream<Object, String>> anotherProcess() {
    ...
}
```

Then you can set the application id for each, using the following binder level properties.

```
spring.cloud.stream.kafka.streams.binder.functions.process.applicationId
```

and

```
spring.cloud.stream.kafka.streams.binder.functions.anotherProcess.applicationId
```

In the case of `StreamListener`, you need to set this on the first input binding on the processor.

For e.g. imagine that you have the following two `StreamListener` based processors.

```

@StreamListener
@SendTo("output")
public KStream<String, String> process(@Input("input") <KStream<Object, String>>
input) {
    ...
}

@StreamListener
@SendTo("anotherOutput")
public KStream<String, String> anotherProcess(@Input("anotherInput") <KStream<Object,
String>> input) {
    ...
}

```

Then you must set the application id for this using the following binding property.

```
spring.cloud.stream.kafka.streams.bindings.input.consumer.applicationId
```

and

```
spring.cloud.stream.kafka.streams.bindings.anotherInput.consumer.applicationId
```

For function based model also, this approach of setting application id at the binding level will work. However, setting per function at the binder level as we have seen above is much easier if you are using the functional model.

For production deployments, it is highly recommended to explicitly specify the application ID through configuration. This is especially going to be very critical if you are auto scaling your application in which case you need to make sure that you are deploying each instance with the same application ID.

If the application does not provide an application ID, then in that case the binder will auto generate a static application ID for you. This is convenient in development scenarios as it avoids the need for explicitly providing the application ID. The generated application ID in this manner will be static over application restarts. In the case of functional model, the generated application ID will be the function bean name followed by the literal `applicationID`, for e.g `process-applicationID` if `process` is the function bean name. In the case of `StreamListener`, instead of using the function bean name, the generated application ID will be use the containing class name followed by the method name followed by the literal `applicationId`.

Summary of setting Application ID

- By default, binder will auto generate the application ID per function or `StreamListener` methods.
- If you have a single processor, then you can use `spring.kafka.streams.applicationId`, `spring.application.name` or `spring.cloud.stream.kafka.streams.binder.applicationId`.
- If you have multiple processors, then application ID can be set per function using the property - `spring.cloud.stream.kafka.streams.binder.functions.<function-name>.applicationId`. In the case of `StreamListener`, this can be done using `spring.cloud.stream.kafka.streams.bindings.input.applicationId`, assuming that the input

binding name is `input`.

Overriding the default binding names generated by the binder with the functional style

By default, the binder uses the strategy discussed above to generate the binding name when using the functional style, i.e. `<function-bean-name>-<in>|<out>-[0..n]`, for e.g. `process-in-0`, `process-out-0` etc. If you want to override those binding names, you can do that by specifying the following properties.

`spring.cloud.stream.function.bindings.<default binding name>`. Default binding name is the original binding name generated by the binder.

For e.g. lets say, you have this function.

```
@Bean
public BiFunction<KStream<String, Long>, KTable<String, String>, KStream<String, Long>> process() {
    ...
}
```

Binder will generate bindings with names, `process-in-0`, `process-in-1` and `process-out-0`. Now, if you want to change them to something else completely, maybe more domain specific binding names, then you can do so as below.

`spring.cloud.stream.function.bindings.process-in-0=users`

`spring.cloud.stream.function.bindings.process-in-0=regions`

and

`spring.cloud.stream.function.bindings.process-out-0=clicks`

After that, you must set all the binding level properties on these new binding names.

Please keep in mind that with the functional programming model described above, adhering to the default binding names make sense in most situations. The only reason you may still want to do this overriding is when you have larger number of configuration properties and you want to map the bindings to something more domain friendly.

Setting up bootstrap server configuration

When running Kafka Streams applications, you must provide the Kafka broker server information. If you don't provide this information, the binder expects that you are running the broker at the default `localhost:9092`. If that is not the case, then you need to override that. There are a couple of ways to do that.

- Using the boot property - `spring.kafka.bootstrapServers`
- Binder level property - `spring.cloud.stream.kafka.streams.binder.brokers`

When it comes to the binder level property, it doesn't matter if you use the broker property

provided through the regular Kafka binder - `spring.cloud.stream.kafka.binder.brokers`. Kafka Streams binder will first check if Kafka Streams binder specific broker property is set (`spring.cloud.stream.kafka.streams.binder.brokers`) and if not found, it looks for `spring.cloud.stream.kafka.binder.brokers`.

25.2.5. Record serialization and deserialization

Kafka Streams binder allows you to serialize and deserialize records in two ways. One is the native serialization and deserialization facilities provided by Kafka and the other one is the message conversion capabilities of Spring Cloud Stream framework. Lets look at some details.

Inbound deserialization

Keys are always deserialized using native Serdes.

For values, by default, deserialization on the inbound is natively performed by Kafka. Please note that this is a major change on default behavior from previous versions of Kafka Streams binder where the deserialization was done by the framework.

Kafka Streams binder will try to infer matching `Serde` types by looking at the type signature of `java.util.function.Function|Consumer` or `StreamListener`. Here is the order that it matches the Serdes.

- If the application provides a bean of type `Serde` and if the return type is parameterized with the actual type of the incoming key or value type, then it will use that `Serde` for inbound deserialization. For e.g. if you have the following in the application, the binder detects that the incoming value type for the `KStream` matches with a type that is parameterized on a `Serde` bean. It will use that for inbound deserialization.

```
@Bean
public Serde<Foo() customSerde{
    ...
}

@Bean
public Function<KStream<String, Foo>, KStream<String, Foo>> process() {
}
```

- Next, it looks at the types and see if they are one of the types exposed by Kafka Streams. If so, use them. Here are the `Serde` types that the binder will try to match from Kafka Streams.

Integer, Long, Short, Double, Float, byte[], UUID and String.

- If none of the Serdes provided by Kafka Streams don't match the types, then it will use `JsonSerde` provided by Spring Kafka. In this case, the binder assumes that the types are JSON friendly. This is useful if you have multiple value objects as inputs since the binder will internally infer them to correct Java types. Before falling back to the `JsonSerde` though, the binder checks at the default `Serde`'s set in the Kafka Streams configuration to see if it is a

`Serde` that it can match with the incoming KStream's types.

If none of the above strategies worked, then the applications must provide the `Serde`s through configuration. This can be configured in two ways - binding or default.

First the binder will look if a **Serde** is provided at the binding level. For e.g. if you have the following processor,

```
@Bean
public BiFunction<KStream<CustomKey, AvroIn1>, KTable<CustomKey, AvroIn2>,
KStream<CustomKey, AvroOutput>> process() {...}
```

then, you can provide a binding level **Serde** using the following:

```
spring.cloud.stream.kafka.streams.bindings.process-in-
0.consumer.keySerde=CustomKeySerde
spring.cloud.stream.kafka.streams.bindings.process-in-
0.consumer.valueSerde=io.confluent.kafka.streams.serdes.avro.SpecificAvroSerde

spring.cloud.stream.kafka.streams.bindings.process-in-
1.consumer.keySerde=CustomKeySerde
spring.cloud.stream.kafka.streams.bindings.process-in-
1.consumer.valueSerde=io.confluent.kafka.streams.serdes.avro.SpecificAvroSerde
```



If you provide **Serde** as above per input binding, then that will take higher precedence and the binder will stay away from any **Serde** inference.

If you want the default key/value Serdes to be used for inbound deserialization, you can do so at the binder level.

```
spring.cloud.stream.kafka.streams.binder.configuration.default.key.serde
spring.cloud.stream.kafka.streams.binder.configuration.default.value.serde
```

If you don't want the native decoding provided by Kafka, you can rely on the message conversion features that Spring Cloud Stream provides. Since native decoding is the default, in order to let Spring Cloud Stream deserialize the inbound value object, you need to explicitly disable native decoding.

For e.g. if you have the same BiFunction processor as above, then `spring.cloud.stream.bindings.process-in-0.consumer.nativeDecoding: false` You need to disable native decoding for all the inputs individually. Otherwise, native decoding will still be applied for those you do not disable.

By default, Spring Cloud Stream will use `application/json` as the content type and use an appropriate json message converter. You can use custom message converters by using the following property and an appropriate **MessageConverter** bean.

```
spring.cloud.stream.bindings.process-in-0.contentType
```

Outbound serialization

Outbound serialization pretty much follows the same rules as above for inbound deserialization. As with the inbound deserialization, one major change from the previous versions of Spring Cloud Stream is that the serialization on the outbound is handled by Kafka natively. Before 3.0 versions of the binder, this was done by the framework itself.

Keys on the outbound are always serialized by Kafka using a matching **Serde** that is inferred by the binder. If it can't infer the type of the key, then that needs to be specified using configuration.

Value serdes are inferred using the same rules used for inbound deserialization. First it matches to see if the outbound type is from a provided bean in the application. If not, it checks to see if it matches with a **Serde** exposed by Kafka such as - **Integer**, **Long**, **Short**, **Double**, **Float**, **byte[]**, **UUID** and **String**. If that doesn't work, then it falls back to **JsonSerde** provided by the Spring Kafka project, but first look at the default **Serde** configuration to see if there is a match. Keep in mind that all these happen transparently to the application. If none of these work, then the user has to provide the **Serde** to use by configuration.

Lets say you are using the same **BiFunction** processor as above. Then you can configure outbound key/value Serdes as following.

```
spring.cloud.stream.kafka.streams.bindings.process-out-  
0.producer.keySerde=CustomKeySerde  
spring.cloud.stream.kafka.streams.bindings.process-out-  
0.producer.valueSerde=io.confluent.kafka.streams.serdes.avro.SpecificAvroSerde
```

If Serde inference fails, and no binding level Serdes are provided, then the binder falls back to the **JsonSerde**, but look at the default Serdes for a match.

Default serdes are configured in the same way as above where it is described under deserialization.

```
spring.cloud.stream.kafka.streams.binder.configuration.default.key.serde  
spring.cloud.stream.kafka.streams.binder.configuration.default.value.serde
```

If your application uses the branching feature and has multiple output bindings, then these have to be configured per binding. Once again, if the binder is capable of inferring the **Serde** types, you don't need to do this configuration.

If you don't want the native encoding provided by Kafka, but want to use the framework provided message conversion, then you need to explicitly disable native encoding since since native encoding is the default. For e.g. if you have the same **BiFunction** processor as above, then **spring.cloud.stream.bindings.process-out-0.producer.nativeEncoding: false** You need to disable native encoding for all the output individually in the case of branching. Otherwise, native encoding will still be applied for those you don't disable.

When conversion is done by Spring Cloud Stream, by default, it will use **application/json** as the

content type and use an appropriate json message converter. You can use custom message converters by using the following property and a corresponding `MessageConverter` bean.

```
spring.cloud.stream.bindings.process-out-0.contentType
```

When native encoding/decoding is disabled, binder will not do any inference as in the case of native Serdes. Applications need to explicitly provide all the configuration options. For that reason, it is generally advised to stay with the default options for de/serialization and stick with native de/serialization provided by Kafka Streams when you write Spring Cloud Stream Kafka Streams applications. The one scenario in which you must use message conversion capabilities provided by the framework is when your upstream producer is using a specific serialization strategy. In that case, you want to use a matching deserialization strategy as native mechanisms may fail. When relying on the default `Serde` mechanism, the applications must ensure that the binder has a way forward with correctly map the inbound and outbound with a proper `Serde`, as otherwise things might fail.

It is worth to mention that the data de/serialization approaches outlined above are only applicable on the edges of your processors, i.e. - inbound and outbound. Your business logic might still need to call Kafka Streams API's that explicitly need `Serde` objects. Those are still the responsibility of the application and must be handled accordingly by the developer.

25.2.6. Error Handling

Apache Kafka Streams provides the capability for natively handling exceptions from deserialization errors. For details on this support, please see [this](#). Out of the box, Apache Kafka Streams provides two kinds of deserialization exception handlers - `LogAndContinueExceptionHandler` and `LogAndFailExceptionHandler`. As the name indicates, the former will log the error and continue processing the next records and the latter will log the error and fail. `LogAndFailExceptionHandler` is the default deserialization exception handler.

Handling Deserialization Exceptions in the Binder

Kafka Streams binder allows to specify the deserialization exception handlers above using the following property.

```
spring.cloud.stream.kafka.streams.binder.deserializationExceptionHandler:  
logAndContinue
```

or

```
spring.cloud.stream.kafka.streams.binder.deserializationExceptionHandler: logAndFail
```

In addition to the above two deserialization exception handlers, the binder also provides a third one for sending the erroneous records (poison pills) to a DLQ (dead letter queue) topic. Here is how you enable this DLQ exception handler.

```
spring.cloud.stream.kafka.streams.binder.deserializationExceptionHandler: sendToDlq
```

When the above property is set, all the records in deserialization error are automatically sent to the DLQ topic.

You can set the topic name where the DLQ messages are published as below.

```
spring.cloud.stream.kafka.streams.bindings.process-in-0.consumer.dlqName: custom-dlq  
(Change the binding name accordingly)
```

If this is set, then the error records are sent to the topic `custom-dlq`. If this is not set, then it will create a DLQ topic with the name `error.<input-topic-name>.<application-id>`. For instance, if your binding's destination topic is `inputTopic` and the application ID is `process-applicationId`, then the default DLQ topic is `error.inputTopic.process-applicationId`. It is always recommended to explicitly create a DLQ topic for each input binding if it is your intention to enable DLQ.

DLQ per input consumer binding

The property `spring.cloud.stream.kafka.streams.binder.deserializationExceptionHandler` is applicable for the entire application. This implies that if there are multiple functions or `StreamListener` methods in the same application, this property is applied to all of them. However, if you have multiple processors or multiple input bindings within a single processor, then you can use the finer-grained DLQ control that the binder provides per input consumer binding.

If you have the following processor,

```
@Bean  
public BiFunction<KStream<String, Long>, KTable<String, String>, KStream<String,  
Long>> process() {  
    ...  
}
```

and you only want to enable DLQ on the first input binding and `logAndSkip` on the second binding, then you can do so on the consumer as below.

```
spring.cloud.stream.kafka.streams.bindings.process-in-  
0.consumer.deserializationExceptionHandler: sendToDlq  
spring.cloud.stream.kafka.streams.bindings.process-in-  
1.consumer.deserializationExceptionHandler: logAndSkip
```

Setting deserialization exception handlers this way has a higher precedence than setting at the binder level.

DLQ partitioning

By default, records are published to the Dead-Letter topic using the same partition as the original record. This means the Dead-Letter topic must have at least as many partitions as the original

record.

To change this behavior, add a `DlqPartitionFunction` implementation as a `@Bean` to the application context. Only one such bean can be present. The function is provided with the consumer group (which is the same as the application ID in most situations), the failed `ConsumerRecord` and the exception. For example, if you always want to route to partition 0, you might use:

```
@Bean
public DlqPartitionFunction partitionFunction() {
    return (group, record, ex) -> 0;
}
```



If you set a consumer binding's `dlqPartitions` property to 1 (and the binder's `minPartitionCount` is equal to 1), there is no need to supply a `DlqPartitionFunction`; the framework will always use partition 0. If you set a consumer binding's `dlqPartitions` property to a value greater than 1 (or the binder's `minPartitionCount` is greater than 1), you **must** provide a `DlqPartitionFunction` bean, even if the partition count is the same as the original topic's.

A couple of things to keep in mind when using the exception handling feature in Kafka Streams binder.

- The property `spring.cloud.stream.kafka.streams.binder.deserializationExceptionHandler` is applicable for the entire application. This implies that if there are multiple functions or `StreamListener` methods in the same application, this property is applied to all of them.
- The exception handling for deserialization works consistently with native deserialization and framework provided message conversion.

Handling Production Exceptions in the Binder

Unlike the support for deserialization exception handlers as described above, the binder does not provide such first class mechanisms for handling production exceptions. However, you still can configure production exception handlers using the `StreamsBuilderFactoryBean` customizer which you can find more details about, in a subsequent section below.

25.2.7. State Store

State stores are created automatically by Kafka Streams when the high level DSL is used and appropriate calls are made those trigger a state store.

If you want to materialize an incoming `KTable` binding as a named state store, then you can do so by using the following strategy.

Lets say you have the following function.

```
@Bean
public BiFunction<KStream<String, Long>, KTable<String, String>, KStream<String,
Long>> process() {
    ...
}
```

Then by setting the following property, the incoming `KTable` data will be materialized in to the named state store.

```
spring.cloud.stream.kafka.streams.bindings.process-in-1.consumer.materializedAs:
incoming-store
```

You can define custom state stores as beans in your application and those will be detected and added to the Kafka Streams builder by the binder. Especially when the processor API is used, you need to register a state store manually. In order to do so, you can create the `StateStore` as a bean in the application. Here are examples of defining such beans.

