# Spring Cloud

## Table of Contents

1. Features .................................................................................. 11
2. Release Train Versions ......................................................... 11
   Spring Cloud Build ................................................................. 12
      1. Building and Deploying .................................................. 12
      2. Contributing .................................................................. 13
         2.1. Sign the Contributor License Agreement ................... 13
         2.2. Code of Conduct .................................................... 13
         2.3. Code Conventions and Housekeeping ...................... 13
         2.4. Checkstyle ............................................................. 14
         2.5. IDE setup .............................................................. 16
      3. Flattening the POMs ......................................................... 19
      4. Reusing the documentation ............................................. 20
      5. Updating the guides ....................................................... 23
   Spring Cloud Bus .................................................................... 24
      1. Quick Start .................................................................... 24
      2. Bus Endpoints ................................................................ 25
         2.1. Bus Refresh Endpoint .................. 25
         2.2. Bus Env Endpoint ................................................ 25
      3. Addressing an Instance .................................................. 26
      4. Addressing All Instances of a Service ............................ 26
      5. Service ID Must Be Unique .......................................... 26
      6. Customizing the Message Broker ................................. 27
      7. Tracing Bus Events ....................................................... 27
      8. Broadcasting Your Own Events ................................. 28
         8.1. Registering events in custom packages .................. 29
      9. Configuration properties ............................................ 30
   Spring Cloud Circuit Breaker ............................................... 30
      1. Usage Documentation .................................................. 30
         1.1. Configuring Resilience4J Circuit Breakers ............... 30
         1.2. Configuring Spring Retry Circuit Breakers .............. 34
      2. Building ................................................................. 36
         2.1. Basic Compile and Test ......................................... 36
         2.2. Documentation .................................................... 37
         2.3. Working with the code ......................................... 37
      3. Contributing ............................................................... 38
         3.1. Sign the Contributor License Agreement .................... 38
3.2. Code of Conduct ................................................................. 38
3.3. Code Conventions and Housekeeping ................................. 38
3.4. Checkstyle .......................................................................... 39
3.5. IDE setup ............................................................................ 41

Spring Boot Cloud CLI ................................................................. 44
1. Installation .............................................................................. 45
2. Running Spring Cloud Services in Development ..................... 45
   2.1. Adding Additional Applications ..................................... 47
3. Writing Groovy Scripts and Running Applications .................. 48
4. Encryption and Decryption ...................................................... 49

Spring Cloud for Cloud Foundry .................................................. 49
1. Discovery ................................................................................. 50
2. Single Sign On ........................................................................ 50
3. Configuration .......................................................................... 51

Cloud Native Applications .............................................................. 51
1. Spring Cloud Context: Application Context Services ................ 51
   1.1. The Bootstrap Application Context ................................. 52
   1.2. Application Context Hierarchies ..................................... 52
   1.3. Changing the Location of Bootstrap Properties ................. 53
   1.4. Overriding the Values of Remote Properties ................... 53
   1.5. Customizing the Bootstrap Configuration ....................... 54
   1.6. Customizing the Bootstrap Property Sources .................. 54
   1.7. Logging Configuration ..................................................... 55
   1.8. Environment Changes ...................................................... 55
   1.9. Refresh Scope ................................................................ 56
   1.10. Encryption and Decryption .......................................... 57
   1.11. Endpoints ...................................................................... 57
2. Spring Cloud Commons: Common Abstractions ......................... 58
   2.1. The @EnableDiscoveryClient Annotation ....................... 58
   2.2. ServiceRegistry .............................................................. 59
   2.3. Spring RestTemplate as a Load Balancer Client ................ 61
   2.4. Spring WebClient as a Load Balancer Client ................... 62
   2.5. Multiple RestTemplate Objects ...................................... 64
   2.6. Multiple WebClient Objects ............................................ 65
   2.7. Spring WebFlux WebClient as a Load Balancer Client .......... 66
   2.8. Ignore Network Interfaces .............................................. 68
   2.9. HTTP Client Factories ................................................... 69
   2.10. Enabled Features .......................................................... 69
   2.11. Spring Cloud Compatibility Verification ......................... 70
3. Spring Cloud LoadBalancer ...................................................... 71
   3.1. Switching between the load-balancing algorithms ............. 72
7. Global Filters

7.1. Combined Global Filter and GatewayFilter Ordering ........................................... 233
7.2. Forward Routing Filter ......................................................................................... 233
7.3. The ReactiveLoadBalancerClientFilter .............................................................. 234
7.4. The Netty Routing Filter ....................................................................................... 234
7.5. The Netty Write Response Filter ......................................................................... 235
7.6. The RouteToRequestUrl Filter .............................................................................. 235
7.7. The Websocket Routing Filter .............................................................................. 235
7.8. The Gateway Metrics Filter .................................................................................. 236
7.9. Marking An Exchange As Routed ......................................................................... 236
8. HttpHeadersFilters ................................................................. 237
  8.1. Forwarded Headers Filter ................................................. 237
  8.2. RemoveHopByHop Headers Filter ...................................... 237
  8.3. XForwarded Headers Filter .............................................. 237
9. TLS and SSL ............................................................... 238
  9.1. TLS Handshake ............................................................ 239
10. Configuration .............................................................. 239
11. Route Metadata Configuration ........................................... 240
12. Http timeouts configuration ............................................... 241
  12.1. Global timeouts ......................................................... 241
  12.2. Per-route timeouts ..................................................... 242
  12.3. Fluent Java Routes API ............................................... 242
  12.4. The DiscoveryClient Route Definition Locator ................... 243
13. Reactor Netty Access Logs ................................................ 244
14. CORS Configuration ........................................................ 245
15. Actuator API ...................................................................... 246
  15.1.Verbose Actuator Format ............................................... 246
  15.2. Retrieving Route Filters .............................................. 247
  15.3. Refreshing the Route Cache ........................................... 248
  15.4. Retrieving the Routes Defined in the Gateway .................... 248
  15.5. Retrieving Information about a Particular Route .................. 249
  15.6. Creating and Deleting a Particular Route ........................... 250
  15.7. Recap: The List of All endpoints ..................................... 250
16. Troubleshooting ............................................................. 251
  16.1. Log Levels .................................................................. 251
  16.2. Wiretap ...................................................................... 251
17. Developer Guide ............................................................. 251
  17.1. Writing Custom Route Predicate Factories......................... 251
  17.2. Writing Custom GatewayFilter Factories ......................... 252
  17.3. Writing Custom Global Filters ..................................... 254
18. Building a Simple Gateway by Using Spring MVC or Webflux ....... 255
19. Configuration properties .................................................... 257

Spring Cloud Kubernetes ............................................................ 257
1. Why do you need Spring Cloud Kubernetes? ......................... 257
2. Starters ........................................................................... 257
3. DiscoveryClient for Kubernetes .......................................... 259
4. Kubernetes native service discovery .................................... 261
5. Kubernetes PropertySource implementations ....................... 262
  5.1. Using a ConfigMap PropertySource ................................. 262
  5.2. Secrets PropertySource ................................................. 268
  5.3. PropertySource Reload .................................................. 272
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: 2020.0.3-SNAPSHOT

Supported Boot Version: 2.4.5-SNAPSHOT
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

2. Release Train Versions

Table 1. Release Train Project Versions

<table>
<thead>
<tr>
<th>Project Name</th>
<th>Project Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>spring-boot</td>
<td>2.4.5-SNAPSHOT</td>
</tr>
<tr>
<td>spring-cloud-build</td>
<td>3.0.3-SNAPSHOT</td>
</tr>
<tr>
<td>spring-cloud-bus</td>
<td>3.0.3-SNAPSHOT</td>
</tr>
<tr>
<td>spring-cloud-circuitbreaker</td>
<td>2.0.2-SNAPSHOT</td>
</tr>
<tr>
<td>spring-cloud-cli</td>
<td>3.0.3-SNAPSHOT</td>
</tr>
<tr>
<td>spring-cloud-cloudfoundry</td>
<td>3.0.2-SNAPSHOT</td>
</tr>
<tr>
<td>spring-cloud-commons</td>
<td>3.0.3-SNAPSHOT</td>
</tr>
<tr>
<td>spring-cloud-config</td>
<td>3.0.4-SNAPSHOT</td>
</tr>
<tr>
<td>spring-cloud-consul</td>
<td>3.0.3-SNAPSHOT</td>
</tr>
<tr>
<td>spring-cloud-contract</td>
<td>3.0.3-SNAPSHOT</td>
</tr>
<tr>
<td>spring-cloud-function</td>
<td>3.1.3-SNAPSHOT</td>
</tr>
<tr>
<td>spring-cloud-consul-gateway</td>
<td>3.0.3-SNAPSHOT</td>
</tr>
<tr>
<td>spring-cloud-kubernetes</td>
<td>2.0.3-SNAPSHOT</td>
</tr>
<tr>
<td>spring-cloud-netflix</td>
<td>3.0.3-SNAPSHOT</td>
</tr>
<tr>
<td>spring-cloud-openfeign</td>
<td>3.0.3-SNAPSHOT</td>
</tr>
<tr>
<td>spring-cloud-sleuth</td>
<td>3.0.3-SNAPSHOT</td>
</tr>
<tr>
<td>spring-cloud-stream</td>
<td>3.1.3-SNAPSHOT</td>
</tr>
<tr>
<td>spring-cloud-task</td>
<td>2.3.2-SNAPSHOT</td>
</tr>
<tr>
<td>spring-cloud-vault</td>
<td>3.0.3-SNAPSHOT</td>
</tr>
</tbody>
</table>
Spring Cloud Build

Spring Cloud Build is a common utility project for Spring Cloud to use for plugin and dependency management.

1. Building and Deploying

To install locally:

```bash
$ mvn install -s .settings.xml
```

and to deploy snapshots to repo.spring.io:

```bash
$ mvn deploy
-DaltSnapshotDeploymentRepository=repo.spring.io::default::https://repo.spring.io/snap shot
```

for a RELEASE build use

```bash
$ mvn deploy
-DaltReleaseDeploymentRepository=repo.spring.io::default::https://repo.spring.io/release
```

and for jcenter use

```bash
$ mvn deploy
```

and for Maven Central use

```bash
$ mvn deploy -P central -DaltReleaseDeploymentRepository=sonatype-nexus-staging::default::https://oss.sonatype.org/service/local/staging/deploy/maven2
```

(the "central" profile is available for all projects in Spring Cloud and it sets up the gpg jar signing, and the repository has to be specified separately for this project because it is a parent of the starter parent which users in turn have as their own parent).
2. 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.

2.1. 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.

2.2. 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.

2.3. 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).
2.4. 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

2.4.1. 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.failsOnError>true</maven-checkstyle-plugin.failsOnError> ①
  <maven-checkstyle-plugin.failsOnViolation>true ②
  <maven-checkstyle-plugin.includeTestSourceDirectory>true 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 ${project.root}/src/checkstyle/checkstyle-suppressions.xml with your suppressions. Example:
It's advisable to copy the `${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
```

### 2.5. IDE setup

#### 2.5.1. 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/
```
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.
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.


⚠️ Remember to set the Scan Scope to All sources since we apply checkstyle rules for production and test sources.

### 3. Flattening the POMs

To avoid propagating build setup that is required to build a Spring Cloud project, we’re using the maven flatten plugin. It has the advantage of letting you use whatever features you need while publishing “clean” pom to the repository.

In order to add it, add the org.codehaus.mojo:flatten-maven-plugin to your pom.xml.
4. Reusing the documentation

Spring Cloud Build publishes its `spring-cloud-build-docs` module that contains helpful scripts (e.g. README generation ruby script) and css, xslt and images for the Spring Cloud documentation. If you want to follow the same convention approach of generating documentation just add these plugins to your `docs` module.
<properties>
  <upload-docs-zip.phase>deploy</upload-docs-zip.phase> ⑧
</properties>
<profiles>
  <profile>
    <id>docs</id>
    <build>
      <plugins>
        <plugin>
          <groupId>pl.project13.maven</groupId>
          <artifactId>git-commit-id-plugin</artifactId> ①
        </plugin>
        <plugin>
          <groupId>org.apache.maven.plugins</groupId>
          <artifactId>maven-dependency-plugin</artifactId> ②
        </plugin>
        <plugin>
          <groupId>org.apache.maven.plugins</groupId>
          <artifactId>maven-resources-plugin</artifactId> ③
        </plugin>
        <plugin>
          <groupId>org.codehaus.mojo</groupId>
          <artifactId>exec-maven-plugin</artifactId> ④
        </plugin>
        <plugin>
          <groupId>org.asciidoctor</groupId>
          <artifactId>asciidoctor-maven-plugin</artifactId> ⑤
        </plugin>
        <plugin>
          <groupId>org.apache.maven.plugins</groupId>
          <artifactId>maven-antrun-plugin</artifactId> ⑥
        </plugin>
        <plugin>
          <artifactId>maven-deploy-plugin</artifactId> ⑦
        </plugin>
      </plugins>
    </build>
  </profile>
</profiles>

① This plugin downloads sets up all the git information of the project
② This plugin downloads the resources of the spring-cloud-build-docs module
③ This plugin unpacks the resources of the spring-cloud-build-docs module
④ This plugin generates an adoc file with all the configuration properties from the classpath
⑤ This plugin is required to parse the Asciidoctor documentation
⑥ This plugin is required to copy resources into proper final destinations and to generate main README.adoc and to assert that no files use unresolved links
This plugin ensures that the generated zip docs will get published

This property turns on the "deploy" phase for <7>

The order of plugin declaration is important!

In order for the build to generate the adoc file with all your configuration properties, your docs module should contain all the dependencies on the classpath, that you would want to scan for configuration properties. The file will be output to `${docsModule}/src/main/asciidoc/_configprops.adoc` file (configurable via the `configprops.path` property).

If you want to modify which of the configuration properties are put in the table, you can tweak the `configprops.inclusionPattern` pattern to include only a subset of the properties (e.g. `<configprops.inclusionPattern>spring.sleuth.*</configprops.inclusionPattern>`).

Spring Cloud Build Docs comes with a set of attributes for asciidoc that you can reuse.
5. Updating the guides

We assume that your project contains guides under the guides folder.
This means that the project contains 3 guides that would correspond to the following guides in Spring Guides org.

- [github.com/spring-guides/gs-guide1](https://github.com/spring-guides/gs-guide1)
- [github.com/spring-guides/gs-guide2](https://github.com/spring-guides/gs-guide2)
- [github.com/spring-guides/gs-guide3](https://github.com/spring-guides/gs-guide3)

If you deploy your project with the `-Pguides` profile like this

```
$ ./mvnw clean deploy -Pguides
```

what will happen is that for GA project versions, we will clone `gs-guide1`, `gs-guide2` and `gs-guide3` and update their contents with the ones being under your `guides` project.

You can skip this by either not adding the `guides` profile, or passing the `-DskipGuides` system property when the profile is turned on.

You can configure the project version passed to guides via the `guides-project.version` (defaults to `${project.version}`). The phase at which guides get updated can be configured by `guides-update.phase` (defaults to `deploy`).

### 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](https://github).

### 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)

```yaml
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.

**2. Bus Endpoints**

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

**2.1. Bus Refresh Endpoint**

The /actuator/busrefresh endpoint clears the RefreshScope cache and rebinds @ConfigurationProperties. See the Refresh Scope documentation for more information.

To expose the /actuator/busrefresh endpoint, you need to add following configuration to your application:

```
management.endpoints.web.exposure.include=busrefresh
```

**2.2. Bus Env Endpoint**

The /actuator/busenv endpoint updates each instances environment with the specified key/value pair across multiple instances.

To expose the /actuator/busenv endpoint, you need to add following configuration to your application:
The /actuator/busenv endpoint accepts POST requests with the following shape:

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

### 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 `/busrefresh/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.

### 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, `/busenv/customers:**` targets all instances of the “customers” service regardless of the rest of the service ID.

### 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.
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. 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:
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.

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.

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:

```java
package com.acme;

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

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

```java
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:

```java
package com.acme;

@Configuration
//@RemoteApplicationEventScan("
//@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.

9. Configuration properties

To see the list of all Bus related configuration properties please check the Appendix page.

Spring Cloud Circuit Breaker

2020.0.3-SNAPSHOT

1. Usage Documentation

The Spring Cloud CircuitBreaker project contains implementations for Resilience4J and Spring Retry. The APIs implemented in Spring Cloud CircuitBreaker live in Spring Cloud Commons. The usage documentation for these APIs are located in the Spring Cloud Commons documentation.

1.1. Configuring Resilience4J Circuit Breakers

1.1.1. 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

1.1.2. Auto-Configuration

You can disable the Resilience4J auto-configuration by setting `spring.cloud.circuitbreaker.resilience4j.enabled` to `false`.

1.1.3. 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());
}

1.1.4. 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.configureDefault(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
customizer = (customizer) -> {
    customizer.addCircuitBreakerCustomizer(circuitBreaker ->
        circuitBreaker.getEventPublisher()
            .onError(normalFluxErrorConsumer).onSuccess(normalFluxSuccessConsumer),
            "normalflux";
    }
}

Reactive Example

@Bean
customizer = (customizer) -> {
    customizer.configure(builder -> builder
        .timeLimiterConfig(TimeLimiterConfig.custom().timeoutDuration(Duration.ofSeconds(2))
            .build())
        .circuitBreakerConfig(CircuitBreakerConfig.ofDefaults()), "slow",
            "slowflux");
    customizer.addCircuitBreakerCustomizer(circuitBreaker ->
        circuitBreaker.getEventPublisher()
            .onError(normalFluxErrorConsumer).onSuccess(normalFluxSuccessConsumer),
            "normalflux";
    }
}

1.1.5. Bulkhead pattern supporting

If resilience4j-bulkhead is on the classpath, Spring Cloud CircuitBreaker will wrap all methods with a Resilience4j Bulkhead. You can disable the Resilience4j Bulkhead by setting spring.cloud.circuitbreaker.bulkhead.resilience4j.enabled to false.

Spring Cloud CircuitBreaker Resilience4j provides two implementation of bulkhead pattern:

- a SemaphoreBulkhead which uses Semaphores
- a FixedThreadPoolBulkhead which uses a bounded queue and a fixed thread pool.

By default, Spring Cloud CircuitBreaker Resilience4j uses FixedThreadPoolBulkhead. For more information on implementation of Bulkhead patterns see the Resilience4j Bulkhead.

The Customizer<Resilience4jBulkheadProvider> can be used to provide a default Bulkhead and ThreadPoolBulkhead configuration.
public Customizer<Resilience4jBulkheadProvider> defaultBulkheadCustomizer() {
    return provider -> provider.configureDefault(id -> new Resilience4jBulkheadConfigurationBuilder()
            .bulkheadConfig(BulkheadConfig.custom().maxConcurrentCalls(4).build())
            .threadPoolBulkheadConfig(ThreadPoolBulkheadConfig.custom().coreThreadPoolSize(1).maxThreadPoolSize(1).build())
            .build());
}

1.1.6. Specific Bulkhead Configuration

Similarly to proving a default 'Bulkhead' or 'ThreadPoolBulkhead' configuration, you can create a Customize bean this is passed a Resilience4jBulkheadProvider.

public Customizer<Resilience4jBulkheadProvider> slowBulkheadProviderCustomizer() {
    return provider -> provider.configure(builder -> builder
            .bulkheadConfig(BulkheadConfig.custom().maxConcurrentCalls(1).build())
            .threadPoolBulkheadConfig(ThreadPoolBulkheadConfig.ofDefaults()),
            "slowBulkhead");
}

In addition to configuring the Bulkhead that is created you can also customize the bulkhead and thread pool bulkhead after they have been created but before they are returned to caller. To do this you can use the addBulkheadCustomizer and addThreadPoolBulkheadCustomizer methods.

Bulkhead Example

public Customizer<Resilience4jBulkheadProvider> customizer() {
    return provider -> provider.addBulkheadCustomizer(bulkhead ->
        bulkhead.getEventPublisher()
        .onCallRejected(slowRejectedConsumer)
        .onCallFinished(slowFinishedConsumer), "slowBulkhead");
}

Thread Pool Bulkhead Example
@Bean
public Customizer<Resilience4jBulkheadProvider> slowThreadPoolBulkheadCustomizer()
{
    return provider -> provider.addThreadPoolBulkheadCustomizer(threadPoolBulkhead
    -> threadPoolBulkhead.getEventPublisher()
        .onCallRejected(slowThreadPoolRejectedConsumer)
        .onCallFinished(slowThreadPoolFinishedConsumer),
"slowThreadPoolBulkhead");
}

1.1.7. Bulkhead Properties Configuration
You can configure ThreadPoolBulkhead and SemaphoreBulkhead instances in your application’s configuration properties file. Property configuration has higher priority than Java Customizer configuration.

```yaml
resilience4j.thread-pool-bulkhead:
  instances:
    backendA:
      maxThreadPoolSize: 1
      coreThreadPoolSize: 1
resilience4j.bulkhead:
  instances:
    backendB:
      maxConcurrentCalls: 10
```

For more information on the Resilience4j property configuration, see Resilience4J Spring Boot 2 Configuration.

1.1.8. 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

1.2. 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.

1.2.1. 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.

```java
@Bean
public Customize<SpringRetryCircuitBreakerFactory> defaultCustomizer() {
    return factory -> factory.configureDefault(id -> new SpringRetryConfigBuilder(id)
            .retryPolicy(new TimeoutRetryPolicy()).build());
}
```

1.2.2. Specific Circuit Breaker Configuration

Similarly to providing a default configuration, you can create a Customize bean this is passed a SpringRetryCircuitBreakerFactory.

```java
@Bean
public Customize<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
class 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) {
        }
    }));
}

2. Building

2.1. 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
runtime 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`).

### 2.2. 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
`${main.basedir}` (defaults to `$/opt/jenkins/data/workspace/spring-cloud-release-train-docs-master-
ci/train-docs/target/unpacked-docs`, 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.

### 2.3. 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.

#### 2.3.1. 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.

#### 2.3.2. 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.

2.3.3. 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.

3. 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.

3.1. 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.

3.2. 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.

3.3. 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).

### 3.4. 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

### 3.4.1. 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.failsOnError>true</maven-checkstyle-plugin.failsOnError> ①
<maven-checkstyle-plugin.failsOnViolation>true ②
<maven-checkstyle-plugin.includeTestSourceDirectory>true ③
</maven-checkstyle-plugin.includeTestSourceDirectory>
</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 ${project.root}/src/checkstyle/checkstyle-suppressions.xml with your suppressions. Example:
It's advisable to copy the `${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:

```bash
$ curl https://raw.githubusercontent.com/spring-cloud/spring-cloud-build/master/.editorconfig -o .editorconfig
$ touch .springformat
```

### 3.5. IDE setup

#### 3.5.1. 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
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.
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 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.

1. Installation

To install, make sure you have Spring Boot CLI (2.0.0 or better):

```
$ spring version
Spring CLI v2.2.3.RELEASE
```

E.g. for SDKMan users

```
$ sdk install springboot 2.2.3.RELEASE
$ sdk use springboot 2.2.3.RELEASE
```

and install the Spring Cloud plugin

```
$ mvn install
$ spring install org.springframework.cloud:spring-cloud-cli:2.2.0.RELEASE
```

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).

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:
<table>
<thead>
<tr>
<th>Service</th>
<th>Name</th>
<th>Address</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>eureka</td>
<td>Eureka Server</td>
<td>localhost:8761</td>
<td>Eureka server for service registration and discovery. All the other services show up in its catalog by default.</td>
</tr>
<tr>
<td>configserver</td>
<td>Config Server</td>
<td>localhost:8888</td>
<td>Spring Cloud Config Server running in the &quot;native&quot; profile and serving configuration from the local directory ./launcher</td>
</tr>
<tr>
<td>h2</td>
<td>H2 Database</td>
<td>localhost:9095 (console), jdbc:h2:tcp://localhost:9096/{data}</td>
<td>Relation database service. Use a file path for <code>{data}</code> (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.</td>
</tr>
<tr>
<td>kafka</td>
<td>Kafka Broker</td>
<td>localhost:9091 (actuator endpoints), localhost:9092</td>
<td></td>
</tr>
<tr>
<td>dataflow</td>
<td>Dataflow Server</td>
<td>localhost:9393</td>
<td>Spring Cloud Dataflow server with UI at /admin-ui. Connect the Dataflow shell to target at root path.</td>
</tr>
<tr>
<td>zipkin</td>
<td>Zipkin Server</td>
<td>localhost:9411</td>
<td>Zipkin Server with UI for visualizing traces. Stores span data in memory and accepts them via HTTP POST of JSON data.</td>
</tr>
<tr>
<td>Service</td>
<td>Name</td>
<td>Address</td>
<td>Description</td>
</tr>
<tr>
<td>---------</td>
<td>--------------------</td>
<td>-----------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>stubrunner</td>
<td>Stub Runner Boot</td>
<td>localhost:8750</td>
<td>Downloads WireMock stubs, starts WireMock and feeds the started servers with stored stubs. Pass stubrunner.ids to pass stub coordinates and then go to localhost:8750/stubs.</td>
</tr>
</tbody>
</table>

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
```

### 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
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

```groovy
app.groovy

@EnableEurekaServer
class Eureka {}
```

which you can run from the command line like this

```bash
$ 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:
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.

```bash
$ 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.

```bash
$ spring encrypt mysecret --key @${HOME}/.ssh/id_rsa.pub
AQAjPgt3eF2tQXwt8tsHAVv/QHiY5sI2dRcR+...
```

Spring Cloud for Cloud Foundry

Spring Cloud for Cloud Foundry 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.

1. Discovery

Here’s a Spring Cloud app with Cloud Foundry discovery:

```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.

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`. 
3. Configuration

To see the list of all Spring Cloud Sloud Foundry related configuration properties please check the Appendix page.

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 github.

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.
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**

```yaml
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).

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.

### 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.

### 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.

### 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.

### 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:
```java
@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:

```
```

### 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.

### 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`.

### 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.extrar-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:
@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).

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.

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. To enabled this endpoint you must set management.endpoint.env.post.enabled=true.
- /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.

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).

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.

2.1.1. Health Indicators

Commons auto-configures the following Spring Boot health indicators.

DiscoveryClientHealthIndicator

This health indicator is based on the currently registered DiscoveryClient implementation.

• To disable entirely, set spring.cloud.discovery.client.health-indicator.enabled=false.
• To disable the description field, set spring.cloud.discovery.client.health-indicator.include-
• To disable service retrieval, set `spring.cloud.discovery.client.health-indicator.use-services-query=false`. By default, the indicator invokes the client's `getServices` method. In deployments with many registered services it may too costly to retrieve all services during every check. This will skip the service retrieval and instead use the client's `probe` method.

**DiscoveryCompositeHealthContributor**

This composite health indicator is based on all registered `DiscoveryHealthIndicator` beans. To disable, set `spring.cloud.discovery.client.composite-indicator.enabled=false`.

### 2.1.2. 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.{clientIdentifier}.discovery.order` (or `eureka.client.order` for Eureka) property to the desired value.

### 2.1.3. 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.

### 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:
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.

### 2.2.1. 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`.

### 2.2.2. 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.

### 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:

```java
@Configuration
public class MyConfiguration {

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

public class MyClass {
    @Autowired
    private RestTemplate restTemplate;

    public String doOtherStuff() {
        String results = restTemplateForObject("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.
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:

```java
@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.

2.4.1. Retrying Failed Requests

A load-balanced `RestTemplate` can be configured to retry failed requests. By default, this logic is disabled. For the non-reactive version (with `RestTemplate`), you can enable it by adding `Spring Retry` to your application’s classpath. For the reactive version (with `WebTestClient`), you need to set `spring.cloud.loadbalancer.retry.enabled=true`.

If you would like to disable the retry logic with Spring Retry or Reactive Retry on the classpath, you can set `spring.cloud.loadbalancer.retry.enabled=false`.

For the non-reactive implementation, 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.
For the reactive implementation, you just need to enable it by setting `spring.cloud.loadbalancer.retry.backoff.enabled` to `false`.

You can set:

- `spring.cloud.loadbalancer.retry.maxRetriesOnSameServiceInstance` - indicates how many times a request should be retried on the same `ServiceInstance` (counted separately for every selected instance)
- `spring.cloud.loadbalancer.retry.maxRetriesOnNextServiceInstance` - indicates how many times a request should be retried a newly selected `ServiceInstance`
- `spring.cloud.loadbalancer.retry.retryableStatusCodes` - the status codes on which to always retry a failed request.

For the reactive implementation, you can additionally set:
- `spring.cloud.loadbalancer.retry.backoff.minBackoff` - Sets the minimum backoff duration (by default, 5 milliseconds)
- `spring.cloud.loadbalancer.retry.backoff.maxBackoff` - Sets the maximum backoff duration (by default, max long value of milliseconds)
- `spring.cloud.loadbalancer.retry.backoff.jitter` - Sets the jitter used for calculating the actual backoff duration for each call (by default, 0.5).

For the reactive implementation, you can also implement your own `LoadBalancerRetryPolicy` to have more detailed control over the load-balanced call retries.

For load-balanced retries, by default, we wrap the `ServiceInstanceListSupplier` bean with `RetryAwareServiceInstanceListSupplier` to select a different instance from the one previously chosen, if available. You can disable this behavior by setting the value of `spring.cloud.loadbalancer.retry.avoidPreviousInstance` to `false`.

```java
@Configuration
public class MyConfiguration {
    @Bean
    LoadBalancedRetryFactory retryFactory() {
        return new LoadBalancedRetryFactory() {
            @Override
            public BackOffPolicy createBackOffPolicy(String service) {
                @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 `LoadBalancedRetryListenserFactory` and return the `RetryListener` array you would like to use for a given service, as the following example shows:
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.

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);
    }
}

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]
2.7.1. 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:

```java
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.

2.7.2. 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.

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

```yaml
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

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

See [Inet4Address.html.isSiteLocalAddress()](https://docs.oracle.com/en/java/javase/11/docs/api/java.net/Inet4Address.html.isSiteLocalAddress()) for more details about what constitutes a site-local address.

### 2.9. HTTP Client Factories

Spring Cloud Commons provides beans for creating both Apache HTTP clients ([ApacheHttpClientFactory](https://github.com/spring-projects/spring-boot/blob/main/spring-boot-project/spring-boot-autoconfigure/src/main/java/org/springframework/cloud/geometry/ApacheHttpClientFactory.java)) and OK HTTP clients ([OkHttpClientFactory](https://github.com/spring-projects/spring-cloud/blob/main/spring-cloud-geometry/src/main/java/org/springframework/cloud/geometry/OkHttpClientFactory.java)). The [OkHttpClientFactory](https://github.com/spring-projects/spring-cloud/blob/main/spring-cloud-geometry/src/main/java/org/springframework/cloud/geometry/OkHttpClientFactory.java) 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: [ApacheHttpClientConnectionManagerFactory](https://github.com/spring-projects/spring-cloud/blob/main/spring-cloud-geometry/src/main/java/org/springframework/cloud/geometry/ApacheHttpClientConnectionManagerFactory.java) for the Apache HTTP client and [OkHttpClientConnectionPoolFactory](https://github.com/spring-projects/spring-cloud/blob/main/spring-cloud-geometry/src/main/java/org/springframework/cloud/geometry/OkHttpClientConnectionPoolFactory.java) 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](https://github.com/spring-projects/spring-boot/blob/main/spring-boot-project/spring-boot-autoconfigure/src/main/java/org/springframework/cloud/geometry/ApacheHttpClientFactory.java) or [OkHttpClient.Builder](https://github.com/spring-projects/spring-cloud/blob/main/spring-cloud-geometry/src/main/java/org/springframework/cloud/geometry/OkHttpClientFactory.java), 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`.

### 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.
2.10.1. 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.

2.10.2. Declaring features

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

```java
@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`.

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. 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` and `Random` implementations have 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.

It is possible to disable Spring Cloud LoadBalancer by setting the value of `spring.cloud.loadbalancer.enabled` to `false`. 
3.1. Switching between the load-balancing algorithms

The ReactLoadBalancer implementation that is used by default is RoundRobinLoadBalancer. To switch to a different implementation, either for selected services or all of them, you can use the custom LoadBalancer configurations mechanism.

For example, the following configuration can be passed via @LoadBalancerClient annotation to switch to using the RandomLoadBalancer:

```java
public class CustomLoadBalancerConfiguration {
    @Bean
    ReactLoadBalancer<ServiceInstance> randomLoadBalancer(Environment environment, LoadBalancerClientFactory loadBalancerClientFactory) {
        String name = environment.getProperty(LoadBalancerClientFactory.PROPERTY_NAME);
        return new RandomLoadBalancer(loadBalancerClientFactory
                .getLazyProvider(name, ServiceInstanceListSupplier.class),
                name);
    }
}
```

The classes you pass as @LoadBalancerClient or @LoadBalancerClients configuration arguments should either not be annotated with @Configuration or be outside component scan scope.

3.2. Spring Cloud LoadBalancer integrations

In order to make it easy to use Spring Cloud LoadBalancer, we provide ReactLoadBalancerExchangeFilterFunction 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 ReactLoadBalancerExchangeFilterFunction

3.3. 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.

3.3.1. 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`.

### 3.3.2. 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.

### 3.3.3. 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.

### 3.4. 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:

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

3.5. 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.

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.
If using any of the Service Discovery-backed suppliers, adding this health-check mechanism is usually not necessary, as we retrieve the health state of the instances directly from the Service Registry.

The `HealthCheckServiceInstanceListSupplier` relies on having updated instances provided by a delegate flux. In the rare cases when you want to use a delegate that does not refresh the instances, even though the list of instances may change (such as the `ReactiveDiscoveryClientServiceInstanceListSupplier` provided by us), you can set `spring.cloud.loadbalancer.health-check.refetch-instances` to `true` to have the instance list refreshed by the `HealthCheckServiceInstanceListSupplier`. You can then also adjust the refresh intervals by modifying the value of `spring.cloud.loadbalancer.health-check.refetch-instances-interval` and opt to disable the additional healthcheck repetitions by setting `spring.cloud.loadbalancer.repeat-health-check` to `false` as every instances refetch will also trigger a healthcheck.

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

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);
    }
}

For the non-reactive stack, create this supplier with the withBlockingHealthChecks(). You can also pass your own WebClient or RestTemplate instance to be used for the checks.

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.

3.6. Same instance preference for LoadBalancer

You can set up the LoadBalancer in such a way that it prefers the instance that was previously selected, if that instance is available.

For that, you need to use SameInstancePreferenceServiceInstanceListSupplier. You can configure it either by setting the value of spring.cloud.loadbalancer.configurations to same-instance-preference or by providing your own ServiceInstanceListSupplier bean — for example:

public class CustomLoadBalancerConfiguration {

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

This is also a replacement for Zookeeper StickyRule.
3.7. Request-based Sticky Session for LoadBalancer

You can set up the LoadBalancer in such a way that it prefers the instance with `instanceId` provided in a request cookie. We currently support this if the request is being passed to the LoadBalancer through either `ClientRequestContext` or `ServerHttpRequestContext`, which are used by the SC LoadBalancer exchange filter functions and filters.

For that, you need to use the `RequestBasedStickySessionServiceInstanceListSupplier`. You can configure it either by setting the value of `spring.cloud.loadbalancer.configurations` to `request-based-sticky-session` or by providing your own `ServiceInstanceListSupplier` bean — for example:

```java
public class CustomLoadBalancerConfiguration {
    @Bean
    public ServiceInstanceListSupplier discoveryClientServiceInstanceListSupplier(
        ConfigurableApplicationContext context) {
        return ServiceInstanceListSupplier.builder()
            .withDiscoveryClient()
            .withRequestBasedStickySession()
            .build(context);
    }
}
```

For that functionality, it is useful to have the selected service instance (which can be different from the one in the original request cookie if that one is not available) to be updated before sending the request forward. To do that, set the value of `spring.cloud.loadbalancer.sticky-session.add-service-instance-cookie` to `true`.

By default, the name of the cookie is `sc-lb-instance-id`. You can modify it by changing the value of the `spring.cloud.loadbalancer.instance-id-cookie-name` property.

3.8. Spring Cloud LoadBalancer Hints

Spring Cloud LoadBalancer lets you set `String` hints that are passed to the LoadBalancer within the `Request` object and that can later be used in `ReactiveLoadBalancer` implementations that can handle them.

You can set a default hint for all services by setting the value of the `spring.cloud.loadbalancer.hint.default` property. You can also set a specific value for any given service by setting the value of the `spring.cloud.loadbalancer.hint.[SERVICE_ID]` property, substituting `[SERVICE_ID]` with the correct ID of your service. If the hint is not set by the user, `default` is used.

3.9. Hint-Based Load-Balancing

We also provide a `HintBasedServiceInstanceListSupplier`, which is a `ServiceInstanceListSupplier` implementation for hint-based instance selection.
**HintBasedServiceInstanceListSupplier** checks for a hint request header (the default header-name is `X-SC-LB-Hint`, but you can modify it by changing the value of the `spring.cloud.loadbalancer.hint-header-name` property) and, if it finds a hint request header, uses the hint value passed in the header to filter service instances.

If no hint header has been added, **HintBasedServiceInstanceListSupplier** uses **hint values from properties** to filter service instances.

If no hint is set, either by the header or by properties, all service instances provided by the delegate are returned.

While filtering, **HintBasedServiceInstanceListSupplier** looks for service instances that have a matching value set under the `hint` key in their `metadataMap`. If no matching instances are found, all instances provided by the delegate are returned.

You could use the following sample configuration to set it up:

```java
public class CustomLoadBalancerConfiguration {

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

### 3.10. Transform the load-balanced HTTP request

You can use the selected `ServiceInstance` to transform the load-balanced HTTP Request.

For `RestTemplate`, you need to implement and define `LoadBalancerRequestTransformer` as follows:
For **WebClient**, you need to implement and define `LoadBalancerClientRequestTransformer` as follows:

```java
@Bean
public LoadBalancerClientRequestTransformer transformer() {
    return new LoadBalancerClientRequestTransformer() {
        @Override
        public ClientRequest transformRequest(ClientRequest request, ServiceInstance instance) {
            return ClientRequest.from(request)
                .header("X-InstanceId", instance.getInstanceId())
                .build();
        }
    };
}
```

If multiple transformers are defined, they are applied in the order in which Beans are defined. Alternatively, you can use `LoadBalancerRequestTransformer.DEFAULT_ORDER` or `LoadBalancerClientRequestTransformer.DEFAULT_ORDER` to specify the order.

### 3.11. 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**.
3.12. 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:

```java
@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
@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();
    }
}

The classes you pass as @LoadBalancerClient or @LoadBalancerClients configuration arguments should either not be annotated with @Configuration or be outside component scan scope.

3.13. Spring Cloud LoadBalancer Lifecycle

One type of bean that it may be useful to register using Custom LoadBalancer configuration is LoadBalancerLifecycle.

The LoadBalancerLifecycle beans provide callback methods, named onStart(Request<RC> request), onStartRequest(Request<RC> request, Response<T> lbResponse) and onComplete(CompletionContext<RES, T, RC> completionContext), that you should implement to specify what actions should take place before and after load-balancing.

onStart(Request<RC> request) takes a Request object as a parameter. It contains data that is used to select an appropriate instance, including the downstream client request and hint. onStartRequest also takes the Request object and, additionally, the Response<T> object as parameters. On the other hand, a CompletionContext object is provided to the onComplete(CompletionContext<RES, T, RC> completionContext) method. It contains the LoadBalancer Response, including the selected service instance, the Status of the request executed against that service instance and (if available) the response returned to the downstream client, and (if an exception has occurred) the corresponding Throwable.

The supports(Class requestContextClass, Class responseClass, Class serverTypeClass) method can be used to determine whether the processor in question handles objects of provided types. If not overridden by the user, it returns true.

In the preceding method calls, RC means RequestContext type, RES means client response type, and T means returned server type.

We provide a `LoadBalancerLifecycle` bean called `MicrometerStatsLoadBalancerLifecycle`, which uses Micrometer to provide statistics for load-balanced calls.

In order to get this bean added to your application context, set the value of the `spring.cloud.loadbalancer.stats.micrometer.enabled` to `true` and have a `MeterRegistry` available (for example, by adding `Spring Boot Actuator` to your project).

`MicrometerStatsLoadBalancerLifecycle` registers the following meters in `MeterRegistry`:

- `loadbalancer.requests.active`: A gauge that allows you to monitor the number of currently active requests for any service instance (service instance data available via tags);
- `loadbalancer.requests.success`: A timer that measures the time of execution of any load-balanced requests that have ended in passing a response on to the underlying client;
- `loadbalancer.requests.failed`: A timer that measures the time of execution of any load-balanced requests that have ended with an exception;
- `loadbalancer.requests.discard`: A counter that measures the number of discarded load-balanced requests, i.e. requests where a service instance to run the request on has not been retrieved by the LoadBalancer.

Additional information regarding the service instances, request data, and response data is added to metrics via tags whenever available.

For some implementations, such as `BlockingLoadBalancerClient`, request and response data might not be available, as we establish generic types from arguments and might not be able to determine the types and read the data.

The meters are registered in the registry when at least one record is added for a given meter.

You can further configure the behavior of those metrics (for example, add publishing percentiles and histograms) by adding `MeterFilters`.

4. Spring Cloud Circuit Breaker

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.

4.1.1. Supported Implementations

Spring Cloud supports the following circuit-breaker implementations:
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:

```java
@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 run 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.