```
@Bean
public StoreBuilder myStore() {
    return Stores.keyValueStoreBuilder(
        Stores.persistentKeyValueStore("my-store"), Serdes.Long(),
        Serdes.Long());
}

@Bean
public StoreBuilder otherStore() {
    return Stores.windowStoreBuilder(
        Stores.persistentWindowStore("other-store",
            1L, 3, 3L, false), Serdes.Long(),
        Serdes.Long());
}
```

These state stores can be then accessed by the applications directly.

During the bootstrap, the above beans will be processed by the binder and passed on to the Streams builder object.

Accessing the state store:


```

Processor<Object, Product>() {

    WindowStore<Object, String> state;

    @Override
    public void init(ProcessorContext processorContext) {
        state = (WindowStore)processorContext.getStateStore("mystate");
    }
    ...
}

```

This will not work when it comes to registering global state stores. In order to register a global state store, please see the section below on customizing [StreamsBuilderFactoryBean](#).

25.2.8. Interactive Queries

Kafka Streams binder API exposes a class called [InteractiveQueryService](#) to interactively query the state stores. You can access this as a Spring bean in your application. An easy way to get access to this bean from your application is to [autowire](#) the bean.

```

@Autowired
private InteractiveQueryService interactiveQueryService;

```

Once you gain access to this bean, then you can query for the particular state-store that you are interested. See below.

```

ReadOnlyKeyValueStore<Object, Object> keyValueStore =
    interactiveQueryService.getQueryableStoreType("my-store",
    QueryableStoreTypes.keyValueStore());

```

During the startup, the above method call to retrieve the store might fail. For e.g it might still be in the middle of initializing the state store. In such cases, it will be useful to retry this operation. Kafka Streams binder provides a simple retry mechanism to accommodate this.

Following are the two properties that you can use to control this retrying.

- `spring.cloud.stream.kafka.streams.binder.stateStoreRetry.maxAttempts` - Default is `1` .
- `spring.cloud.stream.kafka.streams.binder.stateStoreRetry.backOffInterval` - Default is `1000` milliseconds.

If there are multiple instances of the kafka streams application running, then before you can query them interactively, you need to identify which application instance hosts the particular key that you are querying. [InteractiveQueryService](#) API provides methods for identifying the host information.

In order for this to work, you must configure the property `application.server` as below:

```
spring.cloud.stream.kafka.streams.binder.configuration.application.server:  
<server>:<port>
```

Here are some code snippets:

```
org.apache.kafka.streams.state.HostInfo hostInfo =  
interactiveQueryService.getHostInfo("store-name",  
    key, keySerializer);  
  
if (interactiveQueryService.getCurrentHostInfo().equals(hostInfo)) {  
    //query from the store that is locally available  
}  
else {  
    //query from the remote host  
}
```

25.2.9. Health Indicator

The health indicator requires the dependency `spring-boot-starter-actuator`. For maven use:

```
<dependency>  
  <groupId>org.springframework.boot</groupId>  
  <artifactId>spring-boot-starter-actuator</artifactId>  
</dependency>
```

Spring Cloud Stream Kafka Streams Binder provides a health indicator to check the state of the underlying streams threads. Spring Cloud Stream defines a property `management.health.binders.enabled` to enable the health indicator. See the [Spring Cloud Stream documentation](#).

The health indicator provides the following details for each stream thread's metadata:

- Thread name
- Thread state: `CREATED`, `RUNNING`, `PARTITIONS_REVOKED`, `PARTITIONS_ASSIGNED`, `PENDING_SHUTDOWN` or `DEAD`
- Active tasks: task ID and partitions
- Standby tasks: task ID and partitions

By default, only the global status is visible (`UP` or `DOWN`). To show the details, the property `management.endpoint.health.show-details` must be set to `ALWAYS` or `WHEN_AUTHORIZED`. For more details about the health information, see the [Spring Boot Actuator documentation](#).



The status of the health indicator is `UP` if all the Kafka threads registered are in the `RUNNING` state.

Since there are three individual binders in Kafka Streams binder (`KStream`, `KTable` and `GlobalKTable`), all of them will report the health status. When enabling `show-details`, some of the information reported may be redundant.

When there are multiple Kafka Streams processors present in the same application, then the health checks will be reported for all of them and will be categorized by the application ID of Kafka Streams.

25.2.10. Accessing Kafka Streams Metrics

Spring Cloud Stream Kafka Streams binder provides a basic mechanism for accessing Kafka Streams metrics exported through a Micrometer `MeterRegistry`. Kafka Streams metrics that are available through `KafkaStreams#metrics()` are exported to this meter registry by the binder. The metrics exported are from the consumers, producers, admin-client and the stream itself.

The metrics exported by the binder are exported with the format of metrics group name followed by a dot and then the actual metric name. All dashes in the original metric information is replaced with dots.

For e.g. the metric name `network-io-total` from the metric group `consumer-metrics` is available in the micrometer registry as `consumer.metrics.network.io.total`. Similarly, the metric `commit-total` from `stream-metrics` is available as `stream.metrics.commit.total`.

If you have multiple Kafka Streams processors in the same application, then the metric name will be prepended with the corresponding application ID of the Kafka Streams. The application ID in this case will be preserved as is, i.e. no dashes will be converted to dots etc. For example, if the application ID of the first processor is `processor-1`, then the metric name `network-io-total` from the metric group `consumer-metrics` is available in the micrometer registry as `processor-1.consumer.metrics.network.io.total`.

You can either programmatically access the Micrometer `MeterRegistry` in the application and then iterate through the available gauges or use Spring Boot actuator to access the metrics through a REST endpoint. When accessing through the Boot actuator endpoint, make sure to add `metrics` to the property `management.endpoints.web.exposure.include`. Then you can access `/actuator/metrics` to get a list of all the available metrics which then can be individually accessed through the same URI (`/actuator/metrics/<metric-name>`).

Anything beyond the info level metrics available through `KafkaStreams#metrics()`, (for e.g. the debugging level metrics) are still only available through JMX after you set the `metrics.recording.level` to `DEBUG`. Kafka Streams, by default, set this level to `INFO`. [Please see this section](#) from Kafka Streams documentation for more details. In a future release, binder may support exporting these DEBUG level metrics through Micrometer.

25.2.11. Mixing high level DSL and low level Processor API

Kafka Streams provides two variants of APIs. It has a higher level DSL like API where you can chain various operations that maybe familiar to a lot of functional programmers. Kafka Streams also gives access to a low level Processor API. The processor API, although very powerful and gives the ability to control things in a much lower level, is imperative in nature. Kafka Streams binder for

Spring Cloud Stream, allows you to use either the high level DSL or mixing both the DSL and the processor API. Mixing both of these variants give you a lot of options to control various use cases in an application. Applications can use the `transform` or `process` method API calls to get access to the processor API.

Here is a look at how one may combine both the DSL and the processor API in a Spring Cloud Stream application using the `process` API.

```
@Bean
public Consumer<KStream<Object, String>> process() {
    return input ->
        input.process(() -> new Processor<Object, String>() {
            @Override
            @SuppressWarnings("unchecked")
            public void init(ProcessorContext context) {
                this.context = context;
            }

            @Override
            public void process(Object key, String value) {
                //business logic
            }

            @Override
            public void close() {

            }
        });
}
```

Here is an example using the `transform` API.

```

@Bean
public Consumer<KStream<Object, String>> process() {
    return (input, a) ->
        input.transform(() -> new Transformer<Object, String, KeyValue<Object,
String>>() {
            @Override
            public void init(ProcessorContext context) {

            }

            @Override
            public void close() {

            }

            @Override
            public KeyValue<Object, String> transform(Object key, String value) {
                // business logic - return transformed KStream;
            }
        });
}

```

The `process` API method call is a terminal operation while the `transform` API is non terminal and gives you a potentially transformed `KStream` using which you can continue further processing using either the DSL or the processor API.

25.2.12. Partition support on the outbound

A Kafka Streams processor usually sends the processed output into an outbound Kafka topic. If the outbound topic is partitioned and the processor needs to send the outgoing data into particular partitions, the applications needs to provide a bean of type `StreamPartitioner`. See [StreamPartitioner](#) for more details. Let's see some examples.

This is the same processor we already saw multiple times,

```

@Bean
public Function<KStream<Object, String>, KStream<?, WordCount>> process() {

    ...

}

```

Here is the output binding destination:

```
spring.cloud.stream.bindings.process-out-0.destination: outputTopic
```

If the topic `outputTopic` has 4 partitions, if you don't provide a partitioning strategy, Kafka Streams will use default partitioning strategy which may not be the outcome you want depending on the

particular use case. Let's say, you want to send any key that matches to `spring` to partition 0, `cloud` to partition 1, `stream` to partition 2, and everything else to partition 3. This is what you need to do in the application.

```
@Bean
public StreamPartitioner<String, WordCount> streamPartitioner() {
    return (t, k, v, n) -> {
        if (k.equals("spring")) {
            return 0;
        }
        else if (k.equals("cloud")) {
            return 1;
        }
        else if (k.equals("stream")) {
            return 2;
        }
        else {
            return 3;
        }
    };
}
```

This is a rudimentary implementation, however, you have access to the key/value of the record, the topic name and the total number of partitions. Therefore, you can implement complex partitioning strategies if need be.

You also need to provide this bean name along with the application configuration.

```
spring.cloud.stream.kafka.streams.bindings.process-out-
0.producer.streamPartitionerBeanName: streamPartitioner
```

Each output topic in the application needs to be configured separately like this.

25.2.13. `StreamsBuilderFactoryBean` customizer

It is often required to customize the `StreamsBuilderFactoryBean` that creates the `KafkaStreams` objects. Based on the underlying support provided by Spring Kafka, the binder allows you to customize the `StreamsBuilderFactoryBean`. You can use the `StreamsBuilderFactoryBeanCustomizer` to customize the `StreamsBuilderFactoryBean` itself. Then, once you get access to the `StreamsBuilderFactoryBean` through this customizer, you can customize the corresponding `KafkaStreams` using `KafkaStreamsCustomzier`. Both of these customizers are part of the Spring for Apache Kafka project.

Here is an example of using the `StreamsBuilderFactoryBeanCustomizer`.

```

@Bean
public StreamsBuilderFactoryBeanCustomizer streamsBuilderFactoryBeanCustomizer() {
    return sfb -> sfb.setStateListener((newState, oldState) -> {
        //Do some action here!
    });
}

```

The above is shown as an illustration of the things you can do to customize the `StreamsBuilderFactoryBean`. You can essentially call any available mutation operations from `StreamsBuilderFactoryBean` to customize it. This customizer will be invoked by the binder right before the factory bean is started.

Once you get access to the `StreamsBuilderFactoryBean`, you can also customize the underlying `KafkaStreams` object. Here is a blueprint for doing so.

```

@Bean
public StreamsBuilderFactoryBeanCustomizer streamsBuilderFactoryBeanCustomizer() {
    return factoryBean -> {
        factoryBean.setKafkaStreamsCustomizer(new KafkaStreamsCustomizer() {
            @Override
            public void customize(KafkaStreams kafkaStreams) {
                kafkaStreams.setUncaughtExceptionHandler((t, e) -> {

                });
            }
        });
    };
}

```

`KafkaStreamsCustomizer` will be called by the `StreamsBuilderFactoryBean` right before the underlying `KafkaStreams` gets started.

There can only be one `StreamsBuilderFactoryBeanCustomizer` in the entire application. Then how do we account for multiple Kafka Streams processors as each of them are backed up by individual `StreamsBuilderFactoryBean` objects? In that case, if the customization needs to be different for those processors, then the application needs to apply some filter based on the application ID.

For e.g,

```

@Bean
public StreamsBuilderFactoryBeanCustomizer streamsBuilderFactoryBeanCustomizer() {

    return factoryBean -> {
        if
(factoryBean.getStreamsConfiguration().getProperty(StreamsConfig.APPLICATION_ID_CONFIG
)
        .equals("processor1-application-id")) {
            factoryBean.setKafkaStreamsCustomizer(new KafkaStreamsCustomizer() {
                @Override
                public void customize(KafkaStreams kafkaStreams) {
                    kafkaStreams.setUncaughtExceptionHandler((t, e) -> {

                        });
                }
            });
        }
    };
};

```

Using Customizer to register a global state store

As mentioned above, the binder does not provide a first class way to register global state stores as a feature. For that, you need to use the customizer. Here is how that can be done.

```

@Bean
public StreamsBuilderFactoryBeanCustomizer customizer() {
    return fb -> {
        try {
            final StreamsBuilder streamsBuilder = fb.getObject();
            streamsBuilder.addGlobalStore(...);
        }
        catch (Exception e) {

        }
    };
};
}

```

Again, if you have multiple processors, you want to attach the global state store to the right `StreamsBuilder` by filtering out the other `StreamsBuilderFactoryBean` objects using the application id as outlined above.

Using customizer to register a production exception handler

In the error handling section, we indicated that the binder does not provide a first class way to deal with production exceptions. Though that is the case, you can still use the `StreamsBuilderFactoryBean` customizer to register production exception handlers. See below.


```

@Bean
public StreamsBuilderFactoryBeanCustomizer customizer() {
    return fb -> {

        fb.getStreamsConfiguration().put(StreamsConfig.DEFAULT_PRODUCTION_EXCEPTION_HANDLER_CLASS_CONFIG,
                                         CustomProductionExceptionHandler.class);

    };
}

```

Once again, if you have multiple processors, you may want to set it appropriately against the correct `StreamsBuilderFactoryBean`. You may also add such production exception handlers using the configuration property (See below for more on that), but this is an option if you choose to go with a programmatic approach.

25.2.14. Timestamp extractor

Kafka Streams allows you to control the processing of the consumer records based on various notions of timestamp. By default, Kafka Streams extracts the timestamp metadata embedded in the consumer record. You can change this default behavior by providing a different `TimestampExtractor` implementation per input binding. Here are some details on how that can be done.

```

@Bean
public Function<KStream<Long, Order>,
              Function<KTable<Long, Customer>,
                      Function<GlobalKTable<Long, Product>, KStream<Long, Order>>>>
process() {
    return orderStream ->
           customers ->
           products -> orderStream;
}

@Bean
public TimestampExtractor timestampExtractor() {
    return new WallclockTimestampExtractor();
}

```

Then you set the above `TimestampExtractor` bean name per consumer binding.

```

spring.cloud.stream.kafka.streams.bindings.process-in-0.consumer.timestampExtractorBeanName=timestampExtractor
spring.cloud.stream.kafka.streams.bindings.process-in-1.consumer.timestampExtractorBeanName=timestampExtractor
spring.cloud.stream.kafka.streams.bindings.process-in-2.consumer.timestampExtractorBeanName=timestampExtractor"

```

If you skip an input consumer binding for setting a custom timestamp extractor, that consumer will

use the default settings.

25.2.15. Multi binders with Kafka Streams based binders and regular Kafka Binder

You can have an application where you have both a function/consumer/supplier that is based on the regular Kafka binder and a Kafka Streams based processor. However, you cannot mix both of them within a single function or consumer.

Here is an example, where you have both binder based components within the same application.

```
@Bean
public Function<String, String> process() {
    return s -> s;
}

@Bean
public Function<KStream<Object, String>, KStream<?, WordCount>> kstreamProcess() {

    return input -> input;
}
```

This is the relevant parts from the configuration:

```
spring.cloud.stream.function.definition=process;kstreamProcess
spring.cloud.stream.bindings.process-in-0.destination=foo
spring.cloud.stream.bindings.process-out-0.destination=bar
spring.cloud.stream.bindings.kstreamProcess-in-0.destination=bar
spring.cloud.stream.bindings.kstreamProcess-out-0.destination=foobar
```

Things become a bit more complex if you have the same application as above, but is dealing with two different Kafka clusters, for e.g. the regular `process` is acting upon both Kafka cluster 1 and cluster 2 (receiving data from cluster-1 and sending to cluster-2) and the Kafka Streams processor is acting upon Kafka cluster 2. Then you have to use the [multi binder](#) facilities provided by Spring Cloud Stream.

Here is how your configuration may change in that scenario.

```

# multi binder configuration
spring.cloud.stream.binders.kafka1.type: kafka
spring.cloud.stream.binders.kafka1.environment.spring.cloud.stream.kafka.streams.binder.brokers=${kafkaCluster-1} #Replace kafkaCluster-1 with the appropriate IP of the cluster
spring.cloud.stream.binders.kafka2.type: kafka
spring.cloud.stream.binders.kafka2.environment.spring.cloud.stream.kafka.streams.binder.brokers=${kafkaCluster-2} #Replace kafkaCluster-2 with the appropriate IP of the cluster
spring.cloud.stream.binders.kafka3.type: kstream
spring.cloud.stream.binders.kafka3.environment.spring.cloud.stream.kafka.streams.binder.brokers=${kafkaCluster-2} #Replace kafkaCluster-2 with the appropriate IP of the cluster

spring.cloud.stream.function.definition=process;kstreamProcess

# From cluster 1 to cluster 2 with regular process function
spring.cloud.stream.bindings.process-in-0.destination=foo
spring.cloud.stream.bindings.process-in-0.binder=kafka1 # source from cluster 1
spring.cloud.stream.bindings.process-out-0.destination=bar
spring.cloud.stream.bindings.process-out-0.binder=kafka2 # send to cluster 2

# Kafka Streams processor on cluster 2
spring.cloud.stream.bindings.kstreamProcess-in-0.destination=bar
spring.cloud.stream.bindings.kstreamProcess-in-0.binder=kafka3
spring.cloud.stream.bindings.kstreamProcess-out-0.destination=foobar
spring.cloud.stream.bindings.kstreamProcess-out-0.binder=kafka3

```

Pay attention to the above configuration. We have two kinds of binders, but 3 binders all in all, first one is the regular Kafka binder based on cluster 1 (*kafka1*), then another Kafka binder based on cluster 2 (*kafka2*) and finally the *kstream* one (*kafka3*). The first processor in the application receives data from *kafka1* and publishes to *kafka2* where both binders are based on regular Kafka binder but different clusters. The second processor, which is a Kafka Streams processor consumes data from *kafka3* which is the same cluster as *kafka2*, but a different binder type.

Since there are three different binder types available in the Kafka Streams family of binders - *kstream*, *ktable* and *globalktable* - if your application has multiple bindings based on any of these binders, that needs to be explicitly provided as the binder type.

For e.g if you have a processor as below,

```

@Bean
public Function<KStream<Long, Order>,
    Function<KTable<Long, Customer>,
        Function<GlobalKTable<Long, Product>, KStream<Long, EnrichedOrder>>>>
    enrichOrder() {

    ...

}

```

then, this has to be configured in a multi binder scenario as the following. Please note that this is only needed if you have a true multi-binder scenario where there are multiple processors dealing with multiple clusters within a single application. In that case, the binders need to be explicitly provided with the bindings to distinguish from other processor's binder types and clusters.

```

spring.cloud.stream.binders.kafka1.type: kstream
spring.cloud.stream.binders.kafka1.environment.spring.cloud.stream.kafka.streams.binder.brokers=${kafkaCluster-2}
spring.cloud.stream.binders.kafka2.type: ktable
spring.cloud.stream.binders.kafka2.environment.spring.cloud.stream.kafka.streams.binder.brokers=${kafkaCluster-2}
spring.cloud.stream.binders.kafka3.type: globalktable
spring.cloud.stream.binders.kafka3.environment.spring.cloud.stream.kafka.streams.binder.brokers=${kafkaCluster-2}

spring.cloud.stream.bindings.enrichOrder-in-0.binder=kafka1 #kstream
spring.cloud.stream.bindings.enrichOrder-in-1.binder=kafka2 #ktable
spring.cloud.stream.bindings.enrichOrder-in-2.binder=kafka3 #globalktable
spring.cloud.stream.bindings.enrichOrder-out-0.binder=kafka1 #kstream

# rest of the configuration is omitted.

```

25.2.16. State Cleanup

By default, the `KafkaStreams.cleanup()` method is called when the binding is stopped. See [the Spring Kafka documentation](#). To modify this behavior simply add a single `CleanupConfig @Bean` (configured to clean up on start, stop, or neither) to the application context; the bean will be detected and wired into the factory bean.

25.2.17. Kafka Streams topology visualization

Kafka Streams binder provides the following actuator endpoints for retrieving the topology description using which you can visualize the topology using external tools.

`/actuator/topology`

`/actuator/topology/<application-id of the processor>`

You need to include the actuator and web dependencies from Spring Boot to access these endpoints.

Further, you also need to add `topology` to `management.endpoints.web.exposure.include` property. By default, the `topology` endpoint is disabled.

25.2.18. Configuration Options

This section contains the configuration options used by the Kafka Streams binder.

For common configuration options and properties pertaining to binder, refer to the [core documentation](#).

Kafka Streams Binder Properties

The following properties are available at the binder level and must be prefixed with `spring.cloud.stream.kafka.streams.binder.`

configuration

Map with a key/value pair containing properties pertaining to Apache Kafka Streams API. This property must be prefixed with `spring.cloud.stream.kafka.streams.binder.`. Following are some examples of using this property.

```
spring.cloud.stream.kafka.streams.binder.configuration.default.key.serde=org.apache.kafk
a.common.serialization.Serdes$stringSerde
spring.cloud.stream.kafka.streams.binder.configuration.default.value.serde=org.apache.
kafka.common.serialization.Serdes$stringSerde
spring.cloud.stream.kafka.streams.binder.configuration.commit.interval.ms=1000
```

For more information about all the properties that may go into streams configuration, see [StreamsConfig](#) JavaDocs in Apache Kafka Streams docs. All configuration that you can set from [StreamsConfig](#) can be set through this. When using this property, it is applicable against the entire application since this is a binder level property. If you have more than processors in the application, all of them will acquire these properties. In the case of properties like `application.id`, this will become problematic and therefore you have to carefully examine how the properties from [StreamsConfig](#) are mapped using this binder level `configuration` property.

functions.<function-bean-name>.applicationId

Applicable only for functional style processors. This can be used for setting application ID per function in the application. In the case of multiple functions, this is a handy way to set the application ID.

functions.<function-bean-name>.configuration

Applicable only for functional style processors. Map with a key/value pair containing properties pertaining to Apache Kafka Streams API. This is similar to the binder level `configuration` property describe above, but this level of `configuration` property is restricted only against the named function. When you have multiple processors and you want to restrict access to the configuration based on particular functions, you might want to use this. All [StreamsConfig](#) properties can be used here.

brokers

Broker URL

Default: `localhost`

zkNodes

Zookeeper URL

Default: `localhost`

deserializationExceptionHandler

Deserialization error handler type. This handler is applied at the binder level and thus applied against all input binding in the application. There is a way to control it in a more fine-grained way at the consumer binding level. Possible values are - `logAndContinue`, `logAndFail` or `sendToDlq`

Default: `logAndFail`

applicationId

Convenient way to set the application.id for the Kafka Streams application globally at the binder level. If the application contains multiple functions or `StreamListener` methods, then the application id should be set differently. See above where setting the application id is discussed in detail.

Default: application will generate a static application ID. See the application ID section for more details.

stateStoreRetry.maxAttempts

Max attempts for trying to connect to a state store.

Default: 1

stateStoreRetry.backoffPeriod

Backoff period when trying to connect to a state store on a retry.

Default: 1000 ms

Kafka Streams Producer Properties

The following properties are *only* available for Kafka Streams producers and must be prefixed with `spring.cloud.stream.kafka.streams.bindings.<binding name>.producer`. For convenience, if there are multiple output bindings and they all require a common value, that can be configured by using the prefix `spring.cloud.stream.kafka.streams.default.producer..`

keySerde

key serde to use

Default: See the above discussion on message de/serialization

valueSerde

value serde to use

Default: See the above discussion on message de/serialization

useNativeEncoding

flag to enable/disable native encoding

Default: `true`.

`streamPartitionerBeanName`: Custom outbound partitioner bean name to be used at the consumer. Applications can provide custom `StreamPartitioner` as a Spring bean and the name of this bean can be provided to the producer to use instead of the default one.

+ Default: See the discussion above on outbound partition support.

Kafka Streams Consumer Properties

The following properties are available for Kafka Streams consumers and must be prefixed with `spring.cloud.stream.kafka.streams.bindings.<binding-name>.consumer`. For convenience, if there are multiple input bindings and they all require a common value, that can be configured by using the prefix `spring.cloud.stream.kafka.streams.default.consumer..`

applicationId

Setting `application.id` per input binding. This is only preferred for `StreamListener` based processors, for function based processors see other approaches outlined above.

Default: See above.

keySerde

key serde to use

Default: See the above discussion on message de/serialization

valueSerde

value serde to use

Default: See the above discussion on message de/serialization

materializedAs

state store to materialize when using incoming `KTable` types

Default: `none`.

useNativeDecoding

flag to enable/disable native decoding

Default: `true`.

dlqName

DLQ topic name.

Default: See above on the discussion of error handling and DLQ.

startOffset

Offset to start from if there is no committed offset to consume from. This is mostly used when the consumer is consuming from a topic for the first time. Kafka Streams uses `earliest` as the default strategy and the binder uses the same default. This can be overridden to `latest` using this property.

Default: `earliest`.

Note: Using `resetOffsets` on the consumer does not have any effect on Kafka Streams binder. Unlike the message channel based binder, Kafka Streams binder does not seek to beginning or end on demand.

deserializationExceptionHandler

Deserialization error handler type. This handler is applied per consumer binding as opposed to the binder level property described before. Possible values are - `logAndContinue`, `logAndFail` or `sendToDlq`

Default: `logAndFail`

timestampExtractorBeanName

Specific time stamp extractor bean name to be used at the consumer. Applications can provide `TimestampExtractor` as a Spring bean and the name of this bean can be provided to the consumer to use instead of the default one.

Default: See the discussion above on timestamp extractors.

Special note on concurrency

In Kafka Streams, you can control of the number of threads a processor can create using the `num.stream.threads` property. This, you can do using the various `configuration` options described above under binder, functions, producer or consumer level. You can also use the `concurrency` property that core Spring Cloud Stream provides for this purpose. When using this, you need to use it on the consumer. When you have more than one input bindings either in a function or `StreamListener`, set this on the first input binding. For e.g. when setting `spring.cloud.stream.bindings.process-in-0.consumer.concurrency`, it will be translated as `num.stream.threads` by the binder.

Chapter 26. Appendices

Appendix D: Building

26.D.1. Basic Compile and Test

To build the source you will need to install JDK 1.7.

The build uses the Maven wrapper so you don't have to install a specific version of Maven. To enable the tests, you should have Kafka server 0.9 or above running before building. See below for more information on running the servers.

The main build command is

```
$ ./mvnw clean install
```

You can also add '-DskipTests' if you like, to avoid running the tests.



You can also install Maven (>=3.3.3) yourself and run the `mvn` command in place of `./mvnw` in the examples below. If you do that you also might need to add `-P spring` if your local Maven settings do not contain repository declarations for spring pre-release artifacts.



Be aware that you might need to increase the amount of memory available to Maven by setting a `MAVEN_OPTS` environment variable with a value like `-Xmx512m -XX:MaxPermSize=128m`. We try to cover this in the `.mvn` configuration, so if you find you have to do it to make a build succeed, please raise a ticket to get the settings added to source control.

The projects that require middleware generally include a `docker-compose.yml`, so consider using [Docker Compose](#) to run the middleware servers in Docker containers.

26.D.2. Documentation

There is a "full" profile that will generate documentation.

26.D.3. Working with the code

If you don't have an IDE preference we would recommend that you use [Spring Tools Suite](#) or [Eclipse](#) when working with the code. We use the [m2eclipse](#) eclipse plugin for maven support. Other IDEs and tools should also work without issue.

Importing into eclipse with m2eclipse

We recommend the [m2eclipse](#) eclipse plugin when working with eclipse. If you don't already have m2eclipse installed it is available from the "eclipse marketplace".

Unfortunately m2e does not yet support Maven 3.3, so once the projects are imported into Eclipse you will also need to tell m2eclipse to use the `.settings.xml` file for the projects. If you do not do this you may see many different errors related to the POMs in the projects. Open your Eclipse preferences, expand the Maven preferences, and select User Settings. In the User Settings field click Browse and navigate to the Spring Cloud project you imported selecting the `.settings.xml` file in that project. Click Apply and then OK to save the preference changes.



Alternatively you can copy the repository settings from `.settings.xml` into your own `~/.m2/settings.xml`.

Importing into eclipse without m2eclipse

If you prefer not to use m2eclipse you can generate eclipse project metadata using the following command:

```
$ ./mvnw eclipse:eclipse
```

The generated eclipse projects can be imported by selecting `import existing projects` from the `file` menu.

[[contributing] == Contributing

Spring Cloud is released under the non-restrictive Apache 2.0 license, and follows a very standard Github development process, using Github tracker for issues and merging pull requests into master. If you want to contribute even something trivial please do not hesitate, but follow the guidelines below.

26.D.4. Sign the Contributor License Agreement

Before we accept a non-trivial patch or pull request we will need you to sign the [contributor's agreement](#). Signing the contributor's agreement does not grant anyone commit rights to the main repository, but it does mean that we can accept your contributions, and you will get an author credit if we do. Active contributors might be asked to join the core team, and given the ability to merge pull requests.

26.D.5. Code Conventions and Housekeeping

None of these is essential for a pull request, but they will all help. They can also be added after the original pull request but before a merge.

- Use the Spring Framework code format conventions. If you use Eclipse you can import formatter settings using the `eclipse-code-formatter.xml` file from the [Spring Cloud Build](#) project. If using IntelliJ, you can use the [Eclipse Code Formatter Plugin](#) to import the same file.
- Make sure all new `.java` files to have a simple Javadoc class comment with at least an `@author` tag identifying you, and preferably at least a paragraph on what the class is for.
- Add the ASF license header comment to all new `.java` files (copy from existing files in the project)

- Add yourself as an `@author` to the .java files that you modify substantially (more than cosmetic changes).
- Add some Javadocs and, if you change the namespace, some XSD doc elements.
- A few unit tests would help a lot as well — someone has to do it.
- If no-one else is using your branch, please rebase it against the current master (or other target branch in the main project).
- When writing a commit message please follow [these conventions](#), if you are fixing an existing issue please add `Fixes gh-XXXX` at the end of the commit message (where XXXX is the issue number).

Appendix: Compendium of Configuration Properties

Name	Default	Description
encrypt.fail-on-error	true	Flag to say that a process should fail if there is an encryption or decryption error.
encrypt.key		A symmetric key. As a stronger alternative, consider using a keystore.
encrypt.key-store.alias		Alias for a key in the store.
encrypt.key-store.location		Location of the key store file, e.g. classpath:/keystore.jks.
encrypt.key-store.password		Password that locks the keystore.
encrypt.key-store.secret		Secret protecting the key (defaults to the same as the password).
encrypt.key-store.type	jks	The KeyStore type. Defaults to jks.
encrypt.rsa.algorithm		The RSA algorithm to use (DEFAULT or OEAP). Once it is set, do not change it (or existing ciphers will not be decryptable).
encrypt.rsa.salt	deadbeef	Salt for the random secret used to encrypt cipher text. Once it is set, do not change it (or existing ciphers will not be decryptable).
encrypt.rsa.strong	false	Flag to indicate that "strong" AES encryption should be used internally. If true, then the GCM algorithm is applied to the AES encrypted bytes. Default is false (in which case "standard" CBC is used instead). Once it is set, do not change it (or existing ciphers will not be decryptable).

Name	Default	Description
encrypt.salt	deadbeef	A salt for the symmetric key, in the form of a hex-encoded byte array. As a stronger alternative, consider using a keystore.
endpoints.zookeeper.enabled	true	Enable the /zookeeper endpoint to inspect the state of zookeeper.
health.config.enabled	false	Flag to indicate that the config server health indicator should be installed.
health.config.time-to-live	0	Time to live for cached result, in milliseconds. Default 300000 (5 min).
management.endpoint.bindings.cache.time-to-live	0ms	Maximum time that a response can be cached.
management.endpoint.bindings.enabled	true	Whether to enable the bindings endpoint.
management.endpoint.bus-env.enabled	true	Whether to enable the bus-env endpoint.
management.endpoint.bus-refresh.enabled	true	Whether to enable the bus-refresh endpoint.
management.endpoint.channels.cache.time-to-live	0ms	Maximum time that a response can be cached.
management.endpoint.channels.enabled	true	Whether to enable the channels endpoint.
management.endpoint.consul.cache.time-to-live	0ms	Maximum time that a response can be cached.
management.endpoint.consul.enabled	true	Whether to enable the consul endpoint.
management.endpoint.env.post.enabled	false	Enables writable environment endpoint.
management.endpoint.features.cache.time-to-live	0ms	Maximum time that a response can be cached.
management.endpoint.features.enabled	true	Whether to enable the features endpoint.
management.endpoint.gateway.enabled	true	Whether to enable the gateway endpoint.
management.endpoint.pause.enabled	true	Enable the /pause endpoint (to send Lifecycle.stop()).

Name	Default	Description
management.endpoint.refresh.enabled	true	Enable the /refresh endpoint to refresh configuration and re-initialize refresh scoped beans.
management.endpoint.restart.enabled	true	Enable the /restart endpoint to restart the application context.
management.endpoint.resume.enabled	true	Enable the /resume endpoint (to send Lifecycle.start()).
management.endpoint.serviceregistry.cache.time-to-live	0ms	Maximum time that a response can be cached.
management.endpoint.serviceregistry.enabled	true	Whether to enable the serviceregistry endpoint.
management.endpoint.topology.cache.time-to-live	0ms	Maximum time that a response can be cached.
management.endpoint.topology.enabled	true	Whether to enable the topology endpoint.
management.health.binders.enabled	true	Allows to enable/disable binder's health indicators. If you want to disable health indicator completely, then set it to false .
management.health.refresh.enabled	true	Enable the health endpoint for the refresh scope.
management.health.zookeeper.enabled	true	Enable the health endpoint for zookeeper.
spring.cloud.bus.ack.destination-service		Service that wants to listen to acks. By default null (meaning all services).
spring.cloud.bus.ack.enabled	true	Flag to switch off acks (default on).
spring.cloud.bus.destination	springCloudBus	Name of Spring Cloud Stream destination for messages.
spring.cloud.bus.enabled	true	Flag to indicate that the bus is enabled.
spring.cloud.bus.env.enabled	true	Flag to switch off environment change events (default on).
spring.cloud.bus.id	application	The identifier for this application instance.
spring.cloud.bus.refresh.enabled	true	Flag to switch off refresh events (default on).

Name	Default	Description
spring.cloud.bus.trace.enabled	false	Flag to switch on tracing of acks (default off).
spring.cloud.cloudfoundry.discovery.default-server-port	80	Port to use when no port is defined by service discovery.
spring.cloud.cloudfoundry.discovery.enabled	true	Flag to indicate that discovery is enabled.
spring.cloud.cloudfoundry.discovery.heartbeat-frequency	5000	Frequency in milliseconds of poll for heart beat. The client will poll on this frequency and broadcast a list of service ids.
spring.cloud.cloudfoundry.discovery.internal-domain	apps.internal	Default internal domain when configured to use Native DNS service discovery.
spring.cloud.cloudfoundry.discovery.order	0	Order of the discovery client used by <code>CompositeDiscoveryClient</code> for sorting available clients.
spring.cloud.cloudfoundry.discovery.use-container-ip	false	Whether to resolve hostname when BOSH DNS is used. In order to use this feature, <code>spring.cloud.cloudfoundry.discovery.use-dns</code> must be true.
spring.cloud.cloudfoundry.discovery.use-dns	false	Whether to use BOSH DNS for the discovery. In order to use this feature, your Cloud Foundry installation must support Service Discovery.
spring.cloud.cloudfoundry.org		Organization name to initially target.
spring.cloud.cloudfoundry.password		Password for user to authenticate and obtain token.
spring.cloud.cloudfoundry.skip-ssl-validation	false	null
spring.cloud.cloudfoundry.space		Space name to initially target.
spring.cloud.cloudfoundry.url		URL of Cloud Foundry API (Cloud Controller).
spring.cloud.cloudfoundry.username		Username to authenticate (usually an email address).

Name	Default	Description
spring.cloud.compatibility-verifier.compatible-boot-versions		Default accepted versions for the Spring Boot dependency. You can set <code>{@code x}</code> for the patch version if you don't want to specify a concrete value. Example: <code>{@code 3.4.x}</code>
spring.cloud.compatibility-verifier.enabled	false	Enables creation of Spring Cloud compatibility verification.
spring.cloud.config.allow-override	true	Flag to indicate that <code>{@link #isOverrideSystemProperties() systemPropertiesOverride}</code> can be used. Set to false to prevent users from changing the default accidentally. Default true.
spring.cloud.config.discovery.enabled	false	Flag to indicate that config server discovery is enabled (config server URL will be looked up via discovery).
spring.cloud.config.discovery.service-id	configserver	Service id to locate config server.
spring.cloud.config.enabled	true	Flag to say that remote configuration is enabled. Default true;
spring.cloud.config.fail-fast	false	Flag to indicate that failure to connect to the server is fatal (default false).
spring.cloud.config.headers		Additional headers used to create the client request.
spring.cloud.config.label		The label name to use to pull remote configuration properties. The default is set on the server (generally "master" for a git based server).
spring.cloud.config.name		Name of application used to fetch remote properties.

Name	Default	Description
spring.cloud.config.override-none	false	Flag to indicate that when <code>{@link #setAllowOverride(boolean) allowOverride}</code> is true, external properties should take lowest priority and should not override any existing property sources (including local config files). Default false.
spring.cloud.config.override-system-properties	true	Flag to indicate that the external properties should override system properties. Default true.
spring.cloud.config.password		The password to use (HTTP Basic) when contacting the remote server.
spring.cloud.config.profile	default	The default profile to use when fetching remote configuration (comma-separated). Default is "default".
spring.cloud.config.request-connect-timeout	0	timeout on waiting to connect to the Config Server.
spring.cloud.config.request-read-timeout	0	timeout on waiting to read data from the Config Server.
spring.cloud.config.retry.initial-interval	1000	Initial retry interval in milliseconds.
spring.cloud.config.retry.max-attempts	6	Maximum number of attempts.
spring.cloud.config.retry.max-interval	2000	Maximum interval for backoff.
spring.cloud.config.retry.multiplier	1.1	Multiplier for next interval.
spring.cloud.config.send-state	true	Flag to indicate whether to send state. Default true.
spring.cloud.config.server.accept-empty	true	Flag to indicate that If HTTP 404 needs to be sent if Application is not Found.
spring.cloud.config.server.awss3.bucket		Name of the S3 bucket that contains config.
spring.cloud.config.server.awss3.order	0	null

Name	Default	Description
spring.cloud.config.server.awss3.region		AWS region that contains config.
spring.cloud.config.server.boots-trap	false	Flag indicating that the config server should initialize its own Environment with properties from the remote repository. Off by default because it delays startup but can be useful when embedding the server in another application.
spring.cloud.config.server.credhub.ca-cert-files		null
spring.cloud.config.server.credhub.connection-timeout		null
spring.cloud.config.server.credhub.oauth2.registration-id		null
spring.cloud.config.server.credhub.order		null
spring.cloud.config.server.credhub.read-timeout		null
spring.cloud.config.server.credhub.url		null
spring.cloud.config.server.default-application-name	application	Default application name when incoming requests do not have a specific one.
spring.cloud.config.server.default-label		Default repository label when incoming requests do not have a specific label.
spring.cloud.config.server.default-profile	default	Default application profile when incoming requests do not have a specific one.
spring.cloud.config.server.encrypted.enabled	true	Enable decryption of environment properties before sending to client.
spring.cloud.config.server.encrypted.plain-text-encrypt	false	Enable decryption of environment properties served by plain text endpoint {@link org.springframework.cloud.config.server.resource.ResourceController}.

Name	Default	Description
spring.cloud.config.server.git.basedir		Base directory for local working copy of repository.
spring.cloud.config.server.git.clone-on-start	false	Flag to indicate that the repository should be cloned on startup (not on demand). Generally leads to slower startup but faster first query.
spring.cloud.config.server.git.default-label		The default label to be used with the remote repository.
spring.cloud.config.server.git.delete-untracked-branches	false	Flag to indicate that the branch should be deleted locally if its origin tracked branch was removed.
spring.cloud.config.server.git.force-pull	false	Flag to indicate that the repository should force pull. If true discard any local changes and take from remote repository.
spring.cloud.config.server.git.host-key		Valid SSH host key. Must be set if hostKeyAlgorithm is also set.
spring.cloud.config.server.git.host-key-algorithm		One of ssh-dss, ssh-rsa, ecdsa-sha2-nistp256, ecdsa-sha2-nistp384, or ecdsa-sha2-nistp521. Must be set if hostKey is also set.
spring.cloud.config.server.git.ignore-local-ssh-settings	false	If true, use property-based instead of file-based SSH config.
spring.cloud.config.server.git.known-hosts-file		Location of custom .known_hosts file.
spring.cloud.config.server.git.order		The order of the environment repository.
spring.cloud.config.server.git.passphrase		Passphrase for unlocking your ssh private key.
spring.cloud.config.server.git.password		Password for authentication with remote repository.
spring.cloud.config.server.git.preferred-authentications		Override server authentication method order. This should allow for evading login prompts if server has keyboard-interactive authentication before the publickey method.

Name	Default	Description
spring.cloud.config.server.git.private-key		Valid SSH private key. Must be set if ignoreLocalSshSettings is true and Git URI is SSH format.
spring.cloud.config.server.git.proxy		HTTP proxy configuration.
spring.cloud.config.server.git.refresh-rate	0	Time (in seconds) between refresh of the git repository.
spring.cloud.config.server.git.repos		Map of repository identifier to location and other properties.
spring.cloud.config.server.git.search-paths		Search paths to use within local working copy. By default searches only the root.
spring.cloud.config.server.git.skip-ssl-validation	false	Flag to indicate that SSL certificate validation should be bypassed when communicating with a repository served over an HTTPS connection.
spring.cloud.config.server.git.strict-host-key-checking	true	If false, ignore errors with host key.
spring.cloud.config.server.git.timeout	5	Timeout (in seconds) for obtaining HTTP or SSH connection (if applicable), defaults to 5 seconds.
spring.cloud.config.server.git.uri		URI of remote repository.
spring.cloud.config.server.git.username		Username for authentication with remote repository.
spring.cloud.config.server.health.repositories		null
spring.cloud.config.server.jdbc.order	0	null
spring.cloud.config.server.jdbc.sql	SELECT KEY, VALUE from PROPERTIES where APPLICATION=? and PROFILE=? and LABEL=?	SQL used to query database for keys and values.
spring.cloud.config.server.native.add-label-locations	true	Flag to determine whether label locations should be added.
spring.cloud.config.server.native.default-label	master	null

Name	Default	Description
spring.cloud.config.server.native.fail-on-error	false	Flag to determine how to handle exceptions during decryption (default false).
spring.cloud.config.server.native.order		null
spring.cloud.config.server.native.search-locations	[]	Locations to search for configuration files. Defaults to the same as a Spring Boot app so [classpath:/,classpath:/config/,file:/,file:./config/].
spring.cloud.config.server.native.version		Version string to be reported for native repository.
spring.cloud.config.server.overrides		Extra map for a property source to be sent to all clients unconditionally.
spring.cloud.config.server.prefix		Prefix for configuration resource paths (default is empty). Useful when embedding in another application when you don't want to change the context path or servlet path.
spring.cloud.config.server.redis.order		null
spring.cloud.config.server.strip-document-from-yaml	true	Flag to indicate that YAML documents that are text or collections (not a map) should be returned in "native" form.
spring.cloud.config.server.svn.basedir		Base directory for local working copy of repository.
spring.cloud.config.server.svn.default-label		The default label to be used with the remote repository.
spring.cloud.config.server.svn.order		The order of the environment repository.
spring.cloud.config.server.svn.passphrase		Passphrase for unlocking your ssh private key.
spring.cloud.config.server.svn.password		Password for authentication with remote repository.

Name	Default	Description
spring.cloud.config.server.svn.search-paths		Search paths to use within local working copy. By default searches only the root.
spring.cloud.config.server.svn.strict-host-key-checking	true	Reject incoming SSH host keys from remote servers not in the known host list.
spring.cloud.config.server.svn.uri		URI of remote repository.
spring.cloud.config.server.svn.username		Username for authentication with remote repository.
spring.cloud.config.server.vault.app-role.app-role-path	approle	Mount path of the AppRole authentication backend.
spring.cloud.config.server.vault.app-role.role		Name of the role, optional, used for pull-mode.