4.2.1. 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
class DemoControllerService {
  private ReactiveCircuitBreakerFactory cbFactory;
  private WebClient webClient;

  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.

### 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.
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}

6. Security

6.1. 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.

6.1.1. 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.)

6.1.2. 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:

```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):

```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.

### 7. Configuration Properties

To see the list of all Spring Cloud Commons related configuration properties please check the Appendix page.
Spring Cloud Config

2020.0.3-SNAPSHOT

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.

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:

```sh
$ 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:
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

For example:

```
curl localhost:8888/foo/development

curl localhost:8888/foo/development/master

curl localhost:8888/foo/development,db/master

curl localhost:8888/foo-development.yml

curl localhost:8888/foo-db.properties

curl localhost:8888/master/foo-db.properties
```
where \texttt{application} is injected as the \texttt{spring.config.name} in the \texttt{SpringApplication} (what is normally \texttt{application} in a regular Spring Boot app), \texttt{profile} is an active profile (or comma-separated list of properties), and \texttt{label} is an optional git label (defaults to \texttt{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:

```java
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.

### 1.1. Client Side Usage

To use these features in an application, you can build it as a Spring Boot application that depends on \texttt{spring-cloud-config-client} (for an example, see the test cases for the \texttt{config-client} or the sample application). The most convenient way to add the dependency is with a Spring Boot starter \texttt{org.springframework.cloud:spring-cloud-starter-config}. There is also a parent pom and BOM (\texttt{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:
Now you can create a standard Spring Boot application, such as the following HTTP server:
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 `application.properties` as shown in the following example:

```
spring.config.import=optional:configserver: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 `application.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 Config Server properties show up in the `/env` endpoint as a high-priority property source, as shown in the following example.
$ curl localhost:8080/env
{
  "activeProfiles": [],
  {
    "name": "servletContextInitParams",
    "properties": {}
  },
  {
    "name": "configserver:https://github.com/spring-cloud-samples/config-repo/foo.properties",
    "properties": {
      "foo": {
        "value": "bar",
        "origin": "Config Server https://github.com/spring-cloud-samples/config-repo/foo.properties:2:12"
      }
    }
  },
  ...
}

A property source called `configserver:<URL of remote repository>/<file name>` contains the `foo` property with a value of `bar`.

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

If you use Spring Cloud Config Client, you need to set the `spring.config.import` property in order to bind to Config Server. You can read more about it in the Spring Cloud Config Reference Guide.

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:

```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.

### 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:

```
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.

### 2.1.1. 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`).

```yaml
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.

```yaml
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:

```yaml
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:
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:

```yaml
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 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 search-paths. The following example shows a config file at the top level:

```yaml
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
```

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:

```yaml
spring:
  cloud:
    config:
      server:
        git:
          uri: https://github.com/spring-cloud-samples/config-repo
          search-paths:
            - foo
            - bar*
```
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:

```yaml
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 OPENSSH 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:

```
https://git-codecommit.${AWS_REGION}.amazonaws.com/v1/repos/${repo}.
```

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:

```properties
spring:
  cloud:
    config:
      server:
        git:
          uri: git@gitserver.com:team/repo1.git
          ignoreLocalSshSettings: true
          hostKey: someHostKey
          hostKeyAlgorithm: ssh-rsa
          privateKey: |
            -----BEGIN RSA PRIVATE KEY-----
            MIIEpIBAAKCAQEAx4UbaDzY5xjW6hc9jwN0mX33XpTDVW9WqH5AkARbtAC3DqX
            IXFMpgw3K45jxRb93f8tv9vL3rD9CUG1Gv4FM+o7ds7FRES5RTjv2RT/JVNJCqOF
            -----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 `spring.cloud.config.server.git.ignoreLocalSshSettings`, not inside a repository definition.
privateKey | Valid SSH private key. Must be set if `ignoreLocalSshSettings` is true and Git URI is SSH format.
hostKey | Valid SSH host key. Must be set if `hostKeyAlgorithm` is also set.
hostKeyAlgorithm | One of `ssh-dss`, `ssh-rsa`, `ecdsa-sha2-nistp256`, `ecdsa-sha2-nistp384`, or `ecdsa-sha2-nistp521`. Must be set if `hostKey` is also set.
strictHostKeyChecking | true or false. If false, ignore errors with host key.
knownHostsFile | Location of custom `.known_hosts` file.
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:

```yaml
spring:
  cloud:
    config:
      server:
        git:
          uri: https://github.com/spring-cloud-samples/config-repo
          search-paths: '{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:

```yaml
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:

```yaml
spring:
  cloud:
    config:
      server:
        git:
          uri: https://git/team-a/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.

### 2.1.2. 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.

### 2.1.3. 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.

### 2.1.4. 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:

<table>
<thead>
<tr>
<th>Name</th>
<th>Default Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>host</td>
<td>127.0.0.1</td>
</tr>
<tr>
<td>port</td>
<td>8200</td>
</tr>
<tr>
<td>scheme</td>
<td>http</td>
</tr>
<tr>
<td>backend</td>
<td>secret</td>
</tr>
<tr>
<td>defaultKey</td>
<td>application</td>
</tr>
<tr>
<td>profileSeparator</td>
<td>,</td>
</tr>
<tr>
<td>kvVersion</td>
<td>1</td>
</tr>
<tr>
<td>Name</td>
<td>Default Value</td>
</tr>
<tr>
<td>------------------------</td>
<td>---------------</td>
</tr>
<tr>
<td>skipSslValidation</td>
<td>false</td>
</tr>
<tr>
<td>timeout</td>
<td>5</td>
</tr>
<tr>
<td>namespace</td>
<td>null</td>
</tr>
</tbody>
</table>

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:
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:
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.

### 2.1.5. 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>
<thead>
<tr>
<th>Property Name</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>host</td>
<td>The host of the proxy.</td>
</tr>
<tr>
<td>port</td>
<td>The port with which to access the proxy.</td>
</tr>
<tr>
<td>nonProxyHosts</td>
<td>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.</td>
</tr>
<tr>
<td>username</td>
<td>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.</td>
</tr>
<tr>
<td>password</td>
<td>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.</td>
</tr>
</tbody>
</table>

The following configuration uses an HTTPS proxy to access a Git repository.
2.1.6. 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:

```bash
$ 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"}
```

### 2.1.7. 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.

You can disable autoconfiguration for `JdbcEnvironmentRepository` by setting the `spring.cloud.config.server.jdbc.enabled` property to `false`.

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).

### 2.1.8. 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.
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 running 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.

### 2.1.9. 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}.yml` 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.

2.1.10. 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.

```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:

```yaml
spring:
  profiles:
    active: credhub
cloud:
  config:
    server:
      credhub:
        url: https://credhub:8844
```
The properties should be stored as JSON, such as:

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

```bash
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:

```json
{
  "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**

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

2.1.11. 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.

2.1.12. 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:

```yaml
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.

### 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:

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

You can disable the Health Indicator by setting `management.health.config.enabled=false`.

### 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).
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 `application.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 -s -d mysecret
682bc583f4641835fa2db009355293665d2647dade3375c0ee201de2a49f7bda
```
If you are testing with curl, then use `--data-urlencode` (instead of `-d`) and prefix the value to encrypt with `=` (curl requires this) or set an explicit `Content-Type: text/plain` to make sure curl encodes the data correctly when there are special characters (`+` is particularly tricky).

Be sure not to include any of the curl command statistics in the encrypted value, this is why the examples use the `-s` option to silence them. 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 -s -d
682bc583f4641835fa2db009355293665d2647dade3375c0ee201de2a49f7bda
mysecret
```

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).
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

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

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.

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:
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.

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.
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 (".yml", "yaml" 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. 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

### 4.1. Git, SVN, and Native Backends

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

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

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

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

The `application.yml` might resemble the following listing:

```yaml
nginx:
    server:
        name: example.com
---

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

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

### 4.3. 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]

Decryption 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.

### 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](https://docs.spring.io/spring-cloud-project/docs/current-SNAPSHOT/spring-cloud-config/docs/configuration-repository.html#_composite_environments). For example:

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

### 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
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.

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.

7.1. Spring Boot Config Data Import

Spring Boot 2.4 introduced a new way to import configuration data via the spring.config.import property. This is now the default way to bind to Config Server.

To optionally connect to config server set the following in application.properties:

application.properties

```
spring.config.import=optional:configserver:
```

This will connect to the Config Server at the default location of "http://localhost:8888". Removing the optional: prefix will cause the Config Client to fail if it is unable to connect to Config Server. To change the location of Config Server either set spring.cloud.config.uri or add the url to the spring.config.import statement such as, spring.config.import=optional:configserver:http://myhost:8888. The location in the import property has precedence over the uri property.

A bootstrap file (properties or yaml) is not needed for the Spring Boot Config Data method of import via spring.config.import.

7.2. Config First Bootstrap

To use the legacy bootstrap way of connecting to Config Server, bootstrap must be enabled via a property or the spring-cloud-starter-bootstrap starter. The property is spring.cloud.bootstrap.enabled=true. It must be set as a System Property or environment variable. Once bootstrap has been enabled any application with Spring Cloud Config Client on the classpath will connect to Config Server 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”).

### 7.2.1. Discovery First Lookup

Unless you are using config first bootstrap, you will need to have a `spring.config.import` property in your configuration properties. For example, `spring.config.import=optional:configserver:`.

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.

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`). 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:

```yaml
eureka:
  instance:
    ...  
  metadataMap:
    user: osufhalskjrtl
    password: lviuhlszvaorhvlo5847
    configPath: /config
```

### 7.2.2. Discovery First Bootstrap Using Eureka And WebClient

If you use the Eureka DiscoveryClient from Spring Cloud Netflix and also want to use WebClient instead of Jersey or RestTemplate, you need to include WebClient on your classpath as well as set `eureka.client.webclient.enabled=true`.

### 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.

To get similar functionality using `spring.config.import`, simply omit the optional prefix.

### 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.

### 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:

- "application" = `${spring.application.name}`
- "profile" = `${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`).

### 7.6. Specifying MultipleUrls 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.

### 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`.

### 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:

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

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

```properties
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`:

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

If config server requires client side TLS certificate, you can configure client side TLS certificate and trust store via properties, as shown in following example:

```yaml
spring:
  cloud:
    config:
      uri: https://myconfig.myconfig.com
      tls:
        enabled: true
        key-store: <path-of-key-store>
        key-store-type: PKCS12
        key-store-password: <key-store-password>
        key-password: <key-password>
        trust-store: <path-of-trust-store>
        trust-store-type: PKCS12
        trust-store-password: <trust-store-password>
```

The `spring.cloud.config.tls.enabled` needs to be true to enable config client side TLS. When `spring.cloud.config.tls.trust-store` is omitted, a JVM default trust store is used. The default value for `spring.cloud.config.tls.key-store-type` and `spring.cloud.config.tls.trust-store-type` is PKCS12. When password properties are omitted, empty password is assumed.

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).

### 7.8.1. 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).

### 7.8.2. 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
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

```

### 7.8.3. 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:

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

### 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:

```
@Configuration
public class CustomConfigServiceBootstrapConfiguration {
  @Bean
  public ConfigServicePropertySourceLocator configServicePropertySourceLocator() {
    ConfigClientProperties clientProperties = configClientProperties();
    ConfigServicePropertySourceLocator configServicePropertySourceLocator = new
    ConfigServicePropertySourceLocator(clientProperties);
    configServicePropertySourceLocator.setRestTemplate(customRestTemplate(clientProperties));
    return configServicePropertySourceLocator;
  }
}
```
Spring Cloud Consul

2020.0.3-SNAPSHOT

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 and Client Side Load Balancing, Circuit Breaker are provided by integration with other Spring Cloud projects.

1. Quick Start

This quick start walks through using Spring Cloud Consul for Service Discovery and Distributed Configuration.

First, run Consul Agent on your machine. Then you can access it and use it as a Service Registry and Configuration source with Spring Cloud Consul.

1.1. Discovery Client Usage

To use these features in an application, you can build it as a Spring Boot application that depends on spring-cloud-consul-core. The most convenient way to add the dependency is with a Spring Boot starter: org.springframework.cloud:spring-cloud-starter-consul-discovery. We recommend using dependency management and spring-boot-starter-parent. The following example shows a typical Maven configuration:

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

The preceding code would sets the value of the name variable to appAsecret.
The following example shows a typical Gradle setup:
Now you can create a standard Spring Boot application, such as the following HTTP server:

```java
@SpringBootApplication
@RestController
public class Application {

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

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

When this HTTP server runs, it connects to Consul Agent running at the default local 8500 port. To modify the startup behavior, you can change the location of Consul Agent by using `application.properties`, as shown in the following example:
You can now use `DiscoveryClient`, `@LoadBalanced RestTemplate`, or `@LoadBalanced WebClient.Builder` to retrieve services and instances data from Consul, as shown in the following example:

```java
@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;
}
```

### 1.2. Distributed Configuration Usage

To use these features in an application, you can build it as a Spring Boot application that depends on `spring-cloud-consul-core` and `spring-cloud-consul-config`. The most convenient way to add the dependency is with a Spring Boot starter: `org.springframework.cloud:spring-cloud-starter-consul-config`. We recommend using dependency management and `spring-boot-starter-parent`. The following example shows a typical Maven configuration:
The following example shows a typical Gradle setup:
Now you can create a standard Spring Boot application, such as the following HTTP server:

```java
@SpringBootApplication
@RestController
public class Application {

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

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

The application retrieves configuration data from Consul.

If you use Spring Cloud Consul Config, you need to set the `spring.config.import` property in order to bind to Consul. You can read more about it in the Spring Boot Config Data Import section.
2. Install Consul

Please see the installation documentation for instructions on how to install Consul.

3. 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

4. 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.

4.1. 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.

4.2. 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 /actuator/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 SpringApplication(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

```yaml
spring:
  cloud:
    consul:
      host: localhost
      port: 8500
```

If you use Spring Cloud Consul Config, and you have set spring.cloud.bootstrap.enabled=true or spring.config.use-legacy-processing=true or use spring-cloud-starter-bootstrap, then 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.

4.2.1. 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:
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:

```yaml
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:

```java
/** 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
```

4.2.2. HTTP Health Check

The health check for a Consul instance defaults to "/actuator/health", which is the default location of the health endpoint in a Spring Boot Actuator application. You need to change this, 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.

This example illustrates the above (see the `spring.cloud.consul.discovery.health-check-*` properties in the appendix page for more options).

`application.yml`

```yml
spring:
  cloud:
    consul:
      discovery:
        healthCheckPath: ${management.server.servlet.context-path}/actuator/health
        healthCheckInterval: 15s
```

You can disable the HTTP health check entirely by setting `spring.cloud.consul.discovery.register-health-check=false`.

Applying Headers

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:
According to the HTTP standard, each header can have more than one values, in which case, an array can be supplied:

```
spring:
  cloud:
    consul:
      discovery:
        health-check-headers:
          X-Config-Token: 6442e58b-d1ea-182e-cfa5-cf9cddef0722
```

### 4.2.3. Actuator Health Indicator(s)

If the service instance is a Spring Boot Actuator application, it may be provided the following Actuator health indicators.

**DiscoveryClientHealthIndicator**

When Consul Service Discovery is active, a [DiscoveryClientHealthIndicator](https://consul.dev) is configured and made available to the Actuator health endpoint. See [here](https://github.com/spring-cloud/spring-cloud-consul) for configuration options.

**ConsulHealthIndicator**

An indicator is configured that verifies the health of the [ConsulClient](https://consul.dev).

By default, it retrieves the Consul leader node status and all registered services. In deployments that have many registered services it may be costly to retrieve all services on every health check. To skip the service retrieval and only check the leader node status set `spring.cloud.consul.health-indicator.include-services-query=false`.

To disable the indicator set `management.health.consul.enabled=false`.

> When the application runs in [bootstrap context mode](https://spring.io/guides/gs/bootstrap-context/) (the default), this indicator is loaded into the bootstrap context and is not made available to the Actuator health endpoint.
4.2.4. 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>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>'group'</td>
<td>Property <code>spring.cloud.consul.discovery.instance-group</code>. This values is only generated if <code>instance-group</code> is not empty.</td>
</tr>
<tr>
<td>'secure'</td>
<td>True if property <code>spring.cloud.consul.discovery.scheme</code> equals 'https', otherwise false.</td>
</tr>
<tr>
<td>Property <code>spring.cloud.consul.discovery.default-zone-metadata-name</code>, defaults to 'zone'</td>
<td>Property <code>spring.cloud.consul.discovery.instance-zone</code>. This values is only generated if <code>instance-zone</code> is not empty.</td>
</tr>
</tbody>
</table>

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.

4.2.5. 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.

### 4.3. Looking up services

#### 4.3.1. 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 Spring Cloud LoadBalancer 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 now also offers support for Spring Cloud LoadBalancer.

### 4.3.2. 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.

```java
@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;
}
```

### 4.4. 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.

### 5. 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 mimics 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.

### 5.1. 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.

### 5.2. Spring Boot Config Data Import

Spring Boot 2.4 introduced a new way to import configuration data via the `spring.config.import` property. This is now the default way to get configuration from Consul.

To optionally connect to Consul set the following in `application.properties`:

```
spring.config.import=optional:consul:
```

This will connect to the Consul Agent at the default location of "http://localhost:8500". Removing the `optional:` prefix will cause Consul Config to fail if it is unable to connect to Consul. To change the connection properties of Consul Config either set `spring.cloud.consul.host` and `spring.cloud.consul.port` or add the host/port pair to the `spring.config.import` statement such as, `spring.config.import=optional:consul:myhost:8500`. The location in the import property has precedence over the host and port propertie.

Consul Config will try to load values from four automatic contexts based on `spring.cloud.consul.config.name` (which defaults to the value of the `spring.application.name` property) and `spring.cloud.consul.config.default-context` (which defaults to `application`). If you want to specify the contexts rather than using the computed ones, you can add that information to the `spring.config.import` statement.
application.properties

```
spring.config.import=optional:consul:myhost:8500/contextone;/context/two
```

This will optionally load configuration only from `/contextone` and `/context/two`.

A bootstrap file (properties or yaml) is not needed for the Spring Boot Config Data method of import via `spring.config.import`.

### 5.3. Customizing

Consul Config may be customized using the following properties:

```
spring:
  cloud:
    consul:
      config:
        enabled: true
        prefix: configuration
        defaultContext: apps
        profileSeparator: '::'
```

If you have set `spring.cloud.bootstrap.enabled=true` or `spring.config.use-legacy-processing=true`, or included `spring-cloud-starter-bootstrap`, then the above values will need to be placed in `bootstrap.yml` instead of `application.yml`.

- `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

### 5.4. 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.

5.5. 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:

```yaml
spring:
  cloud:
    consul:
      config:
        format: YAML
```

If you have set spring.cloud.bootstrap.enabled=true or spring.config.use-legacy-processing=true, or included spring-cloud-starter-bootstrap, then the above values will need to be placed in bootstrap.yml instead of application.yml.