spring.cloud.config.server.vault.app-role.role-id		The RoleId.
spring.cloud.config.server.vault.app-role.secret-id		The SecretId.
spring.cloud.config.server.vault.authentication		null
spring.cloud.config.server.vault.aws-ec2.aws-ec2-path	aws-ec2	Mount path of the AWS-EC2 authentication backend.
spring.cloud.config.server.vault.aws-ec2.identity-document	169.254.169.254/latest/dynamic/instance-identity/pkcs7	URL of the AWS-EC2 PKCS7 identity document.
spring.cloud.config.server.vault.aws-ec2.nonce		Nonce used for AWS-EC2 authentication. An empty nonce defaults to nonce generation.
spring.cloud.config.server.vault.aws-ec2.role		Name of the role, optional.
spring.cloud.config.server.vault.aws-iam.aws-path	aws	Mount path of the AWS authentication backend.
spring.cloud.config.server.vault.aws-iam.endpoint-uri		STS server URI. @since 2.2
spring.cloud.config.server.vault.aws-iam.role		Name of the role, optional. Defaults to the friendly IAM name if not set.
spring.cloud.config.server.vault.aws-iam.server-name		Name of the server used to set {@code X-Vault-AWS-IAM-Server-ID} header in the headers of login requests.

Name	Default	Description
spring.cloud.config.server.vault.azure-msi.azure-path	azure	Mount path of the Azure MSI authentication backend.
spring.cloud.config.server.vault.azure-msi.identity-token-service		URI to the Azure MSI Identity Service.
spring.cloud.config.server.vault.azure-msi.metadata-service		URI to the Azure MSI Metadata Service.
spring.cloud.config.server.vault.azure-msi.role		Name of the role.
spring.cloud.config.server.vault.backend	secret	Vault backend. Defaults to secret.
spring.cloud.config.server.vault.default-key	application	The key in vault shared by all applications. Defaults to application. Set to empty to disable.
spring.cloud.config.server.vault.gcp-gce.gcp-path	gcp	Mount path of the Kubernetes authentication backend.
spring.cloud.config.server.vault.gcp-gce.role		Name of the role against which the login is being attempted.
spring.cloud.config.server.vault.gcp-gce.service-account		Optional service account id. Using the default id if left unconfigured.
spring.cloud.config.server.vault.gcp-iam.credentials.encoded-key		The base64 encoded contents of an OAuth2 account private key in JSON format.
spring.cloud.config.server.vault.gcp-iam.credentials.location		Location of the OAuth2 credentials private key. <p>Since this is a Resource, the private key can be in a multitude of locations, such as a local file system, classpath, URL, etc.
spring.cloud.config.server.vault.gcp-iam.gcp-path	gcp	Mount path of the Kubernetes authentication backend.
spring.cloud.config.server.vault.gcp-iam.jwt-validity	15m	Validity of the JWT token.
spring.cloud.config.server.vault.gcp-iam.project-id		Overrides the GCP project Id.
spring.cloud.config.server.vault.gcp-iam.role		Name of the role against which the login is being attempted.

Name	Default	Description
spring.cloud.config.server.vault.gcp-iam.service-account-id		Overrides the GCP service account Id.
spring.cloud.config.server.vault.host	127.0.0.1	Vault host. Defaults to 127.0.0.1.
spring.cloud.config.server.vault.kubernetes.kubernetes-path	kubernetes	Mount path of the Kubernetes authentication backend.
spring.cloud.config.server.vault.kubernetes.role		Name of the role against which the login is being attempted.
spring.cloud.config.server.vault.kubernetes.service-account-token-file	/var/run/secrets/kubernetes.io/serviceaccount/token	Path to the service account token file.
spring.cloud.config.server.vault.kv-version	1	Value to indicate which version of Vault kv backend is used. Defaults to 1.
spring.cloud.config.server.vault.namespace		The value of the Vault X-Vault-Namespace header. Defaults to null. This a Vault Enterprise feature only.
spring.cloud.config.server.vault.order		null
spring.cloud.config.server.vault.pcf.instance-certificate		Path to the instance certificate (PEM). Defaults to {@code CF_INSTANCE_CERT} env variable.
spring.cloud.config.server.vault.pcf.instance-key		Path to the instance key (PEM). Defaults to {@code CF_INSTANCE_KEY} env variable.
spring.cloud.config.server.vault.pcf.pcf-path	pcf	Mount path of the Kubernetes authentication backend.
spring.cloud.config.server.vault.pcf.role		Name of the role against which the login is being attempted.
spring.cloud.config.server.vault.port	8200	Vault port. Defaults to 8200.
spring.cloud.config.server.vault.profile-separator	,	Vault profile separator. Defaults to comma.
spring.cloud.config.server.vault.proxy		HTTP proxy configuration.
spring.cloud.config.server.vault.scheme	http	Vault scheme. Defaults to http.

Name	Default	Description
spring.cloud.config.server.vault.skip-ssl-validation	false	Flag to indicate that SSL certificate validation should be bypassed when communicating with a repository served over an HTTPS connection.
spring.cloud.config.server.vault.ssl.cert-auth-path	cert	Mount path of the TLS cert authentication backend.
spring.cloud.config.server.vault.ssl.key-store		Trust store that holds certificates and private keys.
spring.cloud.config.server.vault.ssl.key-store-password		Password used to access the key store.
spring.cloud.config.server.vault.ssl.trust-store		Trust store that holds SSL certificates.
spring.cloud.config.server.vault.ssl.trust-store-password		Password used to access the trust store.
spring.cloud.config.server.vault.timeout	5	Timeout (in seconds) for obtaining HTTP connection, defaults to 5 seconds.
spring.cloud.config.server.vault.token		Static vault token. Required if {@link #authentication} is {@code TOKEN}.
spring.cloud.config.token		Security Token passed thru to underlying environment repository.
spring.cloud.config.uri	[localhost:8888]	The URI of the remote server (default localhost:8888).
spring.cloud.config.username		The username to use (HTTP Basic) when contacting the remote server.
spring.cloud.consul.config.acl-token		null
spring.cloud.consul.config.data-key	data	If format is Format.PROPERTIES or Format.YAML then the following field is used as key to look up consul for configuration.
spring.cloud.consul.config.default-context	application	null
spring.cloud.consul.config.enabled	true	null

Name	Default	Description
spring.cloud.consul.config.fail-fast	true	Throw exceptions during config lookup if true, otherwise, log warnings.
spring.cloud.consul.config.format		null
spring.cloud.consul.config.name		Alternative to spring.application.name to use in looking up values in consul KV.
spring.cloud.consul.config.prefix	config	null
spring.cloud.consul.config.profile-separator	,	null
spring.cloud.consul.config.watch.delay	1000	The value of the fixed delay for the watch in millis. Defaults to 1000.
spring.cloud.consul.config.watch.enabled	true	If the watch is enabled. Defaults to true.
spring.cloud.consul.config.watch.wait-time	55	The number of seconds to wait (or block) for watch query, defaults to 55. Needs to be less than default ConsulClient (defaults to 60). To increase ConsulClient timeout create a ConsulClient bean with a custom ConsulRawClient with a custom HttpClient.
spring.cloud.consul.discovery.actoken		null
spring.cloud.consul.discovery.catalog-services-watch-delay	1000	The delay between calls to watch consul catalog in millis, default is 1000.
spring.cloud.consul.discovery.catalog-services-watch-timeout	2	The number of seconds to block while watching consul catalog, default is 2.
spring.cloud.consul.discovery.consistency-mode		Consistency mode for health service request.
spring.cloud.consul.discovery.datacenters		Map of serviceId's → datacenter to query for in server list. This allows looking up services in another datacenters.

Name	Default	Description
spring.cloud.consul.discovery.default-query-tag		Tag to query for in service list if one is not listed in serverListQueryTags.
spring.cloud.consul.discovery.default-zone-metadata-name	zone	Service instance zone comes from metadata. This allows changing the metadata tag name.
spring.cloud.consul.discovery.disable-register	true	Disable automatic de-registration of service in consul.
spring.cloud.consul.discovery.enable-tag-override		Enable tag override for the registered service.
spring.cloud.consul.discovery.enabled	true	Is service discovery enabled?
spring.cloud.consul.discovery.fail-fast	true	Throw exceptions during service registration if true, otherwise, log warnings (defaults to true).
spring.cloud.consul.discovery.health-check-critical-timeout		Timeout to deregister services critical for longer than timeout (e.g. 30m). Requires consul version 7.x or higher.
spring.cloud.consul.discovery.health-check-headers		Headers to be applied to the Health Check calls.
spring.cloud.consul.discovery.health-check-interval	10s	How often to perform the health check (e.g. 10s), defaults to 10s.
spring.cloud.consul.discovery.health-check-path	/actuator/health	Alternate server path to invoke for health checking.
spring.cloud.consul.discovery.health-check-timeout		Timeout for health check (e.g. 10s).
spring.cloud.consul.discovery.health-check-tls-skip-verify		Skips certificate verification during service checks if true, otherwise runs certificate verification.
spring.cloud.consul.discovery.health-check-url		Custom health check url to override default.
spring.cloud.consul.discovery.heartbeat.enabled	false	null
spring.cloud.consul.discovery.heartbeat.interval-ratio		null

Name	Default	Description
spring.cloud.consul.discovery.heartbeat.ttl	30s	null
spring.cloud.consul.discovery.hostname		Hostname to use when accessing server.
spring.cloud.consul.discovery.include-hostname-in-instance-id	false	Whether hostname is included into the default instance id when registering service.
spring.cloud.consul.discovery.instance-group		Service instance group.
spring.cloud.consul.discovery.instance-id		Unique service instance id.
spring.cloud.consul.discovery.instance-zone		Service instance zone.
spring.cloud.consul.discovery.ip-address		IP address to use when accessing service (must also set preferIpAddress to use).
spring.cloud.consul.discovery.lifecycle.enabled	true	null
spring.cloud.consul.discovery.management-enable-tag-override		Enable tag override for the registered management service.
spring.cloud.consul.discovery.management-metadata		Metadata to use when registering management service.
spring.cloud.consul.discovery.management-port		Port to register the management service under (defaults to management port).
spring.cloud.consul.discovery.management-suffix	management	Suffix to use when registering management service.
spring.cloud.consul.discovery.management-tags		Tags to use when registering management service.
spring.cloud.consul.discovery.metadata		Metadata to use when registering service.
spring.cloud.consul.discovery.order	0	Order of the discovery client used by <code>CompositeDiscoveryClient</code> for sorting available clients.
spring.cloud.consul.discovery.port		Port to register the service under (defaults to listening port).

Name	Default	Description
spring.cloud.consul.discovery.p refer-agent-address	false	Source of how we will determine the address to use.
spring.cloud.consul.discovery.p refer-ip-address	false	Use ip address rather than hostname during registration.
spring.cloud.consul.discovery.q query-passing	false	Add the 'passing` parameter to /v1/health/service/serviceName. This pushes health check passing to the server.
spring.cloud.consul.discovery.r register	true	Register as a service in consul.
spring.cloud.consul.discovery.r register-health-check	true	Register health check in consul. Useful during development of a service.
spring.cloud.consul.discovery.sc heme	http	Whether to register an http or https service.
spring.cloud.consul.discovery.s erver-list-query-tags		Map of serviceId's → tag to query for in server list. This allows filtering services by a single tag.
spring.cloud.consul.discovery.s ervice-name		Service name.
spring.cloud.consul.discovery.ta gs		Tags to use when registering service.
spring.cloud.consul.enabled	true	Is spring cloud consul enabled.
spring.cloud.consul.host	localhost	Consul agent hostname. Defaults to 'localhost'.
spring.cloud.consul.port	8500	Consul agent port. Defaults to '8500'.
spring.cloud.consul.retry.enabl ed	true	If consul retry is enabled.
spring.cloud.consul.retry.initial- interval	1000	Initial retry interval in milliseconds.
spring.cloud.consul.retry.max- attempts	6	Maximum number of attempts.
spring.cloud.consul.retry.max- interval	2000	Maximum interval for backoff.
spring.cloud.consul.retry.multip lier	1.1	Multiplier for next interval.

Name	Default	Description
spring.cloud.consul.scheme		Consul agent scheme (HTTP/HTTPS). If there is no scheme in address - client will use HTTP.
spring.cloud.consul.service-registry.auto-registration.enabled	true	Enables Consul Service Registry Auto-registration.
spring.cloud.consul.service-registry.enabled	true	Enables Consul Service Registry functionality.
spring.cloud.consul.tls.certificate-password		Password to open the certificate.
spring.cloud.consul.tls.certificate-path		File path to the certificate.
spring.cloud.consul.tls.key-store-instance-type		Type of key framework to use.
spring.cloud.consul.tls.key-store-password		Password to an external keystore.
spring.cloud.consul.tls.key-store-path		Path to an external keystore.
spring.cloud.discovery.client.cloudfoundry.order		null
spring.cloud.discovery.client.composite-indicator.enabled	true	Enables discovery client composite health indicator.
spring.cloud.discovery.client.health-indicator.enabled	true	null
spring.cloud.discovery.client.health-indicator.include-description	false	null
spring.cloud.discovery.client.simple.instances		null
spring.cloud.discovery.client.simple.local.instance-id		The unique identifier or name for the service instance.
spring.cloud.discovery.client.simple.local.metadata		Metadata for the service instance. Can be used by discovery clients to modify their behaviour per instance, e.g. when load balancing.

Name	Default	Description
spring.cloud.discovery.client.simple.local.service-id		The identifier or name for the service. Multiple instances might share the same service ID.
spring.cloud.discovery.client.simple.local.uri		The URI of the service instance. Will be parsed to extract the scheme, host, and port.
spring.cloud.discovery.client.simple.order		null
spring.cloud.discovery.enabled	true	Enables discovery client health indicators.
spring.cloud.features.enabled	true	Enables the features endpoint.
spring.cloud.function.compile		Configuration for function bodies, which will be compiled. The key in the map is the function name and the value is a map containing a key "lambda" which is the body to compile, and optionally a "type" (defaults to "function"). Can also contain "inputType" and "outputType" in case it is ambiguous.
spring.cloud.function.definition		Definition of the function to be used. This could be function name (e.g., 'myFunction') or function composition definition (e.g., 'myFunction yourFunction')
spring.cloud.function.imports		Configuration for a set of files containing function bodies, which will be imported and compiled. The key in the map is the function name and the value is another map, containing a "location" of the file to compile and (optionally) a "type" (defaults to "function").

Name	Default	Description
spring.cloud.function.routing-expression		SpEL expression which should result in function definition (e.g., function name or composition instruction). NOTE: SpEL evaluation context's root object is the input argument (e.g., Message).
spring.cloud.function.web.exporter.auto-startup	true	Flag to indicate that the supplier emits HTTP requests automatically on startup.
spring.cloud.function.web.exporter.debug	true	Flag to indicate that extra logging is required for the supplier.
spring.cloud.function.web.exporter.enabled	false	Flag to enable the export of a supplier.
spring.cloud.function.web.exporter.sink.headers		Additional headers to append to the outgoing HTTP requests.
spring.cloud.function.web.exporter.sink.name		The name of a specific existing Supplier to export from the function catalog.
spring.cloud.function.web.exporter.sink.url		URL template for outgoing HTTP requests. Each item from the supplier is POSTed to this target.
spring.cloud.function.web.exporter.source.include-headers	true	Include the incoming headers in the outgoing Supplier. If true the supplier will be of generic type Message of T equal to the source type.
spring.cloud.function.web.exporter.source.type		If the origin url is set, the type of content expected (e.g. a POJO class). Defaults to String.
spring.cloud.function.web.exporter.source.url		URL template for creating a virtual Supplier from HTTP GET.
spring.cloud.function.web.path		Path to web resources for functions (should start with / if not empty).
spring.cloud.function.web.supplier.auto-startup	true	null

Name	Default	Description
spring.cloud.function.web.supplier.debug	true	null
spring.cloud.function.web.supplier.enabled	false	null
spring.cloud.function.web.supplier.headers		null
spring.cloud.function.web.supplier.name		null
spring.cloud.function.web.supplier.template-url		null
spring.cloud.gateway.default-filters		List of filter definitions that are applied to every route.
spring.cloud.gateway.discovery.locator.enabled	false	Flag that enables DiscoveryClient gateway integration.
spring.cloud.gateway.discovery.locator.filters		null
spring.cloud.gateway.discovery.locator.include-expression	true	SpEL expression that will evaluate whether to include a service in gateway integration or not, defaults to: true.
spring.cloud.gateway.discovery.locator.lower-case-service-id	false	Option to lower case serviceId in predicates and filters, defaults to false. Useful with eureka when it automatically uppercases serviceId. so MYSERIVCE, would match /myservice/**
spring.cloud.gateway.discovery.locator.predicates		null
spring.cloud.gateway.discovery.locator.route-id-prefix		The prefix for the routeId, defaults to discoveryClient.getClass().getSimpleName() + "_". Service Id will be appended to create the routeId.
spring.cloud.gateway.discovery.locator.url-expression	'lb://' + serviceId	SpEL expression that create the uri for each route, defaults to: 'lb://' + serviceId.
spring.cloud.gateway.enabled	true	Enables gateway functionality.

Name	Default	Description
spring.cloud.gateway.fail-on-route-definition-error	true	Option to fail on route definition errors, defaults to true. Otherwise, a warning is logged.
spring.cloud.gateway.filter.remove-hop-by-hop.headers		null
spring.cloud.gateway.filter.remove-hop-by-hop.order		null
spring.cloud.gateway.filter.request-rate-limiter.deny-empty-key	true	Switch to deny requests if the Key Resolver returns an empty key, defaults to true.
spring.cloud.gateway.filter.request-rate-limiter.empty-key-status-code		HttpStatus to return when denyEmptyKey is true, defaults to FORBIDDEN.
spring.cloud.gateway.filter.secure-headers.content-security-policy	default-src 'self' https;; font-src 'self' https: data;; img-src 'self' https: data;; object-src 'none'; script-src https;; style-src 'self' https: 'unsafe-inline'	null
spring.cloud.gateway.filter.secure-headers.content-type-options	nosniff	null
spring.cloud.gateway.filter.secure-headers.disable		null
spring.cloud.gateway.filter.secure-headers.download-options	noopen	null
spring.cloud.gateway.filter.secure-headers.frame-options	DENY	null
spring.cloud.gateway.filter.secure-headers.permitted-cross-domain-policies	none	null
spring.cloud.gateway.filter.secure-headers.referrer-policy	no-referrer	null
spring.cloud.gateway.filter.secure-headers.strict-transport-security	max-age=631138519	null
spring.cloud.gateway.filter.secure-headers.xss-protection-header	1 ; mode=block	null
spring.cloud.gateway.forwarded.enabled	true	Enables the ForwardedHeadersFilter.

Name	Default	Description
spring.cloud.gateway.globalcors. .add-to-simple-url-handler- mapping	false	If global CORS config should be added to the URL handler.
spring.cloud.gateway.globalcors. .cors-configurations		null
spring.cloud.gateway.httpclient. connect-timeout		The connect timeout in millis, the default is 45s.
spring.cloud.gateway.httpclient. max-header-size		The max response header size.
spring.cloud.gateway.httpclient. max-initial-line-length		The max initial line length.
spring.cloud.gateway.httpclient. pool.acquire-timeout		Only for type FIXED, the maximum time in millis to wait for acquiring.
spring.cloud.gateway.httpclient. pool.max-connections		Only for type FIXED, the maximum number of connections before starting pending acquisition on existing ones.
spring.cloud.gateway.httpclient. pool.max-idle-time		Time in millis after which the channel will be closed. If NULL, there is no max idle time.
spring.cloud.gateway.httpclient. pool.max-life-time		Duration after which the channel will be closed. If NULL, there is no max life time.
spring.cloud.gateway.httpclient. pool.name	proxy	The channel pool map name, defaults to proxy.
spring.cloud.gateway.httpclient. pool.type		Type of pool for HttpClient to use, defaults to ELASTIC.
spring.cloud.gateway.httpclient. proxy.host		Hostname for proxy configuration of Netty HttpClient.
spring.cloud.gateway.httpclient. proxy.non-proxy-hosts-pattern		Regular expression (Java) for a configured list of hosts. that should be reached directly, bypassing the proxy
spring.cloud.gateway.httpclient. proxy.password		Password for proxy configuration of Netty HttpClient.

Name	Default	Description
spring.cloud.gateway.httpclient.proxy.port		Port for proxy configuration of Netty HttpClient.
spring.cloud.gateway.httpclient.proxy.username		Username for proxy configuration of Netty HttpClient.
spring.cloud.gateway.httpclient.response-timeout		The response timeout.
spring.cloud.gateway.httpclient.ssl.close-notify-flush-timeout	3000ms	SSL close_notify flush timeout. Default to 3000 ms.
spring.cloud.gateway.httpclient.ssl.close-notify-read-timeout		SSL close_notify read timeout. Default to 0 ms.
spring.cloud.gateway.httpclient.ssl.default-configuration-type		The default ssl configuration type. Defaults to TCP.
spring.cloud.gateway.httpclient.ssl.handshake-timeout	10000ms	SSL handshake timeout. Default to 10000 ms
spring.cloud.gateway.httpclient.ssl.key-password		Key password, default is same as keyStorePassword.
spring.cloud.gateway.httpclient.ssl.key-store		Keystore path for Netty HttpClient.
spring.cloud.gateway.httpclient.ssl.key-store-password		Keystore password.
spring.cloud.gateway.httpclient.ssl.key-store-provider		Keystore provider for Netty HttpClient, optional field.
spring.cloud.gateway.httpclient.ssl.key-store-type	JKS	Keystore type for Netty HttpClient, default is JKS.
spring.cloud.gateway.httpclient.ssl.trusted-x509-certificates		Trusted certificates for verifying the remote endpoint's certificate.
spring.cloud.gateway.httpclient.ssl.use-insecure-trust-manager	false	Installs the netty InsecureTrustManagerFactory. This is insecure and not suitable for production.
spring.cloud.gateway.httpclient.websocket.max-frame-payload-length		Max frame payload length.
spring.cloud.gateway.httpclient.websocket.proxy-ping	true	Proxy ping frames to downstream services, defaults to true.
spring.cloud.gateway.httpclient.wiretap	false	Enables wiretap debugging for Netty HttpClient.

Name	Default	Description
spring.cloud.gateway.httpserver.wiretap	false	Enables wiretap debugging for Netty HttpServer.
spring.cloud.gateway.loadbalancer.use404	false	null
spring.cloud.gateway.metrics.enabled	true	Enables the collection of metrics data.
spring.cloud.gateway.metrics.tags		Tags map that added to metrics.
spring.cloud.gateway.proxy.auto-forward		A set of header names that should be send downstream by default.
spring.cloud.gateway.proxy.headers		Fixed header values that will be added to all downstream requests.
spring.cloud.gateway.proxy.sensitive		A set of sensitive header names that will not be sent downstream by default.
spring.cloud.gateway.redis-ratelimiter.burst-capacity-header	X-RateLimit-Burst-Capacity	The name of the header that returns the burst capacity configuration.
spring.cloud.gateway.redis-ratelimiter.config		null
spring.cloud.gateway.redis-ratelimiter.include-headers	true	Whether or not to include headers containing rate limiter information, defaults to true.
spring.cloud.gateway.redis-ratelimiter.remaining-header	X-RateLimit-Remaining	The name of the header that returns number of remaining requests during the current second.
spring.cloud.gateway.redis-ratelimiter.replenish-rate-header	X-RateLimit-Replenish-Rate	The name of the header that returns the replenish rate configuration.
spring.cloud.gateway.redis-ratelimiter.requested-tokens-header	X-RateLimit-Requested-Tokens	The name of the header that returns the requested tokens configuration.
spring.cloud.gateway.routes		List of Routes.
spring.cloud.gateway.set-status.original-status-header-name		The name of the header which contains http code of the proxied request.

Name	Default	Description
spring.cloud.gateway.streaming-media-types		null
spring.cloud.gateway.x-forwarded.enabled	true	If the XForwardedHeadersFilter is enabled.
spring.cloud.gateway.x-forwarded.for-append	true	If appending X-Forwarded-For as a list is enabled.
spring.cloud.gateway.x-forwarded.for-enabled	true	If X-Forwarded-For is enabled.
spring.cloud.gateway.x-forwarded.host-append	true	If appending X-Forwarded-Host as a list is enabled.
spring.cloud.gateway.x-forwarded.host-enabled	true	If X-Forwarded-Host is enabled.
spring.cloud.gateway.x-forwarded.order	0	The order of the XForwardedHeadersFilter.
spring.cloud.gateway.x-forwarded.port-append	true	If appending X-Forwarded-Port as a list is enabled.
spring.cloud.gateway.x-forwarded.port-enabled	true	If X-Forwarded-Port is enabled.
spring.cloud.gateway.x-forwarded.prefix-append	true	If appending X-Forwarded-Prefix as a list is enabled.
spring.cloud.gateway.x-forwarded.prefix-enabled	true	If X-Forwarded-Prefix is enabled.
spring.cloud.gateway.x-forwarded.proto-append	true	If appending X-Forwarded-Proto as a list is enabled.
spring.cloud.gateway.x-forwarded.proto-enabled	true	If X-Forwarded-Proto is enabled.
spring.cloud.httpclientfactories.apache.enabled	true	Enables creation of Apache Http Client factory beans.
spring.cloud.httpclientfactories.ok.enabled	true	Enables creation of OK Http Client factory beans.
spring.cloud.hypermedia.refresh.fixed-delay	5000	null
spring.cloud.hypermedia.refresh.initial-delay	10000	null
spring.cloud.inetutils.default-hostname	localhost	The default hostname. Used in case of errors.
spring.cloud.inetutils.default-ip-address	127.0.0.1	The default IP address. Used in case of errors.

Name	Default	Description
spring.cloud.inetutils.ignored-interfaces		List of Java regular expressions for network interfaces that will be ignored.
spring.cloud.inetutils.preferred-networks		List of Java regular expressions for network addresses that will be preferred.
spring.cloud.inetutils.timeout-seconds	1	Timeout, in seconds, for calculating hostname.
spring.cloud.inetutils.use-only-site-local-interfaces	false	Whether to use only interfaces with site local addresses. See {@link InetAddress#isSiteLocalAddress ()} for more details.
spring.cloud.kubernetes.client.api-version		null
spring.cloud.kubernetes.client.apiVersion	v1	Kubernetes API Version
spring.cloud.kubernetes.client.ca-cert-data		null
spring.cloud.kubernetes.client.ca-cert-file		null
spring.cloud.kubernetes.client.caCertData		Kubernetes API CACertData
spring.cloud.kubernetes.client.caCertFile		Kubernetes API CACertFile
spring.cloud.kubernetes.client.client-cert-data		null
spring.cloud.kubernetes.client.client-cert-file		null
spring.cloud.kubernetes.client.client-key-algo		null
spring.cloud.kubernetes.client.client-key-data		null
spring.cloud.kubernetes.client.client-key-file		null
spring.cloud.kubernetes.client.client-key-passphrase		null
spring.cloud.kubernetes.client.clientCertData		Kubernetes API ClientCertData

Name	Default	Description
spring.cloud.kubernetes.client.clientCertFile		Kubernetes API ClientCertFile
spring.cloud.kubernetes.client.clientKeyAlgo	RSA	Kubernetes API ClientKeyAlgo
spring.cloud.kubernetes.client.clientKeyData		Kubernetes API ClientKeyData
spring.cloud.kubernetes.client.clientKeyFile		Kubernetes API ClientKeyFile
spring.cloud.kubernetes.client.clientKeyPassphrase	changeit	Kubernetes API ClientKeyPassphrase
spring.cloud.kubernetes.client.connection-timeout		null
spring.cloud.kubernetes.client.connectionTimeout	10s	Connection timeout
spring.cloud.kubernetes.client.http-proxy		null
spring.cloud.kubernetes.client.https-proxy		null
spring.cloud.kubernetes.client.logging-interval		null
spring.cloud.kubernetes.client.loggingInterval	20s	Logging interval
spring.cloud.kubernetes.client.master-url		null
spring.cloud.kubernetes.client.masterUrl	kubernetes.default.svc	Kubernetes API Master Node URL
spring.cloud.kubernetes.client.namespace	true	Kubernetes Namespace
spring.cloud.kubernetes.client.no-proxy		null
spring.cloud.kubernetes.client.password		Kubernetes API Password
spring.cloud.kubernetes.client.proxy-password		null
spring.cloud.kubernetes.client.proxy-username		null
spring.cloud.kubernetes.client.request-timeout		null

Name	Default	Description
spring.cloud.kubernetes.client.requestTimeout	10s	Request timeout
spring.cloud.kubernetes.client.rolling-timeout		null
spring.cloud.kubernetes.client.rollingTimeout	900s	Rolling timeout
spring.cloud.kubernetes.client.trust-certs		null
spring.cloud.kubernetes.client.trustCerts	false	Kubernetes API Trust Certificates
spring.cloud.kubernetes.client.username		Kubernetes API Username
spring.cloud.kubernetes.client.watch-reconnect-interval		null
spring.cloud.kubernetes.client.watch-reconnect-limit		null
spring.cloud.kubernetes.client.watchReconnectInterval	1s	Reconnect Interval
spring.cloud.kubernetes.client.watchReconnectLimit	-1	Reconnect Interval limit retries
spring.cloud.kubernetes.config.enable-api	true	null
spring.cloud.kubernetes.config.enabled	true	Enable the ConfigMap property source locator.
spring.cloud.kubernetes.config.name		null
spring.cloud.kubernetes.config.namespace		null
spring.cloud.kubernetes.config.paths		null
spring.cloud.kubernetes.config.sources		null
spring.cloud.kubernetes.discovery.all-namespaces	false	If discovering all namespaces.
spring.cloud.kubernetes.discovery.enabled	true	If Kubernetes Discovery is enabled.

Name	Default	Description
spring.cloud.kubernetes.discovery.filter		SpEL expression to filter services AFTER they have been retrieved from the Kubernetes API server.
spring.cloud.kubernetes.discovery.known-secure-ports		Set the port numbers that are considered secure and use HTTPS.
spring.cloud.kubernetes.discovery.metadata.add-annotations	true	When set, the Kubernetes annotations of the services will be included as metadata of the returned ServiceInstance.
spring.cloud.kubernetes.discovery.metadata.add-labels	true	When set, the Kubernetes labels of the services will be included as metadata of the returned ServiceInstance.
spring.cloud.kubernetes.discovery.metadata.add-ports	true	When set, any named Kubernetes service ports will be included as metadata of the returned ServiceInstance.
spring.cloud.kubernetes.discovery.metadata.annotations-prefix		When addAnnotations is set, then this will be used as a prefix to the key names in the metadata map.
spring.cloud.kubernetes.discovery.metadata.labels-prefix		When addLabels is set, then this will be used as a prefix to the key names in the metadata map.
spring.cloud.kubernetes.discovery.metadata.ports-prefix	port.	When addPorts is set, then this will be used as a prefix to the key names in the metadata map.
spring.cloud.kubernetes.discovery.order		null
spring.cloud.kubernetes.discovery.primary-port-name		If set then the port with a given name is used as primary when multiple ports are defined for a service.
spring.cloud.kubernetes.discovery.service-labels		If set, then only the services matching these labels will be fetched from the Kubernetes API server.

Name	Default	Description
spring.cloud.kubernetes.discovery.service-name	unknown	The service name of the local instance.
spring.cloud.kubernetes.enabled	true	Whether to enable Kubernetes integration.
spring.cloud.kubernetes.reload.enabled	false	Enables the Kubernetes configuration reload on change.
spring.cloud.kubernetes.reload.max-wait-for-restart	2s	If Restart or Shutdown strategies are used, Spring Cloud Kubernetes waits a random amount of time before restarting. This is done in order to avoid having all instances of the same application restart at the same time. This property configures the maximum of amount of wait time from the moment the signal is received that a restart is needed until the moment the restart is actually triggered
spring.cloud.kubernetes.reload.mode		Sets the detection mode for Kubernetes configuration reload.
spring.cloud.kubernetes.reload.monitoring-config-maps	true	Enables monitoring on config maps to detect changes.
spring.cloud.kubernetes.reload.monitoring-secrets	false	Enables monitoring on secrets to detect changes.
spring.cloud.kubernetes.reload.period	15000ms	Sets the polling period to use when the detection mode is POLLING.
spring.cloud.kubernetes.reload.strategy		Sets the reload strategy for Kubernetes configuration reload on change.
spring.cloud.kubernetes.secrets.enable-api	false	null
spring.cloud.kubernetes.secrets.enabled	true	Enable the Secrets property source locator.
spring.cloud.kubernetes.secrets.labels		null
spring.cloud.kubernetes.secrets.name		null

Name	Default	Description
spring.cloud.kubernetes.secrets.namespace		null
spring.cloud.kubernetes.secrets.paths		null
spring.cloud.kubernetes.secrets.sources		null
spring.cloud.loadbalancer.cache.caffeine.spec		The spec to use to create caches. See CaffeineSpec for more details on the spec format.
spring.cloud.loadbalancer.cache.capacity	256	Initial cache capacity expressed as int.
spring.cloud.loadbalancer.cache.enabled	true	Enables Spring Cloud LoadBalancer caching mechanism.
spring.cloud.loadbalancer.cache.ttl	35s	Time To Live - time counted from writing of the record, after which cache entries are expired, expressed as a {@link Duration}. The property {@link String} has to be in keeping with the appropriate syntax as specified in Spring Boot <code>StringToDurationConverter</code> . @see https://github.com/spring-projects/spring-boot/blob/master/spring-boot-project/spring-boot/src/main/java/org/springframework/boot/convert/StringToDurationConverter.java
spring.cloud.loadbalancer.health-check.initial-delay		Initial delay value for the HealthCheck scheduler.
spring.cloud.loadbalancer.health-check.interval	25s	Interval for rerunning the HealthCheck scheduler.
spring.cloud.loadbalancer.health-check.path		null
spring.cloud.loadbalancer.retry.enabled	true	null

Name	Default	Description
spring.cloud.loadbalancer.ribbon.enabled	true	Causes <code>RibbonLoadBalancerClient</code> to be used by default.
spring.cloud.loadbalancer.service-discovery.timeout		String representation of Duration of the timeout for calls to service discovery.
spring.cloud.loadbalancer.zone		Spring Cloud LoadBalancer zone.
spring.cloud.refresh.enabled	true	Enables autoconfiguration for the refresh scope and associated features.
spring.cloud.refresh.extra-refreshable	true	Additional class names for beans to post process into refresh scope.
spring.cloud.refresh.never-refreshable	true	Comma separated list of class names for beans to never be refreshed or rebound.
spring.cloud.service-registry.auto-registration.enabled	true	Whether service auto-registration is enabled. Defaults to true.
spring.cloud.service-registry.auto-registration.fail-fast	false	Whether startup fails if there is no <code>AutoServiceRegistration</code> . Defaults to false.
spring.cloud.service-registry.auto-registration.register-management	true	Whether to register the management as a service. Defaults to true.
spring.cloud.service-registry.enabled	true	Enables Service Registry functionality.

Name	Default	Description
spring.cloud.stream.binders		Additional per-binder properties (see {@link BinderProperties}) if more than one binder of the same type is used (i.e., connect to multiple instances of RabbitMq). Here you can specify multiple binder configurations, each with different environment settings. For example; spring.cloud.stream.binders.rabbit1.environment. . . , spring.cloud.stream.binders.rabbit2.environment. . .
spring.cloud.stream.binding-retry-interval	30	Retry interval (in seconds) used to schedule binding attempts. Default: 30 sec.