YAML must be set in the appropriate data key in consul. Using the defaults above the keys would look like:

```text
config/testApp,dev/data
cfg/testApp/data
cfg/application,dev/data
cfg/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.

5.6. 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 .yml and .properties respectively. Set the spring.cloud.consul.config.format property to FILES. For example:
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.

### 5.7. 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.fail-fast=false` will cause the configuration module to log a warning rather than throw an exception. This will allow the application to continue startup normally.

If you have set `spring.cloud.bootstrap.enabled=true` or `spring.config.use-legacy-processing=true`, or included `spring-cloud-starter-bootstrap`, then the above values will need to be placed in `bootstrap.yml` instead of `application.yml`.

### 6. 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.
7. Spring Cloud Bus with Consul

7.1. 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 how to send custom messages.

8. Circuit Breaker with Hystrix

Applications can use the Hystrix Circuit Breaker provided by the Spring Cloud Netflix project by including this starter in the project's 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. 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:

```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.

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.
application.yml

```yaml
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

```java
@EnableTurbine
@SpringBootApplication
public class Turbine {
  public static void main(String[] args) {
    SpringApplication.run(DemoturbinecommonsApplication.class, args);
  }
}
```

10. Configuration Properties

To see the list of all Consul related configuration properties please check the Appendix page.

Spring Cloud Contract Reference Documentation

Adam Dudczak, Mathias Düsterhöft, Marcin Grzejszczak, Dennis Kieselhorst, Jakub Kubryński, Karol Lassak, Olga Maciaszek-Sharma, Mariusz Smykula, Dave Syer, Jay Bryant

The reference documentation consists of the following sections:

- **Legal**
  - Legal information.

- **Documentation Overview**
  - About the Documentation, Getting Help, First Steps, and more.

- **Getting Started**
  - Introducing Spring Cloud Contract, Developing Your First Spring Cloud Contract-based Application

- **Using Spring Cloud Contract**
  - Spring Cloud Contract usage examples and workflows.
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, Google Cloud Functions, 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](https://github.com).
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 `\J` in a terminal to insert a new line in a literal string like that.)

3. Programming model

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.
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\(\langle I, O \rangle\)
- Consumer\(\langle I \rangle\)

### 3.2.1. Supplier

Supplier can be reactive - Supplier\(\langle Flux\langle T \rangle \rangle\) or imperative - Supplier\(\langle T \rangle\). 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:

```java
@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);
    };
}
```

### 3.2.2. 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.
3.2.3. 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.

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.

3.3.1. 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 `,` instead of pipe (such as `...definition=uppercase,reverse`).

3.3.2. 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.

3.4. Function Routing and Filtering

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:

```java
public class RoutingFunction implements Function<Object, Object> {
    . . .
}
```

The routing instructions could be communicated in several ways. We support providing instructions via Message headers, System properties as well as pluggable strategy. So let's look at some of the details

**MessageRoutingCallback**

The `MessageRoutingCallback` is a strategy to assist with determining the name of the route-to function definition.

```java
public interface MessageRoutingCallback {

    /**
     * Determines the name of the function definition to route incoming `Message`.
     * @param message instance of incoming `Message`
     * @return the name of the route-to function definition
     */
    String functionDefinition(Message<?> message);
}
```

All you need to do is implement it and register it as a bean. The framework will automatically pick it up and use it for routing decisions. For example

```java
@Bean
public MessageRoutingCallback customRouter() {
    return new MessageRoutingCallback() {
        @Override
        public String functionDefinition(Message<?> message) {
            return (String) message.getHeaders().get("func_name");
        }
    };
}
```

In the preceding example you can see a very simple implementation of `MessageRoutingCallback` which determines the function definition from `func_name` header of the incoming `Message`.

**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`).

It is important to understand that providing `spring.cloud.function.definition` or `spring.cloud.function.routing-expression` as Message headers will only work for imperative functions (e.g., `Function<Foo, Bar>`). That is to say that we can only route **per-message** with imperative functions. With reactive functions we can not route **per-message**. Therefore you can only provide your routing instructions as Application Properties. It's all about unit-of-work. In imperative function unit of work is Message so we can route based on such unit-of-work. With reactive function unit-of-work is the entire stream, so we'll act only on the instruction provided via application properties and route the entire stream.

**Order of priority for routing instructions**

Given that we have several mechanisms of providing routing instructions it is important to understand the priorities for conflict resolutions in the event multiple mechanisms are used at the same time, so here is the order:

1. **MessageRoutingCallback** (If function is imperative will take over regardless if anything else is defined)
2. Message Headers (If function is imperative and no **MessageRoutingCallback** provided)
3. Application Properties (Any function)

**Function Filtering** Filtering is the type of routing where there are only two paths - 'go' or 'discard'. In terms of functions it mean you only want to invoke a certain function if some condition returns 'true', otherwise you want to discard input. However, when it comes to discarding input there are many interpretation of what it could mean in the context of your application. For example, you may
want to log it, or you may want to maintain the counter of discarded messages. you may also want to do nothing at all. Because of these different paths, we do not provide a general configuration option for how to deal with discarded messages. Instead we simply recommend to define a simple Consumer which would signify the 'discard' path:

```java
@Bean
public Consumer<?> devNull() {
    // log, count or whatever
}
```

Now you can have routing expression that really only has two paths effectively becoming a filter. For example:

```bash
--spring.cloud.function.routing
-expression=headers.contentType.toString().equals('text/plain') ? 'echo' : 'devNull'
```

Every message that does not fit criteria to go to 'echo' function will go to 'devNull' where you can simply do nothing with it. The signature `Consumer<?>` will also ensure that no type conversion will be attempted resulting in almost no execution overhead.

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 cannot access and/or rely on the routing instructions communicated via individual values (e.g., Message).

### 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),

```java
@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.

### 3.6. Type conversion (Content-Type negotiation)

Content-Type negotiation is one of the core features of Spring Cloud Function as it allows to not only transform the incoming data to the types declared by the function signature, but to do the same transformation during function composition making otherwise un-composable (by type) functions composable.

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 function as an example:

```java
@Bean
public Function<Person, String> personFunction {..}
```

The function shown in the preceding example expects a `Person` object as an argument and produces a `String` type as an output. If such function is invoked with the type `Person`, than all works fine. But typically function plays a role of a handler for the incoming data which most often comes in the raw format such as `byte[]`, `JSON String` etc. In order for the framework to succeed in passing the incoming data as an argument to this function, it has to somehow transform the incoming data to a `Person` type.

Spring Cloud Function relies on two native to Spring mechanisms to accomplish that.

1. *MessageConverter* - to convert from incoming Message data to a type declared by the function.
2. *ConversionService* - to convert from incoming non-Message data to a type declared by the function.

This means that depending on the type of the raw data (Message or non-Message) Spring Cloud Function will apply one or the other mechanisms.

For most cases when dealing with functions that are invoked as part of some other request (e.g., HTTP, Messaging etc) the framework relies on *MessageConverters*, since such requests already converted to Spring `Message`. In other words, the framework locates and applies 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 function 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` header.

Such header usually comes as part of the Message where it is injected by the corresponding adapter
that created such Message in the first place. For example, HTTP POST request will have its content-
type HTTP header copied to contentType header of the Message.

For cases when such header does not exist framework relies on the default content type as application/json.

### 3.6.1. 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 which trigger right before the invocation of the user-defined function (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.

The combination of contentType and argument type is the mechanism by which framework determines if message can be converted to a target type by locating the appropriate MessageConverter. If no appropriate MessageConverter is found, an exception is thrown, which you can handle by adding a custom MessageConverter (see User-defined Message Converters).

---

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. It is a hint, which MessageConverter may or may not take into consideration.

### 3.6.2. 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.

### 3.6.3. 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. **JsonMessageConverter**: Supports conversion of the payload of the Message to/from POJO for cases when contentType is application/json using Jackson or Gson libraries (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. **StringMessageConverter**: Supports conversion of any type to a `String` when `contentType` is `text/plain`.

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**.

### 3.6.4. User-defined Message Converters

Spring Cloud Function 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 MimeType("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));
    }
}

3.6.5. Note on JSON options

In Spring Cloud Function we support Jackson and Gson mechanisms to deal with JSON. And for your benefit have abstracted it under org.springframework.cloud.function.json.JsonMapper which itself is aware of two mechanisms and will use the one selected by you or following the default rule. The default rules are as follows:

- Whichever library is on the classpath that is the mechanism that is going to be used. So if you have com.fasterxml.jackson.* to the classpath, Jackson is going to be used and if you have com.google.code.gson, then Gson will be used.

- If you have both, then Gson will be the default, or you can set spring.cloud.function.preferred-json-mapper property with either of two values: gson or jackson.

That said, the type conversion is usually transparent to the developer, however given that org.springframework.cloud.function.json.JsonMapper is also registered as a bean you can easily inject it into your code if needed.
3.7. Kotlin Lambda support

We also provide support for Kotlin lambdas (since v2.0). Consider the following:

```kotlin
@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 Kotlin SDK libraries on the classpath which will trigger appropriate autoconfiguration and supporting classes.

3.8. 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.

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
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.

### 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.

### 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`).

### 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 it’s integration with Spring Cloud Function. Please refer to `Spring Cloud Function` section of the Spring Cloud Stream reference manual for more details and examples.
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.

```xml
<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:

```java
@SpringBootApplication
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"));
  }
}
```

And here is the example using Maven URI (taken from one of the tests in `FunctionDeployerTests`):
@SpringBootApplication
public class DeployFunctionDemo {

    public static void main(String[] args) {
        String[] args = new String[]{
            "--spring.cloud.function.location=maven://oz.demo:demo-
            uppercase:0.0.1-SNAPSHOT",
            "--spring.cloud.function.function-class=oz.demo.uppercase.MyFunction"
        };

        ApplicationContext context = SpringApplication.run(DeployerApplication.class, args);
        FunctionCatalog catalog = context.getBean(FunctionCatalog.class);
        Function<String, String> function = catalog.lookup("myFunction");

        assertThat(function.apply("bob")).isEqualTo("BOB");
    }
}

Keep in mind that Maven resource such as local and remote repositories, user, password and more are resolved using default MavenProperties which effectively use local defaults and will work for majority of cases. However if you need to customize you can simply provide a bean of type MavenProperties where you can set additional properties (see example below).

@Bean
public MavenProperties mavenProperties() {
    MavenProperties properties = new MavenProperties();
    properties.setLocalRepository("target/it/");
    return properties;
}

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.

6.1.1. 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`.

### 6.1.2. 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):

```java
package function.example;

public class UpperCaseFunction implements Function<String, String> {
    @Override
    public String apply(String value) {
        return value.toUpperCase();
    }
}
```
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:

```bash
--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.

### 6.1.3. 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.
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 it's classpath. From the deployer perspective this doesn't matter.

### 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.

#### 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:

```java
@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));
}

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.

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:

```java
@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:

```java
@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
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");
    }
}

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:
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):
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 is an adapter for Spring Cloud Function jars.

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.

10.1.1. 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 or 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 asked 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/service-function-adapter/master/documents/aws Lambda Functions Readme.md](https://raw.githubusercontent.com/spring-cloud/spring-cloud/service-function-adapter/master/documents/aws Lambda Functions Readme.md)
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.

10.1.2. 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.FunctionInvoker` which is the implementation of AWS’s RequestStreamHandler. User doesn’t need to do anything other then specify it as ‘handler’ on AWS dashboard when deploying function. It will handle most of the case including Kinesis, streaming etc.

If your app has more than one `@Bean` of type `Function` etc. then you can choose the one to use by configuring `spring.cloud.function.definition` property or environment variable. The functions are extracted from the Spring Cloud FunctionCatalog. In the event you don’t specify `spring.cloud.function.definition` the framework will attempt to find a default following the search order where it searches first for `Function` then `Consumer` and finally `Supplier`.

10.1.3. 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>

10.1.4. 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:

```xml
<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.

```xml
<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:

```groovy
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:
You can use the Spring Boot Gradle Plugin and Spring Boot Thin Gradle Plugin to generate the thin jar.

You can find the entire sample build.gradle file for deploying Spring Cloud Function applications to AWS Lambda with Gradle here.
10.1.5. 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).

```bash
./mvnw -U clean package
```

Using the AWS command line tools it looks like this:

```bash
```

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:

```json
{
   "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.

10.1.6. 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.
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 org.springframework.cloud.function.adapter.azure.FunctionInvoker 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:

```java
public class FooHandler extends FunctionInvoker<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 a 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.

10.2.1. 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 we propagate `ExecutionContext` as Message header under `executionContext` name, so all you need is access it is have your function accept a Message and access this header.

Spring Cloud Function will register `ExecutionContext` as bean in the Application context, so it could be injected into your function. For example

```java
@Bean
public Function<Message<Foo>, Bar> uppercase() {
    return message -> {
        ExecutionContext targetContext = message.getHeaders().get("executionContext");
        targetContext.getLogger().info("Invoking 'uppercase' on " + foo.getValue());
        return new Bar(message.getPayload().getValue().toUpperCase());
    };
}
```

With Message you will also have access to additional Azure meta information as Message headers that come as part of your request.

### 10.2.2. 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.

### 10.2.3. 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:

```xml
<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.
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.

10.2.4. Build

```
./mvnw -U clean package
```

10.2.5. 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:
```bash
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:

```json
{
    "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.

10.3. Google Cloud Functions

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.

10.3.1. Project Dependencies

Start by adding the spring-cloud-function-adapter-gcp dependency to your project.

```xml
<dependencies>
    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-function-adapter-gcp</artifactId>
    </dependency>
    ...
</dependencies>
```

In addition, add the spring-boot-maven-plugin which will build the JAR of the function to deploy.

Notice that we also reference spring-cloud-function-adapter-gcp as a dependency of the spring-boot-maven-plugin. This is necessary because it modifies the plugin to package your function in the correct JAR format for deployment on Google Cloud Functions.
Finally, add the Maven plugin provided as part of the Google Functions Framework for Java. This allows you to test your functions locally via `mvn function:run`.

The function target should always be set to `org.springframework.cloud.function.adapter.gcp.GcfJarLauncher`; this is an adapter class which acts as the entry point to your Spring Cloud Function from the Google Cloud Functions platform.

A full example of a working `pom.xml` can be found in the Spring Cloud Functions GCP sample.

### 10.3.2. HTTP Functions

Google Cloud Functions supports deploying HTTP Functions, which are functions that are invoked by HTTP request. The sections below describe instructions for deploying a Spring Cloud Function as an HTTP Function.

#### 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();
    }
}

Specify your configuration main class in resources/META-INF/MANIFEST.MF.

Main-Class: com.example.CloudFunctionMain

Then run the function locally. This is provided by the Google Cloud Functions function-maven-plugin described in the project dependencies section.

mvn function:run

Invoke the HTTP function:

curl http://localhost:8080/ -d "hello"

Deploy to GCP

Start by packaging your application.

mvn package

If you added the custom spring-boot-maven-plugin plugin defined above, you should see the resulting JAR in target/deploy directory. This JAR is correctly formatted for deployment to Google Cloud Functions.

Next, make sure that you have the Cloud SDK CLI installed.

From the project base directory run the following command to deploy.
Invoke the HTTP function:

curl https://REGION-PROJECT_ID.cloudfunctions.net/function-sample-gcp-http -d "hello"

### 10.3.3. Background Functions

Google Cloud Functions also supports deploying **Background Functions** which are invoked indirectly in response to an event, such as a message on a **Cloud Pub/Sub** topic, a change in a **Cloud Storage** bucket, or a **Firebase** event.

The **spring-cloud-function-adapter-gcp** allows for functions to be deployed as background functions as well.

The sections below describe the process for writing a Cloud Pub/Sub topic background function. However, there are a number of different event types that can trigger a background function to execute which are not discussed here; these are described in the **Background Function triggers documentation**.

**Getting Started**

Let's start with a simple Spring Cloud Function which will run as a GCF background function:

```java
@SpringBootApplication
public class BackgroundFunctionMain {

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

    @Bean
    public Consumer<PubSubMessage> pubSubFunction() {
        return message -> System.out.println("The Pub/Sub message data: " + message.getData());
    }
}
```

In addition, create **PubSubMessage** class in the project with the below definition. This class represents the **Pub/Sub event structure** which gets passed to your function on a Pub/Sub topic event.
public class PubSubMessage {

    private String data;
    private Map<String, String> attributes;
    private String messageId;
    private String publishTime;

    public String getData() {
        return data;
    }

    public void setData(String data) {
        this.data = data;
    }

    public Map<String, String> getAttributes() {
        return attributes;
    }

    public void setAttributes(Map<String, String> attributes) {
        this.attributes = attributes;
    }

    public String getMessageId() {
        return messageId;
    }

    public void setMessageId(String messageId) {
        this.messageId = messageId;
    }

    public String getPublishTime() {
        return publishTime;
    }

    public void setPublishTime(String publishTime) {
        this.publishTime = publishTime;
    }

}

Specify your configuration main class in resources/META-INF/MANIFEST.MF.

Main-Class: com.example.BackgroundFunctionMain
Then run the function locally. This is provided by the Google Cloud Functions `function-maven-plugin` described in the project dependencies section.

```
mvn function:run
```

Invoke the HTTP function:

```
curl localhost:8080 -H "Content-Type: application/json" -d '{"data":"hello"}'
```

Verify that the function was invoked by viewing the logs.

**Deploy to GCP**

In order to deploy your background function to GCP, first package your application.

```
mvn package
```

If you added the custom `spring-boot-maven-plugin` plugin defined above, you should see the resulting JAR in `target/deploy` directory. This JAR is correctly formatted for deployment to Google Cloud Functions.

Next, make sure that you have the Cloud SDK CLI installed.

From the project base directory run the following command to deploy.

```
gcloud functions deploy function-sample-gcp-background \
  --entry-point org.springframework.cloud.function.adapter.gcp.GcfJarLauncher \
  --runtime java11 \
  --trigger-topic my-functions-topic \
  --source target/deploy \
  --memory 512MB
```

Google Cloud Function will now invoke the function every time a message is published to the topic specified by `--trigger-topic`.

For a walkthrough on testing and verifying your background function, see the instructions for running the GCF Background Function sample.

**10.3.4. Sample Functions**

The project provides the following sample functions as reference:

- The `function-sample-gcp-http` is an HTTP Function which you can test locally and try deploying.
- The `function-sample-gcp-background` shows an example of a background function that is triggered by a message being published to a specified Pub/Sub topic.
Spring Cloud Gateway

2020.0.3-SNAPSHOT

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.

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.

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 GatewayFilter that have been constructed with a specific factory. Here, you can modify requests and responses before or after sending the downstream request.

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.

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 each section. The arguments are typically listed in the order that would be needed for the shortcut configuration.

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.

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.
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.

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:

```yaml
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).

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 6. application.yml

```yaml
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).

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 7. application.yml

```yaml
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.

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 8. application.yml

```yaml
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.

### 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 9. application.yml

```yaml
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).

### 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 10. 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`.

### 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 11. 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`.

### 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 `matchTrailingSlash` (defaults to `true`). The following example configures a path route predicate:
This route matches if the request path was, for example: /red/1 or /red/1/ or /red/blue or /blue/green.

If `matchTrailingSlash` is set to `false`, then request path /red/1/ will not be matched.

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:

```java
Map<String, String> uriVariables =
ServerWebExchangeUtils.getPathPredicateVariables(exchange);

String segment = uriVariables.get("segment");
```

### 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:
The preceding route matches if the request contained a *green* query parameter.

The preceding route matches if the request contained a *red* query parameter whose value matched the *gree.* regexp, so *green* and *greet* would match.

### 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 14. 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*. 
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 15. application.yml*

```yaml
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 `weighlow.org`

5.11.1. 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:
The following `maxTrustedIndex` values yield the following remote addresses:

<table>
<thead>
<tr>
<th>maxTrustedIndex</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>[Integer.MIN_VALUE,0]</td>
<td>(invalid, IllegalArgumentException during initialization)</td>
</tr>
<tr>
<td>1</td>
<td>0.0.0.3</td>
</tr>
<tr>
<td>2</td>
<td>0.0.0.2</td>
</tr>
<tr>
<td>3</td>
<td>0.0.0.1</td>
</tr>
<tr>
<td>[4, Integer.MAX_VALUE]</td>
<td>0.0.0.1</td>
</tr>
</tbody>
</table>

The following example shows how to achieve the same configuration with Java:

**Example 16. GatewayConfig.java**

```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")
    )
```

### 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.
6.1. The **AddRequestHeader GatewayFilter Factory**

The *AddRequestHeader GatewayFilter* factory takes a *name* and *value* parameter. The following example configures an *AddRequestHeader GatewayFilter*:

**Example 17. application.yml**

```yaml
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 18. application.yml**

```yaml
spring:
  cloud:
    gateway:
      routes:
      - id: add_request_header_route
        uri: https://example.org
        predicates:
          - Path=/red/{segment}
        filters:
          - AddRequestHeader=X-Request-Red, Blue-{segment}
```

6.2. The **AddRequestParameter GatewayFilter Factory**

The *AddRequestParameter GatewayFilter* Factory takes a *name* and *value* parameter. The following example configures an *AddRequestParameter GatewayFilter*:
Example 19. application.yml

```yaml
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 20. application.yml

```yaml
spring:
  cloud:
    gateway:
      routes:
        - id: add_request_parameter_route
          uri: https://example.org
          predicates:
            - Host: {segment}.myhost.org
          filters:
            - AddRequestParameter=foo, bar-{segment}
```

6.3. The `AddResponseHeader GatewayFilter Factory`

The `AddResponseHeader GatewayFilter` Factory takes a `name` and `value` parameter. The following example configures an `AddResponseHeader GatewayFilter`:
Example 21. application.yml

```yaml
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 22. application.yml

```yaml
spring:
  cloud:
    gateway:
      routes:
        - id: add_response_header_route
          uri: https://example.org
          predicates:
            - Host: {segment}.myhost.org
          filters:
            - AddResponseHeader=foo, bar-{segment}
```

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 23. application.yml

```yaml
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`.