spring.cloud.stream.bindings		Additional binding properties (see {@link BinderProperties}) per binding name (e.g., 'input `'). For example; This sets the content-type for the 'input' binding of a Sink application: 'spring.cloud.stream.bindings.input.contentType=text/plain'
spring.cloud.stream.consul.binder.event-timeout	5	null
spring.cloud.stream.default-binder		The name of the binder to use by all bindings in the event multiple binders available (e.g., 'rabbit').
spring.cloud.stream.dynamic-destinations	[]	A list of destinations that can be bound dynamically. If set, only listed destinations can be bound.
spring.cloud.stream.function.batch-mode	false	null
spring.cloud.stream.function.bindings		null
spring.cloud.stream.function.definition		Definition of functions to bind. If several functions need to be composed into one, use pipes (e.g., 'fooFunc barFunc')

Name	Default	Description
spring.cloud.stream.instance-count	1	The number of deployed instances of an application. Default: 1. NOTE: Could also be managed per individual binding "spring.cloud.stream.bindings.foo.consumer.instance-count" where 'foo' is the name of the binding.
spring.cloud.stream.instance-index	0	The instance id of the application: a number from 0 to instanceCount-1. Used for partitioning and with Kafka. NOTE: Could also be managed per individual binding "spring.cloud.stream.bindings.foo.consumer.instance-index" where 'foo' is the name of the binding.
spring.cloud.stream.instance-index-list		A list of instance id's from 0 to instanceCount-1. Used for partitioning and with Kafka. NOTE: Could also be managed per individual binding "spring.cloud.stream.bindings.foo.consumer.instance-index-list" where 'foo' is the name of the binding. This setting will override the one set in 'spring.cloud.stream.instance-index'
spring.cloud.stream.integration.message-handler-not-propagated-headers		Message header names that will NOT be copied from the inbound message.
spring.cloud.stream.kafka.binder.authorization-exception-retry-interval		Time between retries after AuthorizationException is caught in the ListenerContainer; default is null which disables retries. For more info see: {@link org.springframework.kafka.listener.ConsumerProperties#setAuthorizationExceptionRetryInterval(java.time.Duration)}

Name	Default	Description
spring.cloud.stream.kafka.binder.auto-add-partitions	false	null
spring.cloud.stream.kafka.binder.auto-create-topics	true	null
spring.cloud.stream.kafka.binder.brokers	[localhost]	null
spring.cloud.stream.kafka.binder.configuration		Arbitrary kafka properties that apply to both producers and consumers.
spring.cloud.stream.kafka.binder.consumer-properties		Arbitrary kafka consumer properties.
spring.cloud.stream.kafka.binder.header-mapper-bean-name		The bean name of a custom header mapper to use instead of a <code>@link org.springframework.kafka.support.DefaultKafkaHeaderMapper</code> .
spring.cloud.stream.kafka.binder.headers	[]	null
spring.cloud.stream.kafka.binder.health-timeout	60	Time to wait to get partition information in seconds; default 60.
spring.cloud.stream.kafka.binder.jaas		null
spring.cloud.stream.kafka.binder.min-partition-count	1	null
spring.cloud.stream.kafka.binder.producer-properties		Arbitrary kafka producer properties.
spring.cloud.stream.kafka.binder.replication-factor	1	null
spring.cloud.stream.kafka.binder.required-acks	1	null
spring.cloud.stream.kafka.binder.transaction.producer.batch-timeout		null
spring.cloud.stream.kafka.binder.transaction.producer.buffer-size		null

Name	Default	Description
spring.cloud.stream.kafka.binder.transaction.producer.compression-type		null
spring.cloud.stream.kafka.binder.transaction.producer.configuration		null
spring.cloud.stream.kafka.binder.transaction.producer.error-channel-enabled		null
spring.cloud.stream.kafka.binder.transaction.producer.header-mode		null
spring.cloud.stream.kafka.binder.transaction.producer.header-patterns		null
spring.cloud.stream.kafka.binder.transaction.producer.message-key-expression		null
spring.cloud.stream.kafka.binder.transaction.producer.partition-count		null
spring.cloud.stream.kafka.binder.transaction.producer.partition-key-expression		null
spring.cloud.stream.kafka.binder.transaction.producer.partition-key-extractor-name		null
spring.cloud.stream.kafka.binder.transaction.producer.partition-selector-expression		null
spring.cloud.stream.kafka.binder.transaction.producer.partition-selector-name		null
spring.cloud.stream.kafka.binder.transaction.producer.required-groups		null
spring.cloud.stream.kafka.binder.transaction.producer.sync		null
spring.cloud.stream.kafka.binder.transaction.producer.topic		null

Name	Default	Description
spring.cloud.stream.kafka.binder.transaction.producer.use-native-encoding		null
spring.cloud.stream.kafka.binder.transaction.transaction-id-prefix		null
spring.cloud.stream.kafka.bindings		null
spring.cloud.stream.kafka.streams.binder.application-id		null
spring.cloud.stream.kafka.streams.binder.authorization-exception-retry-interval		null
spring.cloud.stream.kafka.streams.binder.auto-add-partitions		null
spring.cloud.stream.kafka.streams.binder.auto-create-topics		null
spring.cloud.stream.kafka.streams.binder.brokers		null
spring.cloud.stream.kafka.streams.binder.configuration		null
spring.cloud.stream.kafka.streams.binder.consumer-properties		null
spring.cloud.stream.kafka.streams.binder.deserialization-exception-handler		{@link org.apache.kafka.streams.errors.DeserializationExceptionHandler} to use when there is a deserialization exception. This handler will be applied against all input bindings unless overridden at the consumer binding.
spring.cloud.stream.kafka.streams.binder.functions		null
spring.cloud.stream.kafka.streams.binder.header-mapper-bean-name		null
spring.cloud.stream.kafka.streams.binder.headers		null

Name	Default	Description
spring.cloud.stream.kafka.streams.binder.health-timeout		null
spring.cloud.stream.kafka.streams.binder.jaas		null
spring.cloud.stream.kafka.streams.binder.min-partition-count		null
spring.cloud.stream.kafka.streams.binder.producer-properties		null
spring.cloud.stream.kafka.streams.binder.replication-factor		null
spring.cloud.stream.kafka.streams.binder.required-acks		null
spring.cloud.stream.kafka.streams.binder.serde-error		null
spring.cloud.stream.kafka.streams.binder.state-store-retry.backoff-period	1000	null
spring.cloud.stream.kafka.streams.binder.state-store-retry.max-attempts	1	null
spring.cloud.stream.kafka.streams.binder.transaction.producer.batch-timeout		null
spring.cloud.stream.kafka.streams.binder.transaction.producer.buffer-size		null
spring.cloud.stream.kafka.streams.binder.transaction.producer.compression-type		null
spring.cloud.stream.kafka.streams.binder.transaction.producer.configuration		null
spring.cloud.stream.kafka.streams.binder.transaction.producer.error-channel-enabled		null
spring.cloud.stream.kafka.streams.binder.transaction.producer.header-mode		null

Name	Default	Description
spring.cloud.stream.kafka.streams.binder.transaction.producer.header-patterns		null
spring.cloud.stream.kafka.streams.binder.transaction.producer.message-key-expression		null
spring.cloud.stream.kafka.streams.binder.transaction.producer.partition-count		null
spring.cloud.stream.kafka.streams.binder.transaction.producer.partition-key-expression		null
spring.cloud.stream.kafka.streams.binder.transaction.producer.partition-key-extractor-name		null
spring.cloud.stream.kafka.streams.binder.transaction.producer.partition-selector-expression		null
spring.cloud.stream.kafka.streams.binder.transaction.producer.partition-selector-name		null
spring.cloud.stream.kafka.streams.binder.transaction.producer.required-groups		null
spring.cloud.stream.kafka.streams.binder.transaction.producer.sync		null
spring.cloud.stream.kafka.streams.binder.transaction.producer.topic		null
spring.cloud.stream.kafka.streams.binder.transaction.producer.use-native-encoding		null
spring.cloud.stream.kafka.transaction.transaction-id-prefix		null
spring.cloud.stream.kafka.streams.bindings		null

Name	Default	Description
spring.cloud.stream.metrics.export-properties		List of properties that are going to be appended to each message. This gets populated by onApplicationEvent, once the context refreshes to avoid overhead of doing per message basis.
spring.cloud.stream.metrics.key		The name of the metric being emitted. Should be a unique value per application. Defaults to: <code>\${spring.application.name:\${vcap.application.name:\${spring.config.name:application}}}</code> .
spring.cloud.stream.metrics.meter-filter		Pattern to control the 'meters' one wants to capture. By default all 'meters' will be captured. For example, 'spring.integration.*' will only capture metric information for meters whose name starts with 'spring.integration'.
spring.cloud.stream.metrics.properties		Application properties that should be added to the metrics payload For example: <code>spring.application**</code> .
spring.cloud.stream.metrics.schedule-interval	60s	Interval expressed as Duration for scheduling metrics snapshots publishing. Defaults to 60 seconds

Name	Default	Description
spring.cloud.stream.override-cloud-connectors	false	This property is only applicable when the cloud profile is active and Spring Cloud Connectors are provided with the application. If the property is false (the default), the binder detects a suitable bound service (for example, a RabbitMQ service bound in Cloud Foundry for the RabbitMQ binder) and uses it for creating connections (usually through Spring Cloud Connectors). When set to true, this property instructs binders to completely ignore the bound services and rely on Spring Boot properties (for example, relying on the spring.rabbitmq.* properties provided in the environment for the RabbitMQ binder). The typical usage of this property is to be nested in a customized environment when connecting to multiple systems.
spring.cloud.stream.poller.cron		Cron expression value for the Cron Trigger.
spring.cloud.stream.poller.fixed-delay	1000	Fixed delay for default poller.
spring.cloud.stream.poller.initial-delay	0	Initial delay for periodic triggers.
spring.cloud.stream.poller.max-messages-per-poll	1	Maximum messages per poll for the default poller.
spring.cloud.stream.rabbit.binder.admin-addresses	[]	Urls for management plugins; only needed for queue affinity.
spring.cloud.stream.rabbit.binder.admin-adresses		null
spring.cloud.stream.rabbit.binder.compression-level	0	Compression level for compressed bindings; see 'java.util.zip.Deflator'.
spring.cloud.stream.rabbit.binder.connection-name-prefix		Prefix for connection names from this binder.

Name	Default	Description
spring.cloud.stream.rabbit.binder.nodes	[]	Cluster member node names; only needed for queue affinity.
spring.cloud.stream.rabbit.bindings		null
spring.cloud.stream.sendto.destination	none	The name of the header used to determine the name of the output destination
spring.cloud.stream.source		A colon delimited string representing the names of the sources based on which source bindings will be created. This is primarily to support cases where source binding may be required without providing a corresponding Supplier. (e.g., for cases where the actual source of data is outside of scope of spring-cloud-stream - HTTP → Stream)
spring.cloud.util.enabled	true	Enables creation of Spring Cloud utility beans.
spring.cloud.vault.app-id.app-id-path	app-id	Mount path of the AppId authentication backend.
spring.cloud.vault.app-id.network-interface		Network interface hint for the "MAC_ADDRESS" UserId mechanism.
spring.cloud.vault.app-id.user-id	MAC_ADDRESS	UserId mechanism. Can be either "MAC_ADDRESS", "IP_ADDRESS", a string or a class name.
spring.cloud.vault.app-role.app-role-path	approle	Mount path of the AppRole authentication backend.
spring.cloud.vault.app-role.role		Name of the role, optional, used for pull-mode.
spring.cloud.vault.app-role.role-id		The RoleId.
spring.cloud.vault.app-role.secret-id		The SecretId.
spring.cloud.vault.application-name	application	Application name for AppId authentication.

Name	Default	Description
spring.cloud.vault.authentication		null
spring.cloud.vault.aws-ec2.aws-ec2-path	aws-ec2	Mount path of the AWS-EC2 authentication backend.
spring.cloud.vault.aws-ec2.identity-document	169.254.169.254/latest/dynamic/instance-identity/pkcs7	URL of the AWS-EC2 PKCS7 identity document.
spring.cloud.vault.aws-ec2.nonce		Nonce used for AWS-EC2 authentication. An empty nonce defaults to nonce generation.
spring.cloud.vault.aws-ec2.role		Name of the role, optional.
spring.cloud.vault.aws-iam.aws-path	aws	Mount path of the AWS authentication backend.
spring.cloud.vault.aws-iam.endpoint-uri		STS server URI. @since 2.2
spring.cloud.vault.aws-iam.role		Name of the role, optional. Defaults to the friendly IAM name if not set.
spring.cloud.vault.aws-iam.server-name		Name of the server used to set {@code X-Vault-AWS-IAM-Server-ID} header in the headers of login requests.
spring.cloud.vault.aws.access-key-property	cloud.aws.credentials.accessKey	Target property for the obtained access key.
spring.cloud.vault.aws.backend	aws	aws backend path.
spring.cloud.vault.aws.enabled	false	Enable aws backend usage.
spring.cloud.vault.aws.role		Role name for credentials.
spring.cloud.vault.aws.secret-key-property	cloud.aws.credentials.secretKey	Target property for the obtained secret key.
spring.cloud.vault.azure-msi.azure-path	azure	Mount path of the Azure MSI authentication backend.
spring.cloud.vault.azure-msi.role		Name of the role.
spring.cloud.vault.cassandra.backend	cassandra	Cassandra backend path.
spring.cloud.vault.cassandra.enabled	false	Enable cassandra backend usage.
spring.cloud.vault.cassandra.password-property	spring.data.cassandra.password	Target property for the obtained password.

Name	Default	Description
spring.cloud.vault.cassandra.role		Role name for credentials.
spring.cloud.vault.cassandra.static-role	false	Enable static role usage. @since 2.2
spring.cloud.vault.cassandra.username-property	spring.data.cassandra.username	Target property for the obtained username.
spring.cloud.vault.config.lifecycle.enabled	true	Enable lifecycle management.
spring.cloud.vault.config.lifecycle.expiry-threshold		The expiry threshold. {@link Lease} is renewed the given {@link Duration} before it expires. @since 2.2
spring.cloud.vault.config.lifecycle.lease-endpoints		Set the {@link LeaseEndpoints} to delegate renewal/revocation calls to. {@link LeaseEndpoints} encapsulates differences between Vault versions that affect the location of renewal/revocation endpoints. Can be {@link LeaseEndpoints#SysLeases} for version 0.8 or above of Vault or {@link LeaseEndpoints#Legacy} for older versions (the default). @since 2.2
spring.cloud.vault.config.lifecycle.min-renewal		The time period that is at least required before renewing a lease. @since 2.2
spring.cloud.vault.config.order	0	Used to set a {@link org.springframework.core.env.PropertySource} priority. This is useful to use Vault as an override on other property sources. @see org.springframework.core.PriorityOrdered
spring.cloud.vault.connection-timeout	5000	Connection timeout.
spring.cloud.vault.consul.backend	consul	Consul backend path.
spring.cloud.vault.consul.enabled	false	Enable consul backend usage.

Name	Default	Description
spring.cloud.vault.consul.role		Role name for credentials.
spring.cloud.vault.consul.token-property	spring.cloud.consul.token	Target property for the obtained token.
spring.cloud.vault.database.backend	database	Database backend path.
spring.cloud.vault.database.enabled	false	Enable database backend usage.
spring.cloud.vault.database.password-property	spring.datasource.password	Target property for the obtained password.
spring.cloud.vault.database.role		Role name for credentials.
spring.cloud.vault.database.static-role	false	Enable static role usage.
spring.cloud.vault.database.username-property	spring.datasource.username	Target property for the obtained username.
spring.cloud.vault.discovery.enabled	false	Flag to indicate that Vault server discovery is enabled (vault server URL will be looked up via discovery).
spring.cloud.vault.discovery.service-id	vault	Service id to locate Vault.
spring.cloud.vault.elasticsearch.backend	database	Database backend path.
spring.cloud.vault.elasticsearch.enabled	false	Enable elasticsearch backend usage.
spring.cloud.vault.elasticsearch.password-property	spring.elasticsearch.rest.password	Target property for the obtained password.
spring.cloud.vault.elasticsearch.role		Role name for credentials.
spring.cloud.vault.elasticsearch.static-role	false	Enable static role usage.
spring.cloud.vault.elasticsearch.username-property	spring.elasticsearch.rest.username	Target property for the obtained username.
spring.cloud.vault.enabled	true	Enable Vault config server.
spring.cloud.vault.fail-fast	false	Fail fast if data cannot be obtained from Vault.
spring.cloud.vault.gcp-gce.gcp-path	gcp	Mount path of the Kubernetes authentication backend.

Name	Default	Description
spring.cloud.vault.gcp-gce.role		Name of the role against which the login is being attempted.
spring.cloud.vault.gcp-gce.service-account		Optional service account id. Using the default id if left unconfigured.
spring.cloud.vault.gcp-iam.credentials.encoded-key		The base64 encoded contents of an OAuth2 account private key in JSON format.
spring.cloud.vault.gcp-iam.credentials.location		Location of the OAuth2 credentials private key. <p>Since this is a Resource, the private key can be in a multitude of locations, such as a local file system, classpath, URL, etc.
spring.cloud.vault.gcp-iam.gcp-path	gcp	Mount path of the Kubernetes authentication backend.
spring.cloud.vault.gcp-iam.jwt-validity	15m	Validity of the JWT token.
spring.cloud.vault.gcp-iam.project-id		Overrides the GCP project Id.
spring.cloud.vault.gcp-iam.role		Name of the role against which the login is being attempted.
spring.cloud.vault.gcp-iam.service-account-id		Overrides the GCP service account Id.
spring.cloud.vault.host	localhost	Vault server host.
spring.cloud.vault.kubernetes.kubernetes-path	kubernetes	Mount path of the Kubernetes authentication backend.
spring.cloud.vault.kubernetes.role		Name of the role against which the login is being attempted.
spring.cloud.vault.kubernetes.service-account-token-file	/var/run/secrets/kubernetes.io/serviceaccount/token	Path to the service account token file.
spring.cloud.vault.kv.application-name	application	Application name to be used for the context.
spring.cloud.vault.kv.backend	secret	Name of the default backend.

Name	Default	Description
spring.cloud.vault.kv.backend-version	2	Key-Value backend version. Currently supported versions are: <ul style="list-style-type: none"> Version 1 (unversioned key-value backend). Version 2 (versioned key-value backend).
spring.cloud.vault.kv.default-context	application	Name of the default context.
spring.cloud.vault.kv.enabled	true	Enable the key-value backend.
spring.cloud.vault.kv.profile-separator	/	Profile-separator to combine application name and profile.
spring.cloud.vault.kv.profiles		List of active profiles. @since 3.0
spring.cloud.vault.mongodb.backend	mongodb	MongoDB backend path.
spring.cloud.vault.mongodb.enabled	false	Enable mongodb backend usage.
spring.cloud.vault.mongodb.password-property	spring.data.mongodb.password	Target property for the obtained password.
spring.cloud.vault.mongodb.role		Role name for credentials.
spring.cloud.vault.mongodb.static-role	false	Enable static role usage. @since 2.2
spring.cloud.vault.mongodb.username-property	spring.data.mongodb.username	Target property for the obtained username.
spring.cloud.vault.mysql.backend	mysql	mysql backend path.
spring.cloud.vault.mysql.enabled	false	Enable mysql backend usage.
spring.cloud.vault.mysql.password-property	spring.datasource.password	Target property for the obtained username.
spring.cloud.vault.mysql.role		Role name for credentials.
spring.cloud.vault.mysql.username-property	spring.datasource.username	Target property for the obtained username.
spring.cloud.vault.namespace		Vault namespace (requires Vault Enterprise).

Name	Default	Description
spring.cloud.vault.pcf.instance-certificate		Path to the instance certificate (PEM). Defaults to {@code CF_INSTANCE_CERT} env variable.
spring.cloud.vault.pcf.instance-key		Path to the instance key (PEM). Defaults to {@code CF_INSTANCE_KEY} env variable.
spring.cloud.vault.pcf.pcf-path	pcf	Mount path of the Kubernetes authentication backend.
spring.cloud.vault.pcf.role		Name of the role against which the login is being attempted.
spring.cloud.vault.port	8200	Vault server port.
spring.cloud.vault.postgresql.backend	postgresql	postgresql backend path.
spring.cloud.vault.postgresql.enabled	false	Enable postgresql backend usage.
spring.cloud.vault.postgresql.password-property	spring.datasource.password	Target property for the obtained username.
spring.cloud.vault.postgresql.role		Role name for credentials.
spring.cloud.vault.postgresql.username-property	spring.datasource.username	Target property for the obtained username.
spring.cloud.vault.rabbitmq.backend	rabbitmq	rabbitmq backend path.
spring.cloud.vault.rabbitmq.enabled	false	Enable rabbitmq backend usage.
spring.cloud.vault.rabbitmq.password-property	spring.rabbitmq.password	Target property for the obtained password.
spring.cloud.vault.rabbitmq.role		Role name for credentials.
spring.cloud.vault.rabbitmq.username-property	spring.rabbitmq.username	Target property for the obtained username.
spring.cloud.vault.read-timeout	15000	Read timeout.
spring.cloud.vault.scheme	https	Protocol scheme. Can be either "http" or "https".
spring.cloud.vault.ssl.cert-auth-path	cert	Mount path of the TLS cert authentication backend.

Name	Default	Description
spring.cloud.vault.ssl.key-store		Trust store that holds certificates and private keys.
spring.cloud.vault.ssl.key-store-password		Password used to access the key store.
spring.cloud.vault.ssl.trust-store		Trust store that holds SSL certificates.
spring.cloud.vault.ssl.trust-store-password		Password used to access the trust store.
spring.cloud.vault.token		Static vault token. Required if {@link #authentication} is {@code TOKEN}.
spring.cloud.vault.uri		Vault URI. Can be set with scheme, host and port.
spring.cloud.zookeeper.base-sleep-time-ms	50	Initial amount of time to wait between retries.
spring.cloud.zookeeper.block-until-connected-unit		The unit of time related to blocking on connection to Zookeeper.
spring.cloud.zookeeper.block-until-connected-wait	10	Wait time to block on connection to Zookeeper.
spring.cloud.zookeeper.connect-string	localhost:2181	Connection string to the Zookeeper cluster.
spring.cloud.zookeeper.connection-timeout		The configured connection timeout in milliseconds.
spring.cloud.zookeeper.dependencies		Mapping of alias to ZookeeperDependency. From LoadBalancer perspective the alias is actually serviceID since SC LoadBalancer can't accept nested structures in serviceID.
spring.cloud.zookeeper.dependency-configurations		null
spring.cloud.zookeeper.dependency-names		null
spring.cloud.zookeeper.discovery.enabled	true	null
spring.cloud.zookeeper.discovery.initial-status		The initial status of this instance (defaults to {@link StatusConstants#STATUS_UP}).

Name	Default	Description
spring.cloud.zookeeper.discovery.instance-host		Predefined host with which a service can register itself in Zookeeper. Corresponds to the {code address} from the URI spec.
spring.cloud.zookeeper.discovery.instance-id		Id used to register with zookeeper. Defaults to a random UUID.
spring.cloud.zookeeper.discovery.instance-port		Port to register the service under (defaults to listening port).
spring.cloud.zookeeper.discovery.instance-ssl-port		Ssl port of the registered service.
spring.cloud.zookeeper.discovery.metadata		Gets the metadata name/value pairs associated with this instance. This information is sent to zookeeper and can be used by other instances.
spring.cloud.zookeeper.discovery.order	0	Order of the discovery client used by <code>CompositeDiscoveryClient</code> for sorting available clients.
spring.cloud.zookeeper.discovery.register	true	Register as a service in zookeeper.
spring.cloud.zookeeper.discovery.root	/services	Root Zookeeper folder in which all instances are registered.
spring.cloud.zookeeper.discovery.uri-spec	{scheme}://{address}:{port}	The URI specification to resolve during service registration in Zookeeper.
spring.cloud.zookeeper.enabled	true	Is Zookeeper enabled.
spring.cloud.zookeeper.max-retries	10	Max number of times to retry.
spring.cloud.zookeeper.max-sleep-ms	500	Max time in ms to sleep on each retry.
spring.cloud.zookeeper.prefix		Common prefix that will be applied to all Zookeeper dependencies' paths.

Name	Default	Description
spring.cloud.zookeeper.session-timeout		The configured/negotiated session timeout in milliseconds. Please refer to Curator's Tech Note 14 to understand how Curator implements connection sessions. @see Curator's Tech Note 14
spring.sleuth.async.configurer.enabled	true	Enable default AsyncConfigurer.
spring.sleuth.async.enabled	true	Enable instrumenting async related components so that the tracing information is passed between threads.
spring.sleuth.async.ignored-beans		List of java.util.concurrent.Executor bean names that should be ignored and not wrapped in a trace representation.
spring.sleuth.baggage.correlation-enabled	true	Adds a CorrelationScopeDecorator to put baggage values into the correlation context.
spring.sleuth.baggage.correlation-fields		A list of BaggageField#name() fields to add to correlation (MDC) context. @see CorrelationScopeConfig.SingleCorrelationField#create(BaggageField)
spring.sleuth.baggage.local-fields		Same as #remoteFields except that this field is not propagated to remote services. @see BaggagePropagationConfig.SingleBaggageField#local(BaggageField)

Name	Default	Description
spring.sleuth.baggage.remote-fields		List of fields that are referenced the same in-process as it is on the wire. For example, the field "x-vcap-request-id" would be set as-is including the prefix. @see BaggagePropagationConfig.SingleBaggageField#remote(BaggageField) @see BaggagePropagationConfig.SingleBaggageField.Builder#addKeyName(String)
spring.sleuth.baggage.tag-fields		A list of {@link BaggageField#name() fields} to tag into the span. @see Tags#BAGGAGE_FIELD
spring.sleuth.circuitbreaker.enabled	true	Enable Spring Cloud CircuitBreaker instrumentation.
spring.sleuth.enabled	true	null
spring.sleuth.feign.enabled	true	Enable span information propagation when using Feign.
spring.sleuth.feign.processor.enabled	true	Enable post processor that wraps Feign Context in its tracing representations.
spring.sleuth.grpc.enabled	true	Enable span information propagation when using GRPC.
spring.sleuth.http.enabled	true	null
spring.sleuth.http.legacy.enabled	false	null
spring.sleuth.integration.enabled	true	Enable Spring Integration sleuth instrumentation.
spring.sleuth.integration.patterns	[!hystrixStreamOutput*, , !channel]	An array of patterns against which channel names will be matched. @see org.springframework.integration.config.GlobalChannelInterceptor#patterns() Defaults to any channel name not matching the Hystrix Stream and functional Stream channel names.
spring.sleuth.integration.websockets.enabled	true	Enable tracing for WebSockets.

Name	Default	Description
spring.sleuth.messaging.enabled	false	Should messaging be turned on.
spring.sleuth.messaging.jms.enabled	true	Enable tracing of JMS.
spring.sleuth.messaging.jms.remote-service-name	jms	null
spring.sleuth.messaging.kafka.enabled	true	Enable tracing of Kafka.
spring.sleuth.messaging.kafka.mapper.enabled	true	Enable DefaultKafkaHeaderMapper tracing for Kafka.
spring.sleuth.messaging.kafka.remote-service-name	kafka	null
spring.sleuth.messaging.rabbit.enabled	true	Enable tracing of RabbitMQ.
spring.sleuth.messaging.rabbit.remote-service-name	rabbitmq	null
spring.sleuth.opentracing.enabled	true	null
spring.sleuth.reactor.decorate-on-each	true	When true decorates on each operator, will be less performing, but logging will always contain the tracing entries in each operator. When false decorates on last operator, will be more performing, but logging might not always contain the tracing entries.
spring.sleuth.reactor.enabled	true	When true enables instrumentation for reactor.
spring.sleuth.redis.enabled	true	Enable span information propagation when using Redis.
spring.sleuth.redis.remote-service-name	redis	Service name for the remote Redis endpoint.
spring.sleuth.rpc.enabled	true	Enable tracing of RPC.
spring.sleuth.rxjava.schedulers.hook.enabled	true	Enable support for RxJava via RxJavaSchedulersHook.
spring.sleuth.rxjava.schedulers.ignoredthreads	[HystrixMetricPoller, ^RxComputation.*\$]	Thread names for which spans will not be sampled.

Name	Default	Description
spring.sleuth.sampler.probability		Probability of requests that should be sampled. E.g. 1.0 - 100% requests should be sampled. The precision is whole-numbers only (i.e. there's no support for 0.1% of the traces).
spring.sleuth.sampler.rate	10	A rate per second can be a nice choice for low-traffic endpoints as it allows you surge protection. For example, you may never expect the endpoint to get more than 50 requests per second. If there was a sudden surge of traffic, to 5000 requests per second, you would still end up with 50 traces per second. Conversely, if you had a percentage, like 10%, the same surge would end up with 500 traces per second, possibly overloading your storage. Amazon X-Ray includes a rate-limited sampler (named Reservoir) for this purpose. Brave has taken the same approach via the <code>{@link brave.sampler.RateLimitingSampler}</code> .
spring.sleuth.scheduled.enabled	true	Enable tracing for <code>{@link org.springframework.scheduling.annotation.Scheduled}</code> .
spring.sleuth.scheduled.skip-pattern		Pattern for the fully qualified name of a class that should be skipped.
spring.sleuth.supports-join	true	True means the tracing system supports sharing a span ID between a client and server.
spring.sleuth.trace-id128	false	When true, generate 128-bit trace IDs instead of 64-bit ones.

Name	Default	Description
spring.sleuth.web.additional-skip-pattern		Additional pattern for URLs that should be skipped in tracing. This will be appended to the {@link SleuthWebProperties#skipPattern}.
spring.sleuth.web.client.enabled	true	Enable interceptor injecting into {@link org.springframework.web.client.RestTemplate}.
spring.sleuth.web.client.skip-pattern		Pattern for URLs that should be skipped in client side tracing.
spring.sleuth.web.enabled	true	When true enables instrumentation for web applications.
spring.sleuth.web.filter-order		Order in which the tracing filters should be registered. Defaults to {@link TraceHttpAutoConfiguration#TRACING_FILTER_ORDER}.
spring.sleuth.web.ignore-auto-configured-skip-patterns	false	If set to true, auto-configured skip patterns will be ignored. @see SkipPatternConfiguration
spring.sleuth.web.skip-pattern	/api-docs. /swagger. \.png \.css \.js \.html /favicon.ico /hystrix.stream	Pattern for URLs that should be skipped in tracing.
spring.zipkin.activemq.message-max-bytes	100000	Maximum number of bytes for a given message with spans sent to Zipkin over ActiveMQ.
spring.zipkin.activemq.queue	zipkin	Name of the ActiveMQ queue where spans should be sent to Zipkin.
spring.zipkin.base-url	localhost:9411/	URL of the zipkin query server instance. You can also provide the service id of the Zipkin server if Zipkin's registered in service discovery (e.g. zipkinserver/).
spring.zipkin.compression.enabled	false	null

Name	Default	Description
spring.zipkin.discovery-client-enabled		If set to {@code false}, will treat the {@link ZipkinProperties#baseUrl} as a URL always.
spring.zipkin.enabled	true	Enables sending spans to Zipkin.
spring.zipkin.encoder		Encoding type of spans sent to Zipkin. Set to {@link SpanBytesEncoder#JSON_V1} if your server is not recent.
spring.zipkin.kafka.topic	zipkin	Name of the Kafka topic where spans should be sent to Zipkin.
spring.zipkin.locator.discovery.enabled	false	Enabling of locating the host name via service discovery.
spring.zipkin.message-timeout	1	Timeout in seconds before pending spans will be sent in batches to Zipkin.
spring.zipkin.rabbitmq.addresses		Addresses of the RabbitMQ brokers used to send spans to Zipkin
spring.zipkin.rabbitmq.queue	zipkin	Name of the RabbitMQ queue where spans should be sent to Zipkin.
spring.zipkin.sender.type		Means of sending spans to Zipkin.
spring.zipkin.service.name		The name of the service, from which the Span was sent via HTTP, that should appear in Zipkin.
stubrunner.amqp.enabled	false	Whether to enable support for Stub Runner and AMQP.
stubrunner.amqp.mockConnection	true	Whether to enable support for Stub Runner and AMQP mocked connection factory.
stubrunner.classifier	stubs	The classifier to use by default in ivy co-ordinates for a stub.
stubrunner.cloud.consul.enabled	true	Whether to enable stubs registration in Consul.

Name	Default	Description
stubrunner.cloud.delegate.enabled	true	Whether to enable DiscoveryClient's Stub Runner implementation.
stubrunner.cloud.enabled	true	Whether to enable Spring Cloud support for Stub Runner.
stubrunner.cloud.eureka.enabled	true	Whether to enable stubs registration in Eureka.
stubrunner.cloud.loadbalancer.enabled	true	Whether to enable Stub Runner's Spring Cloud Load Balancer integration.
stubrunner.cloud.stubbed.discovery.enabled	true	Whether Service Discovery should be stubbed for Stub Runner. If set to false, stubs will get registered in real service discovery.
stubrunner.cloud.zookeeper.enabled	true	Whether to enable stubs registration in Zookeeper.
stubrunner.consumer-name		You can override the default <code>spring.application.name</code> of this field by setting a value to this parameter.
stubrunner.delete-stubs-after-test	true	If set to <code>false</code> will NOT delete stubs from a temporary folder after running tests.
stubrunner.fail-on-no-stubs	true	When enabled, this flag will tell stub runner to throw an exception when no stubs / contracts were found.
stubrunner.generate-stubs	false	When enabled, this flag will tell stub runner to not load the generated stubs, but convert the found contracts at runtime to a stub format and run those stubs.
stubrunner.http-server-stub-configurer		Configuration for an HTTP server stub.

Name	Default	Description
stubrunner.ids	[]	The ids of the stubs to run in "ivy" notation ([groupId]:artifactId:[version]:[classifier][:port]). {@code groupId}, {@code classifier}, {@code version} and {@code port} can be optional.
stubrunner.ids-to-service-ids		Mapping of Ivy notation based ids to serviceIds inside your application. Example "a:b" → "myService" "artifactId" → "myOtherService"
stubrunner.integration.enabled	true	Whether to enable Stub Runner integration with Spring Integration.
stubrunner.jms.enabled	true	Whether to enable Stub Runner integration with Spring JMS.
stubrunner.kafka.enabled	true	Whether to enable Stub Runner integration with Spring Kafka.
stubrunner.kafka.initializer.enabled	true	Whether to allow Stub Runner to take care of polling for messages instead of the KafkaStubMessages component. The latter should be used only on the producer side.
stubrunner.mappings-output-folder		Dumps the mappings of each HTTP server to the selected folder.
stubrunner.max-port	15000	Max value of a port for the automatically started WireMock server.
stubrunner.min-port	10000	Min value of a port for the automatically started WireMock server.
stubrunner.password		Repository password.
stubrunner.properties		Map of properties that can be passed to custom {@link org.springframework.cloud.contract.stubrunner.StubDownloaderBuilder}.
stubrunner.proxy-host		Repository proxy host.

Name	Default	Description
stubrunner.proxy-port		Repository proxy port.
stubrunner.server-id		null
stubrunner.stream.enabled	true	Whether to enable Stub Runner integration with Spring Cloud Stream.
stubrunner.stubs-mode		Pick where the stubs should come from.
stubrunner.stubs-per-consumer	false	Should only stubs for this particular consumer get registered in HTTP server stub.
stubrunner.username		Repository username.
wiremock.placeholders.enabled	true	Flag to indicate that http URLs in generated wiremock stubs should be filtered to add or resolve a placeholder for a dynamic port.
wiremock.reset-mappings-after-each-test	false	null
wiremock.rest-template-ssl-enabled	false	null
wiremock.server.files	[]	null
wiremock.server.https-port	-1	null
wiremock.server.https-port-dynamic	false	null
wiremock.server.port	8080	null
wiremock.server.port-dynamic	false	null
wiremock.server.stubs	[]	null