### 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 24. application.yml

```yaml
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 25. application.yml**

```yaml
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 26. Application.java**

```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 LoadBalancer 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:
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.

6.5.1. Tripping The Circuit Breaker On Status Codes

In some cases you might want to trip a circuit breaker based on the status code returned from the route it wraps. The circuit breaker config object takes a list of status codes that if returned will cause the circuit breaker to be tripped. When setting the status codes you want to trip the circuit breaker you can either use an integer with the status code value or the String representation of the HttpStatus enumeration.
Example 28. application.yml

```yaml
spring:
  cloud:
    gateway:
      routes:
        - id: circuitbreaker_route
          uri: lb://backing-service:8088
          predicates:
            - Path=/consumingServiceEndpoint
          filters:
            - name: CircuitBreaker
              args:
                name: myCircuitBreaker
                fallbackUri: forward:/inCaseOfFailureUseThis
                statusCodes:
                  - 500
                  - "NOT_FOUND"
```

Example 29. Application.java

```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").addStatusCode("INTERNAL_SERVER_ERROR")
      ).rewritePath("/consumingServiceEndpoint",
        "/backingServiceEndpoint").uri("lb://backing-service:8088")
    ).build();
}
```

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 30. application.yml

```yaml
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.

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
The following example configures a `MapRequestHeader`:

**Example 31. application.yml**

```yaml
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.

### 6.8. The **PrefixPath** GatewayFilter Factory

The **PrefixPath** GatewayFilter factory takes a single `prefix` parameter. The following example configures a **PrefixPath** GatewayFilter:

**Example 32. application.yml**

```yaml
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`.

### 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 33. application.yml

```yaml
spring:
  cloud:
    gateway:
      routes:
        - id: preserve_host_route
          uri: https://example.org
          filters:
            - PreserveHostHeader
```

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 34. KeyResolver.java

```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*:

```
# INVALID SHORTCUT CONFIGURATION
spring.cloud.gateway.routes[0].filters[0]=RequestRateLimiter=2, 2, #{@userkeyresolver}
```

### 6.10.1. 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 bellow `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.
The following example configures a KeyResolver in Java:

**Example 37. Config.java**

```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 38. application.yml

```yaml
spring:
  cloud:
    gateway:
      routes:
        - id: requestratelimiter_route
          uri: https://example.org
          filters:
            - name: RequestRateLimiter
              args:
                rate-limiter: "#{@myRateLimiter}"
                key-resolver: "#{@userKeyResolver}"
```

### 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 39. application.yml

```yaml
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.

### 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 40. application.yml

```yaml
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.

### 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 41. application.yml

```yaml
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.

### 6.14. The RemoveRequestParameter GatewayFilter Factory

The `RemoveRequestParameter GatewayFilter` factory takes a `name` parameter. It is the name of the query parameter to be removed. The following example configures a `RemoveRequestParameter GatewayFilter`:
Example 42. application.yml

```yaml
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.

6.15. The **RewritePath GatewayFilter Factory**

The **RewritePath GatewayFilter** factory takes a path `regexp` 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 43. application.yml

```yaml
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.

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**:
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` (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`.

### 6.17. The **RewriteResponseHeader GatewayFilter Factory**

The `RewriteResponseHeader GatewayFilter` factory takes `name`, `regexp`, 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 45. application.yml**

```yaml
spring:
  cloud:
    gateway:
      routes:
      - id: rewriteresponseheader_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.

### 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 46. application.yml**

```yaml
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.

### 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
- **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.

### 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 47. 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.
6.21. The **SetRequestHeader GatewayFilter Factory**

The **SetRequestHeader GatewayFilter** factory takes name and value parameters. The following listing configures a **SetRequestHeader GatewayFilter**:

```yaml
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:

```yaml
spring:
  cloud:
    gateway:
      routes:
        - id: setrequestheader_route
          uri: https://example.org
          predicates:
            - Host: {segment}.myhost.org
          filters:
            - SetRequestHeader=f00, bar-{segment}
```

6.22. The **SetResponseHeader GatewayFilter Factory**

The **SetResponseHeader GatewayFilter** factory takes name and value parameters. The following listing configures a **SetResponseHeader GatewayFilter**:

```yaml
```
Example 50. `application.yml`

```yaml
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 51. `application.yml`

```yaml
spring:
  cloud:
    gateway:
      routes:
        - id: setresponseheader_route
          uri: https://example.org
          predicates:
            - Host: {segment}.myhost.org
          filters:
            - SetResponseHeader=foo, bar-{segment}
```

### 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:
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 53. application.yml

```
spring:
  cloud:
    gateway:
      set-status:
        original-status-header-name: original-http-status
```

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:
When a request is made through the gateway to `/name/blue/red`, the request made to `nameservice` looks like `nameservice/red`.

### 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`.
- **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 `firstBackoff * (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 `prevBackoff * 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**:
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`.

### 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`:
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`:

```yml
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.

### 6.27. The `SetRequestHostHeader GatewayFilter` Factory

There are certain situations when the host header may need to be overridden. In this situation, the `SetRequestHostHeader GatewayFilter` factory can replace the existing host header with a specified value. The filter takes a `host` parameter. The following listing configures a `SetRequestHostHeader GatewayFilter`.

```java
setRequestHostHeader:
  host: "example.com"
```
Example 57. application.yml

```yaml
spring:
  cloud:
    gateway:
      routes:
        - id: set_request_host_header_route
          uri: http://localhost:8080/headers
          predicates:
            - Path=/headers
          filters:
            - name: SetRequestHostHeader
              args:
                host: example.org
```

The `SetRequestHostHeader` GatewayFilter factory replaces the value of the host header with `example.org`.

### 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.

ℹ️ This filter can be configured only by using the Java DSL.

The following listing shows how to modify a request body GatewayFilter:
```java
@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;
    }
}
```

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`:

If the request has no body, the *RewriteFilter* will be passed `null.Mono.empty()` should be returned to assign a missing body in the request.
6.30. **Token Relay** GatewayFilter Factory

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.

Spring Cloud Gateway can forward OAuth2 access tokens downstream to the services it is proxying. To add this functionality to gateway you need to add the `TokenRelayGatewayFilterFactory` like this:

**App.java**

```java
@Bean
public RouteLocator customRouteLocator(RouteLocatorBuilder builder) {
    return builder.routes()
        .route("resource", r -> r.path("/resource")
            .filters(f -> f.tokenRelay())
            .uri("http://localhost:9000")
        ).build();
}
```

or this

```java
@Bean
public RouteLocator customRouteLocator(RouteLocatorBuilder builder) {
    return builder.routes()
        .route("resource", r -> r.path("/resource")
            .filters(f -> f.tokenRelay())
            .uri("http://localhost:9000")
        ).build();
}
```
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`

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.

A `TokenRelayGatewayFilterFactory` bean will only be created if the proper `spring.security.oauth2.client.*` properties are set which will trigger creation of a `ReactiveClientRegistrationRepository` bean.

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.

### 6.31. 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 58. application.yml*
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.

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:

```java
@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;
    }
}
```

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.

7.3. 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:

Example 60. application.yml

```yaml
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.

Gateway supports all the LoadBalancer features. You can read more about them in the Spring Cloud Commons documentation.

7.4. 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.)

### 7.5. 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.)

### 7.6. 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.

### 7.7. 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.

<table>
<thead>
<tr>
<th>Info</th>
</tr>
</thead>
<tbody>
<tr>
<td>If you use SockJS as a fallback over normal HTTP, you should configure a normal HTTP route as well as the websocket Route.</td>
</tr>
</tbody>
</table>

The following listing configures a websocket routing filter:
Example 61. application.yml

```yaml
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/**
```

7.8. 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.
- **httpStatusCode**: The HTTP Status of the request returned to the client.
- **httpMethod**: 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.

7.9. 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”.

## 8. HttpHeadersFilters

HttpHeadersFilters are applied to requests before sending them downstream, such as in the NettyRoutingFilter.

### 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.

### 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-hop-by-hop.headers` property to the list of header names to remove.

### 8.3. XForwarded Headers Filter

The `XForwarded` Headers Filter creates various a `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

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 62. application.yml

```yaml
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 63. application.yml

```yaml
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:

```yaml
Example 64. 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).

### 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:

```yaml
Example 65. application.yml

```spring:
  cloud:
    gateway:
      httpClient:
        ssl:
          handshake-timeout-millis: 10000
          close-notify-flush-timeout-millis: 3000
          close-notify-read-timeout-millis: 0
```

### 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 66. RouteDefinitionLocator.java

```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 67. application.yml

```yaml
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.

11. Route Metadata Configuration

You can configure additional parameters for each route by using metadata, as follows:
You could acquire all metadata properties from an exchange, as follows:

```java
Route route = exchange.getAttribute(GATEWAY_ROUTE_ATTR);
// get all metadata properties
route.getMetadata();
// get a single metadata property
route.getMetadata(someKey);
```

## 12. Http timeouts configuration

Http timeouts (response and connect) can be configured for all routes and overridden for each specific route.

### 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**

```yaml
spring:
  cloud:
    gateway:
      httpclient:
        connect-timeout: 1000
        response-timeout: 5s
```
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**

```yaml
- id: per_route_timeouts
  uri: https://example.org
  predicates:
    - name: Path
      args:
        pattern: /delay/{timeout}
  metadata:
    response-timeout: 200
    connect-timeout: 200
```

**per-route timeouts configuration using Java DSL**

```java
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();
}
```

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:
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.

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.

12.4.1. Configuring Predicates and Filters For **DiscoveryClient** Routes

By default, the gateway defines a single predicate and filter for routes created with a `DiscoveryClient`.

```java
// static imports from GatewayFilters and RoutePredicates
@Bean
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();
}
```
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 70. application.properties

```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}'"
```

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:
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 72. 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.
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 73. application.properties

```
management.endpoint.gateway.enabled=true  # default value
management.endpoints.web.exposure.include=gateway
```

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 74. application.properties

```
spring.cloud.gateway.actuator.verbose.enabled=false
```

This will default to true in a future release.
15.2. Retrieving Route Filters

This section details how to retrieve route filters, including:

- Global Filters
- [gateway-route-filters]

15.2.1. 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:

```json
{
    "org.springframework.cloud.gateway.filter.ReactiveLoadBalancerClientFilter@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.ReactiveLoadBalancerClientFilter@77856cc5`) and the corresponding order in the filter chain.

15.2.2. 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:
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@fceab5d 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.

### 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.

### 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:
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:

<table>
<thead>
<tr>
<th>Path</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>route_id</td>
<td>String</td>
<td>The route ID.</td>
</tr>
<tr>
<td>route_object.predicate</td>
<td>Object</td>
<td>The route predicate.</td>
</tr>
<tr>
<td>route_object.filters</td>
<td>Array</td>
<td>The GatewayFilter factories applied to the route.</td>
</tr>
<tr>
<td>order</td>
<td>Number</td>
<td>The route order.</td>
</tr>
</tbody>
</table>

### 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:
The following table describes the structure of the response:

<table>
<thead>
<tr>
<th>Path</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>String</td>
<td>The route ID.</td>
</tr>
<tr>
<td>predicates</td>
<td>Array</td>
<td>The collection of route predicates. Each item defines the name and the arguments of a given predicate.</td>
</tr>
<tr>
<td>filters</td>
<td>Array</td>
<td>The collection of filters applied to the route.</td>
</tr>
<tr>
<td>uri</td>
<td>String</td>
<td>The destination URI of the route.</td>
</tr>
<tr>
<td>order</td>
<td>Number</td>
<td>The route order.</td>
</tr>
</tbody>
</table>

### 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}`.

### 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):

<table>
<thead>
<tr>
<th>ID</th>
<th>HTTP Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>globalfilters</td>
<td>GET</td>
<td>Displays the list of global filters applied to the routes.</td>
</tr>
<tr>
<td>routefilters</td>
<td>GET</td>
<td>Displays the list of GatewayFilter factories applied to a particular route.</td>
</tr>
<tr>
<td>refresh</td>
<td>POST</td>
<td>Clears the routes cache.</td>
</tr>
<tr>
<td>routes</td>
<td>GET</td>
<td>Displays the list of routes defined in the gateway.</td>
</tr>
<tr>
<td>ID</td>
<td>HTTP Method</td>
<td>Description</td>
</tr>
<tr>
<td>---------------</td>
<td>-------------</td>
<td>--------------------------------------------------</td>
</tr>
<tr>
<td>routes/{id}</td>
<td>GET</td>
<td>Displays information about a particular route.</td>
</tr>
<tr>
<td>routes/{id}</td>
<td>POST</td>
<td>Adds a new route to the gateway.</td>
</tr>
<tr>
<td>routes/{id}</td>
<td>DELETE</td>
<td>Removes an existing route from the gateway.</td>
</tr>
</tbody>
</table>

16. Troubleshooting

This section covers common problems that may arise when you use Spring Cloud Gateway.

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

16.2. 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.

17. Developer Guide

These are basic guides to writing some custom components of the gateway.

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

```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
    }
}
```

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 75. PreGatewayFilterFactory.java

```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
    }
}
```
17.2.1. 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.

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
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
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();
}

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.

19. Configuration properties

To see the list of all Spring Cloud Gateway related configuration properties, see the appendix.

Spring Cloud Kubernetes

This reference guide covers how to use Spring Cloud Kubernetes.

1. Why do you need Spring Cloud Kubernetes?

Spring Cloud Kubernetes provides implementations of well known Spring Cloud interfaces allowing developers to build and run Spring Cloud applications on Kubernetes. While this project may be useful to you when building a cloud native application, it is also not a requirement in order to deploy a Spring Boot app on Kubernetes. If you are just getting started in your journey to running your Spring Boot app on Kubernetes you can accomplish a lot with nothing more than a basic Spring Boot app and Kubernetes itself. To learn more, you can get started by reading the Spring Boot reference documentation for deploying to Kubernetes and also working through the workshop material Spring and Kubernetes.

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. Starters that begin with spring-cloud-starter-kubernetes-fabric8 provide implementations using the Fabric8 Kubernetes Java Client. Starters that begin with spring-cloud-starter-kubernetes-client provide implementations using the Kubernetes Java Client.
<table>
<thead>
<tr>
<th><strong>Starter</strong></th>
<th><strong>Features</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Fabric8 Dependency</strong></td>
<td>Discovery Client implementation that resolves service names to Kubernetes Services.</td>
</tr>
<tr>
<td><code>&lt;dependency&gt;</code></td>
<td></td>
</tr>
<tr>
<td><code>&lt;groupId&gt;org.springframework.cloud&lt;/groupId&gt;</code></td>
<td></td>
</tr>
<tr>
<td><code>&lt;artifactId&gt;spring-cloud-starter-kubernetes-fabric8&lt;/artifactId&gt;</code></td>
<td></td>
</tr>
<tr>
<td><code>&lt;/dependency&gt;</code></td>
<td></td>
</tr>
<tr>
<td><strong>Kubernetes Client Dependency</strong></td>
<td></td>
</tr>
<tr>
<td><code>&lt;dependency&gt;</code></td>
<td></td>
</tr>
<tr>
<td><code>&lt;groupId&gt;org.springframework.cloud&lt;/groupId&gt;</code></td>
<td></td>
</tr>
<tr>
<td><code>&lt;artifactId&gt;spring-cloud-starter-kubernetes-client&lt;/artifactId&gt;</code></td>
<td></td>
</tr>
<tr>
<td><code>&lt;/dependency&gt;</code></td>
<td></td>
</tr>
<tr>
<td><strong>Fabric8 Dependency</strong></td>
<td>Load application properties from Kubernetes ConfigMaps and Secrets. Reload application properties when a ConfigMap or Secret changes.</td>
</tr>
<tr>
<td><code>&lt;dependency&gt;</code></td>
<td></td>
</tr>
<tr>
<td><code>&lt;groupId&gt;org.springframework.cloud&lt;/groupId&gt;</code></td>
<td></td>
</tr>
<tr>
<td><code>&lt;artifactId&gt;spring-cloud-starter-kubernetes-fabric8-config&lt;/artifactId&gt;</code></td>
<td></td>
</tr>
<tr>
<td><code>&lt;/dependency&gt;</code></td>
<td></td>
</tr>
<tr>
<td><strong>Kubernetes Client Dependency</strong></td>
<td></td>
</tr>
<tr>
<td><code>&lt;dependency&gt;</code></td>
<td></td>
</tr>
<tr>
<td><code>&lt;groupId&gt;org.springframework.cloud&lt;/groupId&gt;</code></td>
<td></td>
</tr>
<tr>
<td><code>&lt;artifactId&gt;spring-cloud-starter-kubernetes-client-config&lt;/artifactId&gt;</code></td>
<td></td>
</tr>
<tr>
<td><code>&lt;/dependency&gt;</code></td>
<td></td>
</tr>
</tbody>
</table>
### Starter Features

<table>
<thead>
<tr>
<th>Fabric8 Dependency</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;dependency&gt;</code></td>
</tr>
<tr>
<td><code>&lt;groupId&gt;org.springframework.cloud&lt;/groupId&gt;</code></td>
</tr>
<tr>
<td><code>&lt;artifactId&gt;spring-cloud-starter-kubernetes-fabric8-all&lt;/artifactId&gt;</code></td>
</tr>
<tr>
<td><code>&lt;/dependency&gt;</code></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Kubernetes Client Dependency</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;dependency&gt;</code></td>
</tr>
<tr>
<td><code>&lt;groupId&gt;org.springframework.cloud&lt;/groupId&gt;</code></td>
</tr>
<tr>
<td><code>&lt;artifactId&gt;spring-cloud-starter-kubernetes-client-all&lt;/artifactId&gt;</code></td>
</tr>
<tr>
<td><code>&lt;/dependency&gt;</code></td>
</tr>
</tbody>
</table>

### 3. DiscoveryClient for Kubernetes

This project provides an implementation of [Discovery Client for Kubernetes](https://kubernetes.io). This client lets you query Kubernetes endpoints (see [services](https://kubernetes.io/docs/concepts/services-networking/service/)) by name. A service is typically exposed by the Kubernetes API server as a collection of endpoints that represent [http](http) and [https](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:

<table>
<thead>
<tr>
<th>Fabric8 Kubernetes Client</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;dependency&gt;</code></td>
</tr>
<tr>
<td><code>&lt;groupId&gt;org.springframework.cloud&lt;/groupId&gt;</code></td>
</tr>
<tr>
<td><code>&lt;artifactId&gt;spring-cloud-starter-kubernetes-fabric8&lt;/artifactId&gt;</code></td>
</tr>
<tr>
<td><code>&lt;/dependency&gt;</code></td>
</tr>
</tbody>
</table>
Kubernetes Java Client

```xml
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-kubernetes-client</artifactId>
</dependency>
```

To enable loading of the `DiscoveryClient`, add `@EnableDiscoveryClient` to the according configuration or application class, as the following example shows:

```java
@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:

```java
@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
```

To discover service endpoint addresses that are not marked as "ready" by the kubernetes api server, you can set the following property in `application.properties` (default: false):

```
spring.cloud.kubernetes.discovery.include-not-ready-addresses=true
```

This might be useful when discovering services for monitoring purposes, and would enable inspecting the `/health` endpoint of not-ready service instances.
If your service exposes multiple ports, you will need to specify which port the DiscoveryClient should use. The DiscoveryClient will choose the port using the following logic.

1. If the service has a label primary-port-name it will use the port with the name specified in the label's value.
2. If no label is present, then the port name specified in spring.cloud.kubernetes.discovery.primary-port-name will be used.
3. If neither of the above are specified it will use the port named https.
4. If none of the above conditions are met it will use the port named http.
5. As a last resort it will pick the first port in the list of ports.

The last option may result in non-deterministic behaviour. Please make sure to configure your service and/or application accordingly.

By default all of the ports and their names will be added to the metadata of the ServiceInstance.

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.

### 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=)

5. Kubernetes PropertySource implementations

The most common approach to configuring your Spring Boot application is to create an application.properties or application.yml or an application-profile.properties or application-profile.yml 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.

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.yml 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 Fabric8ConfigMapPropertySource 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:
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 features 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:
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:

```yaml
kind: ConfigMap
apiVersion: v1
metadata:
  name: demo
data:
  application.yaml: |-
    pool:
      size:
        core: 1
        max: 16
```

The following example also works:

```yaml
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.yml: |
    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

```yaml
kind: ConfigMap
apiVersion: v1
metadata:
  name: demo
data:
  application.yml: |-
    greeting:
      message: Say Hello to the World
    farewell:
      message: Say Goodbye
```

```yaml
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
```

```yaml
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:

```yaml
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.

If you use `spring.cloud.kubernetes.config.paths` or `spring.cloud.kubernetes.secrets.path` the automatic reload functionality will not work. You will need to make a POST request to the `/actuator/refresh` endpoint or restart/redeploy the application.

Table 5. Properties:
<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>spring.cloud.kubernetes.config.enabled</td>
<td>Boolean</td>
<td>true</td>
<td>Enable ConfigMaps PropertySource</td>
</tr>
<tr>
<td>spring.cloud.kubernetes.config.name</td>
<td>String</td>
<td>${spring.application.name}</td>
<td>Sets the name of ConfigMap to look up</td>
</tr>
<tr>
<td>spring.cloud.kubernetes.config.namespace</td>
<td>String</td>
<td>Client namespace</td>
<td>Sets the Kubernetes namespace where to lookup</td>
</tr>
<tr>
<td>spring.cloud.kubernetes.config.paths</td>
<td>List</td>
<td>null</td>
<td>Sets the paths where ConfigMap instances are mounted</td>
</tr>
<tr>
<td>spring.cloud.kubernetes.config.enableApi</td>
<td>Boolean</td>
<td>true</td>
<td>Enable or disable consuming ConfigMap instances through APIs</td>
</tr>
</tbody>
</table>

### 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 Fabric8SecretsPropertySource 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:
The preceding command would create the following secret (which you can see by using `kubectl get secrets db-secret -o yaml`):

```yaml
apiVersion: v1
data:
  password: cDQ1NXcwcmQ=
  username: dXNlcg==
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
class: 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:
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:

```yaml
spring:
  application:
    name: cloud-k8s-app
  cloud:
    kubernetes:
      secrets:
        name: default-name
        namespace: default-namespace
        sources:
          - name: s1
            # Spring Cloud Kubernetes looks up a Secret named s1 in namespace default-namespace
          - namespace: n2
            # Spring Cloud Kubernetes looks up a Secret named default-name in whatever namespace n2
            - name: s2
          - namespace: n3
            # Spring Cloud Kubernetes looks up a Secret named s3 in 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 6. Properties:**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>spring.cloud.kubernetes.secrets.enabled</code></td>
<td>Boolean</td>
<td>true</td>
<td>Enable Secrets PropertySource</td>
</tr>
<tr>
<td><code>spring.cloud.kubernetes.secrets.name</code></td>
<td>String</td>
<td><code>${spring.application.name}</code></td>
<td>Sets the name of the secret to look up</td>
</tr>
<tr>
<td><code>spring.cloud.kubernetes.secrets.namespace</code></td>
<td>String</td>
<td></td>
<td>Client namespace</td>
</tr>
<tr>
<td><code>spring.cloud.kubernetes.secrets.labels</code></td>
<td>Map</td>
<td>null</td>
<td>Sets the labels used to lookup secrets</td>
</tr>
<tr>
<td>Name</td>
<td>Type</td>
<td>Default</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------------------------------------</td>
<td>----------</td>
<td>---------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>spring.cloud.kubernetes.secrets.paths</td>
<td>List</td>
<td>null</td>
<td>Sets the paths where secrets are mounted (example 1)</td>
</tr>
<tr>
<td>spring.cloud.kubernetes.secrets.enableApi</td>
<td>Boolean</td>
<td>false</td>
<td>Enables or disables consuming secrets through APIs (examples 2 and 3)</td>
</tr>
</tbody>
</table>

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](https://spring-boot-camel-config).

## 5.3. PropertySource Reload

⚠️ This functionality has been deprecated in the 2020.0 release. Please see the [Spring Cloud Kubernetes Configuration Watcher](https://spring-cloud-kubernetes) controller for an alternative way to achieve the same functionality.

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. In order for the restart context functionality to work properly you must enable and expose the restart actuator endpoint.
management:
  endpoint:
    restart:
      enabled: true
  endpoints:
    web:
      exposure:
        include: restart

• 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:

```java
@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

```java
@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:

```yaml
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 7. Properties:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>spring.cloud.kubernetes.reload.enabled</code></td>
<td>Boolean</td>
<td><code>false</code></td>
<td>Enables monitoring of property sources and configuration reload</td>
</tr>
<tr>
<td><code>spring.cloud.kubernetes.reload.monitoring-config-maps</code></td>
<td>Boolean</td>
<td><code>true</code></td>
<td>Allow monitoring changes in config maps</td>
</tr>
<tr>
<td><code>spring.cloud.kubernetes.reload.monitoring-secrets</code></td>
<td>Boolean</td>
<td><code>false</code></td>
<td>Allow monitoring changes in secrets</td>
</tr>
<tr>
<td><code>spring.cloud.kubernetes.reload.strategy</code></td>
<td>Enum</td>
<td><code>refresh</code></td>
<td>The strategy to use when firing a reload (refresh, restart_context, or shutdown)</td>
</tr>
<tr>
<td>Name</td>
<td>Type</td>
<td>Default</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>----------</td>
<td>---------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>spring.cloud.kubernetes.reload.mode</td>
<td>Enum</td>
<td>event</td>
<td>Specifies how to listen for changes in property sources (event or polling)</td>
</tr>
<tr>
<td>spring.cloud.kubernetes.reload.period</td>
<td>Duration</td>
<td>15s</td>
<td>The period for verifying changes when using the polling strategy</td>
</tr>
</tbody>
</table>

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.

### 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](https://fabric8.io/), 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|yml}` (or the profile specific one) otherwise it should be in `application.{properties|yml}` (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|yml}`

### 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).

### 6.2. Istio Awareness

When you include the `spring-cloud-kubernetes-fabric8-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 `@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.

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

8. Info Contributor

Spring Cloud Kubernetes includes an `InfoContributor` which adds Pod information to Spring Boot’s `/info` Acturator endpoint.

You can disable this `InfoContributor` by setting `management.info.kubernetes.enabled` to `false` in `bootstrap.[properties | yaml]`.

9. Leader Election

The Spring Cloud Kubernetes leader election mechanism implements the leader election API of Spring Integration using a Kubernetes ConfigMap.

Multiple application instances compete for leadership, but leadership will only be granted to one. When granted leadership, a leader application receives an `OnGrantedEvent` application event with leadership `Context`. Applications periodically attempt to gain leadership, with leadership granted to the first caller. A leader will remain a leader until either it is removed from the cluster, or it yields its leadership. When leadership removal occurs, the previous leader receives `OnRevokedEvent` application event. After removal, any instances in the cluster may become the new leader, including the old leader.

To include it in your project, add the following dependency.

```xml
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-kubernetes-fabric8-leader</artifactId>
</dependency>
```

To specify the name of the configmap used for leader election use the following property.
10. LoadBalancer for Kubernetes

This project includes Spring Cloud Load Balancer for load balancing based on Kubernetes Endpoints and provides implementation of load balancer based on Kubernetes Service. To include it to your project add the following dependency.

Fabric8 Implementation

```xml
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-kubernetes-fabric8-loadbalancer</artifactId>
</dependency>
```

Kubernetes Java Client Implementation

```xml
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-kubernetes-client-loadbalancer</artifactId>
</dependency>
```

To enable load balancing based on Kubernetes Service name use the following property. Then load balancer would try to call application using address, for example `service-a.default.svc.cluster.local`

```
spring.cloud.kubernetes.loadbalancer.mode=SERVICE
```

To enable load balancing across all namespaces use the following property. Property from `spring-cloud-kubernetes-discovery` module is respected.

```
spring.cloud.kubernetes.discovery.all-namespaces=true
```
11. Security Configurations Inside Kubernetes

11.1. 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:

```yaml
env:
  - name: "KUBERNETES_NAMESPACE"
    valueFrom:
      fieldRef:
        fieldPath: "metadata.namespace"
```

11.2. 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>
<thead>
<tr>
<th>Dependency</th>
<th>Resources</th>
</tr>
</thead>
<tbody>
<tr>
<td>spring-cloud-starter-kubernetes-fabric8</td>
<td>pods, services, endpoints</td>
</tr>
<tr>
<td>spring-cloud-starter-kubernetes-fabric8-config</td>
<td>configmaps, secrets</td>
</tr>
<tr>
<td>spring-cloud-starter-kubernetes-client</td>
<td>pods, services, endpoints</td>
</tr>
<tr>
<td>spring-cloud-starter-kubernetes-client-config</td>
<td>configmaps, secrets</td>
</tr>
</tbody>
</table>

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:
12. 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.

13. Spring Cloud Kubernetes Configuration Watcher

Kubernetes provides the ability to mount a ConfigMap or Secret as a volume in the container of your application. When the contents of the ConfigMap or Secret changes, the mounted volume will be updated with those changes.

However, Spring Boot will not automatically update those changes unless you restart the application. Spring Cloud provides the ability refresh the application context without restarting the application by either hitting the actuator endpoint `/refresh` or via publishing a `RefreshRemoteApplicationEvent` using Spring Cloud Bus.
To achieve this configuration refresh of a Spring Cloud app running on Kubernetes, you can deploy the Spring Cloud Kubernetes Configuration Watcher controller into your Kubernetes cluster.

The application is published as a container and is available on Docker Hub.

Spring Cloud Kubernetes Configuration Watcher can send refresh notifications to applications in two ways.

1. Over HTTP in which case the application being notified must of the /refresh actuator endpoint exposed and accessible from within the cluster

2. Using Spring Cloud Bus, in which case you will need a message broker deployed to your custer for the application to use.

13.1. Deployment YAML

Below is a sample deployment YAML you can use to deploy the Kubernetes Configuration Watcher to Kubernetes.
---

apiVersion: v1
kind: List
items:
- apiVersion: v1
  kind: Service
  metadata:
    labels:
      app: spring-cloud-kubernetes-configuration-watcher
      name: spring-cloud-kubernetes-configuration-watcher
  spec:
    ports:
    - name: http
      port: 8888
      targetPort: 8888
    selector:
      app: spring-cloud-kubernetes-configuration-watcher
    type: ClusterIP
- apiVersion: v1
  kind: ServiceAccount
  metadata:
    labels:
      app: spring-cloud-kubernetes-configuration-watcher
      name: spring-cloud-kubernetes-configuration-watcher
- apiVersion: rbac.authorization.k8s.io/v1
  kind: RoleBinding
  metadata:
    labels:
      app: spring-cloud-kubernetes-configuration-watcher
      name: spring-cloud-kubernetes-configuration-watcher:view
  roleRef:
    kind: Role
    apiGroup: rbac.authorization.k8s.io
    name: namespace-reader
  subjects:
  - kind: ServiceAccount
    name: spring-cloud-kubernetes-configuration-watcher
- apiVersion: rbac.authorization.k8s.io/v1
  kind: Role
  metadata:
    namespace: default
    name: namespace-reader
  rules:
  - apiGroups: ["", "extensions", "apps"]
    resources: ["configmaps", "pods", "services", "endpoints", "secrets"]
    verbs: ["get", "list", "watch"]
- apiVersion: apps/v1
  kind: Deployment
  metadata:
name: spring-cloud-kubernetes-configuration-watcher-deployment
spec:
  selector:
    matchLabels:
      app: spring-cloud-kubernetes-configuration-watcher
template:
    metadata:
      labels:
        app: spring-cloud-kubernetes-configuration-watcher
    spec:
      serviceAccount: spring-cloud-kubernetes-configuration-watcher
      containers:
      - name: spring-cloud-kubernetes-configuration-watcher
        image: springcloud/spring-cloud-kubernetes-configuration-watcher:2.0.1-SNAPSHOT
        imagePullPolicy: IfNotPresent
        readinessProbe:
          httpGet:
            port: 8888
            path: /actuator/health/readiness
        livenessProbe:
          httpGet:
            port: 8888
            path: /actuator/health/liveness
      ports:
      - containerPort: 8888

The Service Account and associated Role Binding is important for Spring Cloud Kubernetes Configuration to work properly. The controller needs access to read data about ConfigMaps, Pods, Services, Endpoints and Secrets in the Kubernetes cluster.

13.2. Monitoring ConfigMaps and Secrets

Spring Cloud Kubernetes Configuration Watcher will react to changes in ConfigMaps with a label of `spring.cloud.kubernetes.config` with the value `true` or any Secret with a label of `spring.cloud.kubernetes.secret` with the value `true`. If the ConfigMap or Secret does not have either of those labels or the values of those labels is not `true` then any changes will be ignored.

The labels Spring Cloud Kubernetes Configuration Watcher looks for on ConfigMaps and Secrets can be changed by setting `spring.cloud.kubernetes.configuration.watcher.configLabel` and `spring.cloud.kubernetes.configuration.watcher.secretLabel` respectively.

If a change is made to a ConfigMap or Secret with valid labels then Spring Cloud Kubernetes Configuration Watcher will take the name of the ConfigMap or Secret and send a notification to the application with that name.
13.3. HTTP Implementation

The HTTP implementation is what is used by default. When this implementation is used Spring Cloud Kubernetes Configuration Watcher and a change to a ConfigMap or Secret occurs then the HTTP implementation will use the Spring Cloud Kubernetes Discovery Client to fetch all instances of the application which match the name of the ConfigMap or Secret and send an HTTP POST request to the application’s actuator /refresh endpoint. By default it will send the post request to /actuator/refresh using the port registered in the discovery client.

13.3.1. Non-Default Management Port and Actuator Path

If the application is using a non-default actuator path and/or using a different port for the management endpoints, the Kubernetes service for the application can add an annotation called boot.spring.io/actuator and set its value to the path and port used by the application. For example

```yaml
apiVersion: v1
kind: Service
metadata:
  labels:
    app: config-map-demo
  name: config-map-demo
annotations:
  boot.spring.io/actuator: http://:9090/myactuator/home
spec:
  ports:
    - name: http
      port: 8080
      targetPort: 8080
  selector:
    app: config-map-demo
```

Another way you can choose to configure the actuator path and/or management port is by setting spring.cloud.kubernetes.configuration.watcher.actuatorPath and spring.cloud.kubernetes.configuration.watcher.actuatorPort.

13.4. Messaging Implementation

The messaging implementation can be enabled by setting profile to either bus-amqp (RabbitMQ) or bus-kafka (Kafka) when the Spring Cloud Kubernetes Configuration Watcher application is deployed to Kubernetes.

13.5. Configuring RabbitMQ

When the bus-amqp profile is enabled you will need to configure Spring RabbitMQ to point it to the location of the RabbitMQ instance you would like to use as well as any credentials necessary to
authenticate. This can be done by setting the standard Spring RabbitMQ properties, for example

```yaml
spring:
rabbitmq:
    username: user
    password: password
    host: rabbitmq
```

### 13.6. Configuring Kafka

When the `bus-kafka` profile is enabled you will need to configure Spring Kafka to point it to the location of the Kafka Broker instance you would like to use. This can be done by setting the standard Spring Kafka properties, for example

```yaml
spring:
kafka:
    producer:
        bootstrap-servers: localhost:9092
```

### 14. 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**
15. 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.

16. Configuration properties

To see the list of all Kubernetes related configuration properties please check the Appendix page.

17. Building

17.1. 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

```bash
$ ./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.
### 17.2. 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 `${main.basedir}` (defaults to `$/opt/jenkins/data/workspace/spring-cloud-release-train-docs-master-ci/train-docs/target/unpacked-docs`, 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.

### 17.3. 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.

#### 17.3.1. 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.

#### 17.3.2. 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`.

#### 17.3.3. 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. 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.

18.1. 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.

18.2. 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.

18.3. 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).
18.4. 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

18.4.1. 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.failsOnError>true</maven-checkstyle-plugin.failsOnError>
    <maven-checkstyle-plugin.failsOnViolation>true</maven-checkstyle-plugin.failsOnViolation>
    <maven-checkstyle-plugin.includeTestSourceDirectory>true</maven-checkstyle-plugin.includeTestSourceDirectory>
</properties>

<build>
    <plugins>
        <plugin>④</plugin>
        <groupId>io.spring.javaformat</groupId>
        <artifactId>spring-javaformat-maven-plugin</artifactId>
        </plugin>
        <plugin>⑤</plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-checkstyle-plugin</artifactId>
        </plugin>
    </plugins>
    <reporting>
        <plugins>⑤</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 `${project.root}/src/checkstyle/checkstyle-suppressions.xml` with your suppressions. Example:
It's advisable to copy the `${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
```

### 18.5. IDE setup

#### 18.5.1. 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
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.
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 Netflix**

**2020.0.3-SNAPSHOT**

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)
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.

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.

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:

```java
@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

```yaml
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`.

### 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.

When Eureka server requires client side certificate for authentication, the client side certificate and trust store can be configured via properties, as shown in following example:
The `eureka.client.tls.enabled` needs to be true to enable Eureka client side TLS. When `eureka.client.tls.trust-store` is omitted, a JVM default trust store is used. The default value for `eureka.client.tls.key-store-type` and `eureka.client.tls.trust-store-type` is PKCS12. When password properties are omitted, empty password is assumed.

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.

If you want to customize the RestTemplate used by the Eureka HTTP Client you may want to create a bean of `EurekaClientHttpRequestFactorySupplier` and provide your own logic for generating a `ClientHttpRequestFactory` instance.

### 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:

```yaml
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.
1.5. Registering a Secure Application

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

- `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.

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 then 'UP'. The following example shows how to enable health checks for the client:
If you require more control over the health checks, consider implementing your own `com.netflix.appinfo.HealthCheckHandler`.

### 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.

#### 1.7.1. 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

```
1.7.2. 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 bean = new EurekaInstanceConfigBean(inetUtils);
AmazonInfo info = AmazonInfo.Builder.newBuilder().autoBuild("eureka");
bean.setDataCenterInfo(info);
return bean;
}

1.7.3. 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}:${spring.application.instance_id:${s
erver.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:${spring.application.instanc
e_id:${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.

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.

1.8.1. EurekaClient without Jersey

By default, EurekaClient uses Spring's RestTemplate for HTTP communication. If you wish to use Jersey instead, you need to add the Jersey dependencies to your classpath. The following example shows the dependencies you need to add:

```xml
<dependency>
    <groupId>com.sun.jersey</groupId>
    <artifactId>jersey-client</artifactId>
</dependency>
<dependency>
    <groupId>com.sun.jersey</groupId>
    <artifactId>jersey-core</artifactId>
</dependency>
<dependency>
    <groupId>com.sun.jersey.contribs</groupId>
    <artifactId>jersey-apache-client4</artifactId>
</dependency>
```

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;
}

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.

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

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

Service 1 in Zone 2

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

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).

2. Service Discovery: Eureka Server

This section describes how to set up a Eureka server.

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:

```
spring:
  freemarker:
    template-loader-path: classpath:/templates/
    prefer-file-system-access: false
```
2.2. How to Run a Eureka Server

The following example shows a minimal Eureka server:

```java
@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:

```gradle
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}"
    }
}
```

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.

### 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)
```

```yaml
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.

### 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.
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}`.

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:
```java
@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](#).

## 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.

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

## 3. Configuration properties

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

### Spring Cloud OpenFeign

**2020.0.3-SNAPSHOT**

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

## 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, Spring Cloud CircuitBreaker, as well as
Spring Cloud LoadBalancer to provide a load-balanced http client when using Feign.

### 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**

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

**StoreClient.java**

```java
@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 `qualifiers` 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 configure a list of servers in your external configuration using `SimpleDiscoveryClient`.

Spring Cloud OpenFeign supports all the features available for the blocking mode of Spring Cloud LoadBalancer. You can read more about them in the project documentation.
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:

```java
@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`.

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.

```java
@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
• **MicrometerCapability** micrometerCapability: If feign-micrometer is on the classpath and MeterRegistry is available
• **Contract** feignContract: SpringMvcContract
• **Feign.Builder** feignBuilder: FeignCircuitBreaker.Builder
• **Client** feignClient: If Spring Cloud LoadBalancer is on the classpath, FeignBlockingLoadBalancerClient is used. If none of them is on the classpath, the default feign client is used.

---

spring-cloud-starter-openfeign supports spring-cloud-starter-loadbalancer. However, as is an optional dependency, you need to make sure it been added to your project if you want to use it.

The OkHttpClient and ApacheHttpClient and ApacheHC5 feign clients can be used by setting feign.okhttp.enabled or feign.httpclient.enabled or feign.httpclient.hc5.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 or org.apache.hc.client5.http.impl.classic.CloseableHttpClient when using Apache HC5.

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
- Capability (MicrometerCapability is provided by default)

A bean of Retryer.NEVER_RETRY with the type Retryer is created by default, which will disable retrying. Notice this retrying behavior is different from the Feign default one, where it will automatically retry IOExceptions, treating them as transient network related exceptions, and any RetryableException thrown from an ErrorDecoder.

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:
```java
@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.

```yaml
application.yml
```
```
feign:
  client:
    config:
      feignName:
        connectTimeout: 5000
        readTimeout: 5000
        loggerLevel: full
        errorDecoder: com.example.SimpleErrorDecoder
        retryer: com.example.SimpleRetryer
        defaultQueryParameters:
          query: queryValue
        defaultRequestHeaders:
          header: headerValue
        requestInterceptors:
          - com.example.FooRequestInterceptor
          - com.example.BarRequestInterceptor
        decode404: false
        encoder: com.example.SimpleEncoder
        decoder: com.example.SimpleDecoder
        contract: com.example.SimpleContract
        capabilities:
          - com.example.FooCapability
          - com.example.BarCapability
        metrics.enabled: false
```

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.

You can use `feign.client.config.feignName.defaultQueryParameters` and `feign.client.config.feignName.defaultRequestHeaders` to specify query parameters and headers that will be sent with every request of the client named `feignName`.

```yaml
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 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.

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

```java
@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;
        }
    };
}

By default, Feign clients do not encode slash / characters. You can change this behaviour, by setting the value of `feign.client.decodeSlash` to `false`.

### 1.2.1. SpringEncoder configuration

In the `SpringEncoder` that we provide, we set `null` charset for binary content types and `UTF-8` for all the other ones.

You can modify this behaviour to derive the charset from the `Content-Type` header charset instead by setting the value of `feign.encoder.charset-from-content-type` to `true`.

### 1.3. Timeout Handling

We can configure timeouts on both the default and the named client. OpenFeign works with two timeout parameters:

- **connectTimeout** prevents blocking the caller due to the long server processing time.
- **readTimeout** is applied from the time of connection establishment and is triggered when returning the response takes too long.

In case the server is not running or available a packet results in *connection refused*. The communication ends either with an error message or in a fallback. This can happen *before* the `connectTimeout` if it is set very low. The time taken to perform a lookup and to receive such a packet causes a significant part of this delay. It is subject to change based on the remote host that involves a DNS lookup.

### 1.4. 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](https://feign.openfeign.co). Below is
an example which creates two Feign Clients with the same interface but configures each one with a separate request interceptor.

```java
@Import(FeignClientsConfiguration.class)
class FooController {
    private FooClient fooClient;
    private FooClient adminClient;

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

        this.adminClient = Feign.builder().client(client)
            .encoder(encoder)
            .decoder(decoder)
            .contract(contract)
            .addCapability(micrometerCapability)
            .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 OpenFeign.

`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`.

### 1.5. Feign Spring Cloud CircuitBreaker Support

If Spring Cloud CircuitBreaker is on the classpath and `feign.circuitbreaker.enabled=true`, Feign will wrap all methods with a circuit breaker.
To disable Spring Cloud CircuitBreaker support on a per-client basis create a vanilla `Feign.Builder` with the "prototype" scope, e.g.:

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

The circuit breaker name follows this pattern `<feignClientName>#<calledMethod>`. When calling a `@FeignClient` with name `foo` and the called interface method is `bar` then the circuit breaker name will be `foo_bar`.

To enable Spring Cloud CircuitBreaker group set the `feign.circuitbreaker.group.enabled` property to `true` (by default `false`).

### 1.6. Feign Spring Cloud CircuitBreaker Fallbacks

Spring Cloud CircuitBreaker supports the notion of a fallback: a default code path that is executed when the 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.
If one needs access to the cause that made the fallback trigger, one can use the `fallbackFactory` attribute inside `@FeignClient`. 
1.7. Feign and @Primary

When using Feign with Spring Cloud CircuitBreaker 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 OpenFeign 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
}

1.8. Feign Inheritance Support

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

UserService.java

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

UserResource.java

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

UserClient.java

```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).

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

```yaml
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:

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

### 1.10. 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.

```yaml
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:

```java
@Configuration
public class FooConfiguration {
    @Bean
    Logger.Level feignLoggerLevel() {
        return Logger.Level.FULL;
    }
}
```
1.11. Feign Capability support

The Feign capabilities expose core Feign components so that these components can be modified. For example, the capabilities can take the Client, decorate it, and give the decorated instance back to Feign. The support for metrics libraries is a good real-life example for this. See Feign metrics.

Creating one or more Capability beans and placing them in a @FeignClient configuration lets you register them and modify the behavior of the involved client.

```java
@Configuration
public class FooConfiguration {
    @Bean
    Capability customCapability() {
        return new CustomCapability();
    }
}
```

1.12. Feign metrics

If all of the following conditions are true, a MicrometerCapability bean is created and registered so that your Feign client publishes metrics to Micrometer:

- feign-micrometer is on the classpath
- A MeterRegistry bean is available
- feign metrics properties are set to true (by default)
  - feign.metrics.enabled=true (for all clients)
  - feign.client.config.feignName.metrics.enabled=true (for a single client)

If your application already uses Micrometer, enabling metrics is as simple as putting feign-micrometer onto your classpath.

You can also disable the feature by either:

- excluding feign-micrometer from your classpath
- setting one of the feign metrics properties to false
  - feign.metrics.enabled=false
  - feign.client.config.feignName.metrics.enabled=false

feign.metrics.enabled=false disables metrics support for all Feign clients regardless of the value of the client-level flags: feign.client.config.feignName.metrics.enabled. If you want to enable or disable metrics per client, don’t set feign.metrics.enabled and use feign.client.config.feignName.metrics.enabled.
You can also customize the `MicrometerCapability` by registering your own bean:

```java
@Configuration
public class FooConfiguration {
    @Bean
    public MicrometerCapability micrometerCapability(MeterRegistry meterRegistry) {
        return new MicrometerCapability(meterRegistry);
    }
}
```

### 1.13. 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`:

```java
// 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:

```java
@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.

### 1.14. HATEOAS support

Spring provides some APIs to create REST representations that follow the HATEOAS principle, `Spring Hateoas` and `Spring Data REST`.
If your project uses 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();
}
```

### 1.15. 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();
}
```
1.16. Feign CollectionFormat support

We support feign.CollectionFormat by providing the @CollectionFormat annotation. You can annotate a Feign client method with it by passing the desired feign.CollectionFormat as annotation value.

In the following example, the CSV format is used instead of the default EXPLODED to process the method.

```java
@FeignClient(name = "demo")
protected interface PageableFeignClient {

    @CollectionFormat(feign.CollectionFormat.CSV)
    @GetMapping(path = "/page")
    ResponseEntity performRequest(Pageable page);
}
```

Set the CSV format while sending Pageable as a query parameter in order for it to be encoded correctly.

1.17. 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.

1.17.1. 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.

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

1.18. Spring Data Support

You may consider enabling Jackson Modules for the support org.springframework.data.domain.Page and org.springframework.data.domain.Sort decoding.

```java
feign.autoconfiguration.jackson.enabled=true
```
1.19. Spring @RefreshScope Support

If Feign client refresh is enabled, each feign client is created with `feign.Request.Options` as a refresh-scoped bean. This means properties such as `connectTimeout` and `readTimeout` can be refreshed against any Feign client instance through `POST /actuator/refresh`.

By default, refresh behavior in Feign clients is disabled. Use the following property to enable refresh behavior:

```java
feign.client.refresh-enabled=true
```

DO NOT annotate the `@FeignClient` interface with the `@RefreshScope` annotation.

2. Configuration properties

To see the list of all Spring Cloud OpenFeign related configuration properties please check the Appendix page.

Spring Cloud Sleuth Reference Documentation

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

The reference documentation consists of the following sections:

- **Legal**: Legal information.
- **Documentation Overview**: About the Documentation, Getting Help, First Steps, and more.
- **Getting Started**: Introducing Spring Cloud Sleuth, Developing Your First Spring Cloud Sleuth-based Application
- **Using Spring Cloud Sleuth**: Spring Cloud Sleuth usage examples and workflows.
- **Spring Cloud Sleuth Features**: Span creation, context propagation, and more.
- **“How-to” Guides**: Add sampling, propagate remote tags, and more.
- **Spring Cloud Sleuth Integrations**: Instrumentation configuration, context propagation, and more.
1. Preface

1.1. 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 . . .

1.2. 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

### 1.2.1. Creating a Sample Application by Using Spring Initializr

To get started, visit the [Spring Initializr](https://start.spring.io). 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 Initiailzer 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](/docs/src/main/asciidoc/images/spring-initializr.png)

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.

### 1.2.2. 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.

### 1.2.3. Adding a Message Handler, Building, and Running

Modify the `com.example.loggingconsumer.LoggingConsumerApplication` class to look as follows:

```java
@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.CbMIwdkJSB01ZoPD0tHtCg, 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.CbMIwdkJSB01ZoPD0tHtCg
--- [ 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.CbMIwdkJSB01ZoPD0tHtCg. The anonymous.CbMIwdkJSB01ZoPD0tHtCg 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.

### 2. What’s New in 3.0?

#### 2.1. 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.

### 2.2. 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 if 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.

### 3. 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 adding `spring-cloud-stream` dependencies to the classpath of your application, you get immediate connectivity to a message broker exposed by the provided `spring-cloud-stream` binder (more on that later), and you can implement your functional requirement, which is run (based on the incoming message) by a `java.util.function.Function`.

The following listing shows a quick example:
```java
@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();
    }
}
```

The following listing shows the corresponding test:

```java
@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());
    }
}
```

4. 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
4.1. 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-
4.1.1. 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.

4.2. 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 [spring-cloud-stream-overview-custom-binder-impl] 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.

4.3. 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-
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.

### 4.4. 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.github.com/spring-cloud/spring-cloud-]
Figure 9. 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.

4.5. 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.

4.5.1. 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).

4.6. 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-
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.

5. 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).
5.1. 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.

5.2. 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.

```java
@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();
        };
    }
}
```

The above example looks no different then any vanilla spring-boot application. It defines a single bean of type Function and that it is. So, how does it became 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.

5.2.1. 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:

```java
@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:
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 altogether, 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.

**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.

5.3. 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.

5.3.1. 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. Some time this auto-discovery can get in the way, since single bean of type java.util.function.[Supplier/Function/Consumer] could be there for purposes other then handling messages, yet being single it is auto-discovered and auto-bound. For these rare scenarios you can disable auto-discovery by providing spring.cloud.stream.function.autodetect property with value set to false.

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.
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`:

```java
@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`:

```java
@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:

```java
@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 example:

```java
@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:

```java
@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.

**Supplier & threading**

As you have learned by now, unlike `Function` and `Consumer`, which are triggered by an event (they have input data), `Supplier` does not have any input and thus triggered by a different mechanism - `poller`, which may have an unpredictable threading mechanism. And while the details of the threading mechanism most of the time are not relevant to the downstream execution of the function it may present an issue in certain cases especially with integrated frameworks that may have certain expectations to thread affinity. For example, Spring Cloud Sleuth which relies on tracing data stored in thread local. For those cases we have another mechanism via `StreamBridge`, where user has more control over threading mechanism. You can get more details in Sending arbitrary data to an output (e.g. Foreign event-driven sources) section.
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:

```java
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.

**cron**

Cron expression value for the Cron Trigger.

Default: none.

**initialDelay**

Initial delay for periodic triggers.

Default: 0.

**timeUnit**

The TimeUnit to apply to delay values.

Default: MILLISECONDS.

For example --spring.cloud.stream.poller.fixed-delay=2000 sets the poller interval to poll every two seconds.
Sending arbitrary data to an output (e.g. 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 StreamBridge mechanism.

```java
@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

```java
@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("myDestination", 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 `myDestination` name which does not exist as a binding. Therefore such name will be treated as dynamic destination as described in Routing FROM Consumer section.

In the preceding example, we are using `ApplicationRunner` as a foreign source to feed the stream.

A more practical example, where the foreign source is REST endpoint.
As you can see inside of `delegateToSupplier` method we're using StreamBridge to send data to `myBinidng` binding. And here you're also benefiting from the dynamic features of StreamBridge where if `myBinidng` doesn't exist it will be created automatically, otherwise existing binding will be used.

By showing two example we want to emphasize the approach will work with any 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
@Controller
public class WebSourceApplication {

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

    @Autowired
    private StreamBridge streamBridge;

    @RequestMapping
    @ResponseStatus(HttpStatus.ACCEPTED)
    public void delegateToSupplier(@RequestBody String body) {
        //processor.onNext(body);
        streamBridge.send("myBinidng", body);
    }
}
```
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

```java
@Bean
class 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-in-0=quotedUpperCaseInput` allowing other configuration properties to refer to that binding name (e.g., `spring.cloud.stream.bindings.quotedUpperCaseInput.destination=myDestination`).

Functional Composition and Cross-cutting Concerns

Function composition effectively allows you to address complexity by breaking it down to a set of simple and individually manageable/testable components that could still be represented as one at runtime. But that is not the only benefit.

You can also use composition to address certain cross-cutting non-functional concerns, such as content enrichment. For example, assume you have an incoming message that may be lacking certain headers, or some headers are not in the exact state your business function would expect. You can now implement a separate function that addresses those concerns and then compose it with the main business function.

Let's look at the example
@SpringBootApplication
public class DemoStreamApplication {

    public static void main(String[] args) {
        SpringApplication.run(DemoStreamApplication.class,
                "--spring.cloud.function.definition=enrich|echo",
                "--spring.cloud.stream.function.bindings.enrich|echo-in-0=input",
                "--spring.cloud.stream.bindings.input.destination=myDestination",
                "--spring.cloud.stream.bindings.input.group=myGroup");
    }

    @Bean
    public Function<Message<String>, Message<String>> enrich() {
        return message -> {
            Assert.isTrue(!message.getHeaders().containsKey("foo"), "Should NOT contain 'foo' header");
            return MessageBuilder.fromMessage(message).setHeader("foo", "bar").build();
        };
    }

    @Bean
    public Function<Message<String>, Message<String>> echo() {
        return message -> {
            Assert.isTrue(message.getHeaders().containsKey("foo"), "Should contain 'foo' header");
            System.out.println("Incoming message " + message);
            return message;
        };
    }
}

While trivial, this example demonstrates how one function enriches the incoming Message with the additional header(s) (non-functional concern), so the other function - echo - can benefit form it. The echo function stays clean and focused on business logic only. You can also see the usage of spring.cloud.stream.function.bindings property to simplify composed binding name.

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 the 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:

```java
@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. For example, the following will override content-type for gather-in-0 binding:

```
--spring.cloud.stream.bindings.gather-in-0.content-type=text/plain
```
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 scatter-in-0 and the output bindings are scatter-out-0 and scatter-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")) {

        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 a conflict (more then 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:

```java
@Test
public void testMultipleFunctions() {
    try (ConfigurableApplicationContext context = new SpringApplicationBuilder(
            TestChannelBinderConfiguration.getCompleteConfiguration(ReactiveFunctionConfiguration.class))
            .run("--spring.cloud.function.definition=uppercase;reverse")) {

        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
defindFirstPerson() {
    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:

```java
@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.

5.3.2. Using Polled Consumers

Overview

When using polled consumers, you poll the PollableMessageSource on demand. To define binding for polled consumer you need to provide spring.cloud.stream.pollable-source property.

Consider the following example of a polled consumer binding:

```
--spring.cloud.stream.pollable-source=myDestination
```

The pollable-source name myDestination in the preceding example will result in myDestination-in-0 binding name to stay consistent with functional programming model.

Given the polled consumer in the preceding example, you might use it as follows:

```java
@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,
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:

```java
@Bean
class PollerExample {
    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:

```java
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.

### 5.4. Event Routing

Event Routing, in the context of Spring Cloud Stream, is the ability to either a) route events to a particular event subscriber or b) route events produced by an event subscriber to a particular destination. Here we'll refer to it as route ‘TO’ and route ‘FROM’.

#### 5.4.1. 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

```java
@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 a 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 a `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 the 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']`).
Passing instructions via application properties is especially important for reactive functions given that a reactive function is only invoked once to pass the Publisher, so access to the individual items is limited.

5.4.2. 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 (that is, intentionally allowed values). 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
@Component
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:

```java
@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 a 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:

```java
void configure(String destinationName, MessageChannel channel, ProducerProperties producerProperties,
                T extendedProducerProperties);
```

The following example shows how to use the RabbitMQ binder:

```java
@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.

## 5.5. 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, the techniques are dependent on binder implementation and the capability of the underlying messaging middleware as well as programming model (more on this later).

Whenever Message handler (function) throws an exception, it is propagated back to the binder, and the binder subsequently propagates the error back to the messaging system. The framework then will make several attempts at re-trying the same message (3 by default) using RetryTemplate provided by the Spring Retry library.
After that, 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.

Keep in mind however, the reactive function does NOT qualify as a Message handler, since it does not handle individual messages and instead provides a way to connect stream (i.e., Flux) provided by the framework with the one provided by the user. In other way of looking at it is - Message handler (i.e., imperative function) is invoked for each Message, while the reactive function is invoked only once during the initialization to connect two stream definitions at which point framework effectively hands off any and all control to the reactive API.

Why is this important? That is because anything you read later in this section with regard to Retry Template, dropping failed messages, retrying, DLQ and configuration properties that assist with all of it only applies to Message handlers (i.e., imperative functions).

Reactive API provides a very rich library of its own operators and mechanisms to assist you with error handling specific to variety of reactive uses cases which are far more complex then simple Message handler cases, So use them, such as public final Flux<T> retryWhen(Retry retrySpec); that you can find in reactor.core.publisher.Flux.

```java
@Bean
public Function<Flux<String>, Flux<String>> uppercase() {
    return flux -> flux
        .retryWhen(Retry.backoff(3, Duration.ofMillis(1000)))
        .map(v -> v.toUpperCase());
}
```

5.5.1. 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.

5.5.2. DLQ - Dead Letter Queue

Perhaps the most common mechanism, DLQ allows failed messages to be sent to a special destination: the Dead Letter Queue.

When configured, failed messages are sent to this destination for subsequent re-processing or auditing and reconciliation.

Consider the following example:
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 the group property must be provided for proper naming of the DLQ destination. However group is 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:
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
```

### 5.5.3. Retry Template

In this section we cover configuration properties relevant to configuration of retry capabilities.

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.

**backOffInitialInterval**

The backoff initial interval on retry.

Default 1000 milliseconds.

**backOffMaxInterval**

The maximum backoff interval.

Default 10000 milliseconds.

**backOffMultiplier**

The backoff multiplier.
**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 the 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,

```java
@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.

```java
spring.cloud.stream.bindings.<foo>.consumer.retry-template-name=<your-retry-template-bean-name>
```

### 6. 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.

#### 6.1. Producers and Consumers

The following image shows the general relationship of producers and consumers:
producers consumers | https://raw.githubusercontent.com/spring-cloud/spring-cloud-
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 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).

6.2. 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:

```java
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:

```bash
kafka:
org.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 [spring-cloud-stream-overview-custom-binder-impl] section.

### 6.3. 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.

#### 6.3.1. 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:

```xml
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-stream-binder-rabbit</artifactId>
</dependency>
```

For the specific Maven coordinates of other binder dependencies, see the documentation of that binder implementation.

### 6.4. 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:
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:

```java
spring.cloud.stream.bindings.input.binder=kafka
spring.cloud.stream.bindings.output.binder=rabbit
```

### 6.5. 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:
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:

```yaml
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:

```yaml
environment:
  spring:
    profiles:
      active: myBinderProfile
```
6.6. Customizing binders in multi binder applications

When an application has multiple binders in it and wants to customize the binders, then that can be achieved by providing a `BinderCustomizer` implementation. In the case of applications with a single binder, this special customizer is not necessary since the binder context can access the customization beans directly. However, this is not the case in a multi-binder scenario, since various binders live in different application contexts. By providing an implementation of `BinderCustomizer` interface, the binders, although reside in different application contexts, will receive the customization. Spring Cloud Stream ensures that the customizations take place before the applications start using the binders. The user must check for the binder type and then apply the necessary customizations.

Here is an example of providing a `BinderCustomizer` bean.

```java
@Bean
public BinderCustomizer binderCustomizer() {
    return (binder, binderName) -> {
        if (binder instanceof KafkaMessageChannelBinder) {
            ((KafkaMessageChannelBinder) binder).setRebalanceListener(...);
        }
        else if (binder instanceof KStreamBinder) {
            ...
        } else if (binder instanceof RabbitMessageChannelBinder) {
            ...
        }
    };
}
```

Note that, when there are more than one instance of the same type of the binder, the binder name can be used to filter customization.

6.7. Binding visualization and control

Spring Cloud Stream supports visualization and control of the Bindings through Actuator endpoints as well as programmatic way.

6.7.1. Programmatic way

Since version 3.1 we expose `org.springframework.cloud.stream.binding.BindingsLifecycleController` which is registered as bean and once injected could be used to control the lifecycle of individual bindings.

For example, looks at the fragment from one of the test cases. As you can see we retrieve `BindingsLifecycleController` from spring application context and execute individual methods to control the lifecycle of `echo-in-0` binding.
BindingsLifecycleController bindingsController = context.getBean(BindingsLifecycleController.class);
Binding binding = bindingsController.queryState("echo-in-0");
assertThat(binding.isRunning()).isTrue();
bindingsController.changeState("echo-in-0", State.STOPPED);
//bindingsController.stop("echo-in-0") alternative way of changing state. For
//convenience we expose start/stop and pause/resume operations.
assertThat(binding.isRunning()).isFalse();

==== Actuator
Since 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:

[source,xml]

<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:

[source,xml]

<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-webflux</artifactId>
</dependency>

You can add the Actuator dependency as follows:

[source,xml]

<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>

NOTE: 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:
'http://<host>:<port>/actuator/bindings'

Alternative, to see a single binding, access one of the URLs similar to the following:
'http://<host>:<port>/actuator/bindings/<bindingName>'

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:

```bash
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
```

NOTE: `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.

=== 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`.

[[spring-cloud-stream-overview-custom-binder-impl]]

=== 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)_:

```xml
<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:

```java
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:

[source,java]

```java
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);
      }
    });
  }
}
```
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);
    }
}

NOTE: 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{
```
NOTE: 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.:_
[source,java]

```java
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)_:

[source,java]

@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:

[source]

```
myFileBinder:
com.example.springcloudstreamcustombinder.config.FileMessageBinderConfiguration
```

== 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.

=== 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,Multiple Binders on the Classpath>>.
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 <<multiple-systems, 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'

[[binding-properties]]
=== 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
[source,java]

@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,Consumer Groups>>.
+ Default: `null` (indicating an anonymous consumer).

**contentType::**
The content type of this binding.
See `<<content-type-management>>`.
+ Default: `application/json`.

**binder::**
The binder used by this binding.
See `<<multiple-binders>>` 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.uppercase-in-0.consumer`). (for example, `spring.cloud.stream.bindings.uppercase-in-0.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 `<<spring-cloud-stream-overview-instance-index-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 `<<spring-cloud-stream-overview-instance-index-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’.
+ 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:

```java
@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>>`
  + 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 de-serializer) 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.

[[content-type-management]]
== 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:

. To convert the contents of the incoming message to match the signature of the application-provided handler.

. 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`.

NOTE: As a supplement to the details to follow, you may also want to read the following https://spring.io/blog/2018/02/26/spring-cloud-stream-2-0-content-type-negotiation-and-transformation[blog post].

=== 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:

[source, java]

public Function<Person, Person> personFunction {..}

NOTE: 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):

. *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`.

. *BINDING*: The `contentType` can be set per destination binding by setting the `spring.cloud.stream.bindings.input.content-type` property.

+ NOTE: 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`.

. *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 `<<spring-cloud-stream-overview-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.

NOTE: 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:

```
[source, java]
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:

[source, java]
```
Object fromMessage(Message<?> message, Class<?> targetClass);
Message<byte[]> toMessage(Object payload, @Nullable MessageHeaders headers);
```

=== 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):

- `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).
- `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.
- `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[])`.
- `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 `<<spring-cloud-stream-overview-user-defined-message-converters>>`.

[[spring-cloud-stream-overview-user-defined-message-converters]]

### 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.

NOTE: 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 MimeTypeInfo("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:

* `<<spring-cloud-stream-overview-connecting-multiple-application-instances>>`
* `<<spring-cloud-stream-overview-instance-index-instance-count>>`
* `<<spring-cloud-stream-overview-partitioning>>`

[[spring-cloud-stream-overview-connecting-multiple-application-instances]]

=== 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
```

### 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.

### Partitioning

Partitioning in Spring Cloud Stream consists of two tasks:

* `<spring-cloud-stream-overview-configuring-output-bindings-partitioning>`
* `<spring-cloud-stream-overview-configuring-input-bindings-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:

[source]

```--spring.cloud.stream.bindings.func-out-0.producer.partitionKeyExtractorName=customPartitionKeyExtractor```

NOTE: 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:

[source]

```--spring.cloud.stream.bindings.func-out-0.producer.partitionSelectorName=customPartitionSelector```
NOTE: 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.

```java
@Bean public CustomPartitionSelectorClass customPartitionSelector() { return new CustomPartitionSelectorClass(); }
```

==== 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.

== Testing
Spring Cloud Stream provides support for testing your microservice applications without connecting to a messaging system.

--- 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 https://spring.io/projects/spring-integration [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.

```xml
<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>
```
Or for build.gradle.kts
[source,kotlin]
--testImplementation("org.springframework.cloud:spring-cloud-stream") {
artifact {
name = "spring-cloud-stream"
extension = "jar"
type ="test-jar"
classifier = "test-binder"
}
}
---

==== Test Binder usage
Now you can test your microservice as a simple unit test
[source,java]
@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();
    }
}
```java
@Bean
public Function<String, String> reverse() {
    return value -> new StringBuilder(value).reverse().toString();
}
}

and the actual test

```java
@Test public void testMultipleFunctions() { try (ConfigurableApplicationContext context = new SpringApplicationBuilder( TestChannelBinderConfiguration.getCompleteConfiguration( SampleFunctionConfiguration.class)) .run("--spring.cloud.function.definition=uppercase;reverse")
{

    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, "reverse-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:

```java
@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")
{
```

```
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, "reverse-in-0");

Message<byte[]> outputMessage = outputDestination.receive(0, "myOutput");
assertThat(outputMessage.getPayload()).isEqualTo("HELLO").getBytes());

outputMessage = outputDestination.receive(0, "reverse-out-0");
assertThat(outputMessage.getPayload()).isEqualTo("olleH").getBytes());

==== 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:

[source, java]

@Test public void samplePollingTest() { ApplicationContext context = new
SpringApplicationBuilder(SamplePolledConfiguration.class) .web(WebApplicationType.NONE)
.run("--spring.jmx.enabled=false", "--spring.cloud.stream.pollable-source=myDestination");
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())); }

@Import(TestChannelBinderConfiguration.class) @EnableAutoConfiguration public static class
SamplePolledConfiguration { @Bean public ApplicationRunner poller(PollableMessageSource
polledMessageSource, StreamBridge 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("myOutput", newPayload); }) { Thread.sleep(2000); } } catch (Exception e) { // handle failure } });

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-A823-4053D030B5C74

NOTE: 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.

== 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_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 <<multiple-systems,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.

Let's 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 check 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.
== Samples

For Spring Cloud Stream samples, see the https://github.com/spring-cloud/spring-cloud-stream-samples[spring-cloud-stream-samples] repository on GitHub.

=== Deploying Stream Applications on CloudFoundry

On CloudFoundry, services are usually exposed through a special environment variable called https://docs.cloudfoundry.org/devguide/deploy-apps/environment-variable.html#VCAP-SERVICES[VCAP_SERVICES].

When configuring your binder connections, you can use the values from an environment variable as explained on the http://docs.spring.io/spring-cloud-dataflow-server-cloudfoundry/docs/current-SNAPSHOT/reference/htmlsingle/#getting-started-ups[Dataflow Cloud Foundry Server] docs.

== Binder Implementations

The following is the list of available binder implementations

* https://cloud.spring.io/spring-cloud-stream-binder-rabbit/[RabbitMQ]
* https://cloud.spring.io/spring-cloud-stream-binder-kafka/[Apache Kafka]
* https://github.com/spring-cloud/spring-cloud-stream-binder-aws-kinesis[Amazon Kinesis]
* https://github.com/SolaceProducts/spring-cloud-stream-binder-solace[Solace PubSub+ _(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 https://medium.com/@domenicosibilio/how-to-create-a-spring-cloud-stream-binder-from-scratch-ab8b29ee931b[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 ‘<<spring-cloud-stream-overview-custom-binder-impl>>’ section.
This section provides a brief overview of the Spring Cloud Task reference documentation. Think of it as a map for the rest of the document. You can read this reference guide in a linear fashion or you can skip sections if something does not interest you.

== About the documentation
The Spring Cloud Task reference guide is available in {spring-cloud-task-docs}[html] and {spring-cloud-task-docs}/index.pdf[pdf], {spring-cloud-task-docs}/index.epub[epub]. The latest copy is available at {spring-cloud-task-docs-current}.

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.

== Getting help
Having trouble with Spring Cloud Task? We would like to help!

NOTE: All of Spring Cloud Task is open source, including the documentation. If you find a problem with the docs or if you just want to improve them, please {github-code}[get involved].

[[task-documentation-first-steps]]
== First Steps
If you are just getting started with Spring Cloud Task or with 'Spring' in general, we suggesting reading the <<getting-started.adoc#getting-started>> chapter.

To get started from scratch, read the following sections:

* <<getting-started.adoc#getting-started-introducing-spring-cloud-task, Introducing Spring Cloud Task>>
* <<getting-started.adoc#getting-started-system-requirements, System Requirements>>

To follow the tutorial, read
<<getting-started.adoc#getting-started-developing-first-task, Developing Your First Spring Cloud Task Application>>

To run your example, read
<<getting-started.adoc#getting-started-running-the-example, Running the Example>>

[[getting-started]]
= Getting started

[[partintro]]
--
If you are just getting started with Spring Cloud Task, you should read this section. Here, we answer the basic "`what?`", "`how?`", and "`why?`" questions. We start with a gentle introduction to Spring Cloud Task. We then build a Spring Cloud Task application, discussing some core principles as we go.
--

[[getting-started-introducing-spring-cloud-task]]
== Introducing Spring Cloud Task

Spring Cloud Task makes it easy to create short-lived microservices. It provides capabilities that let short lived JVM processes be executed on demand in a production environment.

[[getting-started-system-requirements]]
== System Requirements

You need to have Java installed (Java 8 or better). To build, you need to have Maven installed as well.

=== Database Requirements
Spring Cloud Task uses a relational database to store the results of an executed task. While you can begin developing a task without a database (the status of the task is logged as part of the task repository's updates), for production environments, you want to use a supported database. Spring Cloud Task currently supports the following databases:

- DB2
- H2
- HSQLDB
- MySql
- Oracle
- Postgres
- SqlServer

[[getting-started-developing-first-task]]
== Developing Your First Spring Cloud Task Application

A good place to start is with a simple "Hello, World!" application, so we create the Spring Cloud Task equivalent to highlight the features of the framework. Most IDEs have good support for Apache Maven, so we use it as the build tool for this project.

NOTE: The spring.io website contains many guides that use Spring Boot. If you need to solve a specific problem, check there first. You can shortcut the following steps by going to the https://start.spring.io/ [Spring Initializr] and creating a new project. Doing so automatically generates a new project structure so that you can start coding right away.

We recommend experimenting with the Spring Initializr to become familiar with it.

[[getting-started-creating-project]]
=== Creating the Spring Task Project using Spring Initializr

Now we can create and test an application that prints 'Hello, World!' to the console.

To do so:

2. Create a new Maven project with a *Group* name of `io.spring.demo` and an *Artifact* name of `helloworld`.
3. In the Dependencies text box, type `task` and then select the `Cloud Task` dependency.
4. In the Dependencies text box, type `jdbc` and then select the `JDBC` dependency.
5. In the Dependencies text box, type `h2` and then select the `H2` (or your favorite database).
6. Click the *Generate Project* button.
7. Unzip the helloworld.zip file and import the project into your favorite IDE.

[[getting-started-writing-the-code]]
=== Writing the Code
To finish our application, we need to update the generated `HelloworldApplication` with the following contents so that it launches a Task.

```java
package io.spring.demo.helloworld;

import org.springframework.boot.CommandLineRunner;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.annotation.Bean;

@SpringBootApplication @EnableTask public class HelloworldApplication {

    @Bean
    public CommandLineRunner commandLineRunner() {
        return new HelloWorldCommandLineRunner();
    }

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

    public static class HelloWorldCommandLineRunner implements CommandLineRunner {

        @Override
        public void run(String... strings) throws Exception {
            System.out.println("Hello, World!");
        }
    }
}
While it may seem small, quite a bit is going on. For more about Spring Boot specifics, see the https://docs.spring.io/spring-boot/docs/current/reference/html/[Spring Boot reference documentation].

Now we can open the `application.properties` file in `src/main/resources`. We need to configure two properties in `application.properties`:

* `application.name`: To set the application name (which is translated to the task name)
* `logging.level`: To set the logging for Spring Cloud Task to `DEBUG` in order to get a view of what is going on.

The following example shows how to do both:

```
logging.level.org.springframework.cloud.task=DEBUG spring.application.name=helloWorld
```

`logging.level.org.springframework.cloud.task=DEBUG spring.application.name=helloWorld`

When including Spring Cloud Task Starter dependency, Task auto configures all beans to bootstrap its functionality.

Part of this configuration registers the `TaskRepository` and the infrastructure for its use.

In our demo, the `TaskRepository` uses an embedded H2 database to record the results of a task. This H2 embedded database is not a practical solution for a production environment, since the H2 DB goes away once the task ends. However, for a quick getting-started experience, we can use this in our example as well as echoing to the logs what is being updated in that repository. In the <<features-configuration>> section (later in this documentation), we cover how to customize the configuration of the pieces provided by Spring Cloud Task.

When our sample application runs, Spring Boot launches our `HelloWorldCommandLineRunner` and outputs our "Hello, World!" message to standard out. The `TaskLifecycleListener` records the start of the task and the end of the task in the repository.

The main method serves as the entry point to any java application. Our main method delegates to Spring Boot's https://docs.spring.io/spring-boot/docs/current/reference/html/boot-features-spring-
application.html

[getting-started-clr]

== The CommandLineRunner

Spring includes many ways to bootstrap an application's logic. Spring Boot provides a convenient method of doing so in an organized manner through its `*Runner` interfaces (`CommandLineRunner` or `ApplicationRunner`). A well behaved task can bootstrap any logic by using one of these two runners.

The lifecycle of a task is considered from before the `*Runner#run` methods are executed to once they are all complete. Spring Boot lets an application use multiple `*Runner` implementations, as does Spring Cloud Task.

NOTE: Any processing bootstrapped from mechanisms other than a `CommandLineRunner` or `ApplicationRunner` (by using `InitializingBean#afterPropertiesSet` for example) is not recorded by Spring Cloud Task.

[getting-started-running-the-example]

=== Running the Example

At this point, our application should work. Since this application is Spring Boot-based, we can run it from the command line by using `$ mvn spring-boot:run` from the root of our application, as shown (with its output) in the following example:

[source]

```
$ mvn clean spring-boot:run ........... (Maven log output here) ...........
```

1. _ _ _ _ /\ / '_ / _ _ _ _ \ \ / ( ) _ \
   \ \ \ \ \ \ \ \ / __ |
   | | | | | | | | | | | |
   \_ \_ \_ \_ __/\ / __|

:: Spring Boot

(v2.0.3.RELEASE)

The preceding output has three lines that of interest to us here:

* `SimpleTaskRepository` logged the creation of the entry in the `TaskRepository`.
* The execution of our `CommandLineRunner`, demonstrated by the "Hello, World!"
  output.
* `SimpleTaskRepository` logs the completion of the task in the `TaskRepository`.

NOTE: A simple task application can be found in the samples module of the Spring Cloud Task Project

https://github.com/spring-cloud/spring-cloud-task/tree/master/spring-cloud-task-samples/timestamp[here].

[[features]]

= Features

[[partintro]]

--

This section goes into more detail about Spring Cloud Task, including how to use it, how to configure it, and the appropriate extension points.

--

[[features-lifecycle]]
The lifecycle of a Spring Cloud Task

In most cases, the modern cloud environment is designed around the execution of processes that are not expected to end. If they do end, they are typically restarted. While most platforms do have some way to run a process that is not restarted when it ends, the results of that run are typically not maintained in a consumable way. Spring Cloud Task offers the ability to execute short-lived processes in an environment and record the results. Doing so allows for a microservices architecture around short-lived processes as well as longer running services through the integration of tasks by messages.

While this functionality is useful in a cloud environment, the same issues can arise in a traditional deployment model as well. When running Spring Boot applications with a scheduler such as cron, it can be useful to be able to monitor the results of the application after its completion.

Spring Cloud Task takes the approach that a Spring Boot application can have a start and an end and still be successful. Batch applications are one example of how processes that are expected to end (and that are often short-lived) can be helpful.

Spring Cloud Task records the lifecycle events of a given task. Most long-running processes, typified by most web applications, do not save their lifecycle events. The tasks at the heart of Spring Cloud Task do.

The lifecycle consists of a single task execution. This is a physical execution of a Spring Boot application configured to be a task (that is, it has the Sprint Cloud Task dependencies).

At the beginning of a task, before any `CommandLineRunner` or `ApplicationRunner` implementations have been run, an entry in the `TaskRepository` that records the start event is created. This event is triggered through `SmartLifecycle#start` being triggered by the Spring Framework. This indicates to the system that all beans are ready for use and comes before running any of the `CommandLineRunner` or `ApplicationRunner` implementations provided by Spring Boot.

NOTE: The recording of a task only occurs upon the successful bootstrapping of an `ApplicationContext`. If the context fails to bootstrap at all, the task’s run is not recorded.

Upon completion of all of the `*Runner#run` calls from Spring Boot or the failure of an `ApplicationContext` (indicated by an `ApplicationFailedEvent`), the task execution is updated in the repository with the results.
NOTE: If the application requires the 'ApplicationContext' to be closed at the completion of a task (all '*Runner#run' methods have been called and the task repository has been updated), set the property 'spring.cloud.task.closecontextEnabled' to true.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>'executionid'</td>
<td>The unique ID for the task's run.</td>
</tr>
<tr>
<td>'exitCode'</td>
<td>The exit code generated from an 'ExitCodeExceptionMapper' implementation. If there is no exit code generated but an 'ApplicationFailedEvent' is thrown, 1 is set. Otherwise, it is assumed to be 0.</td>
</tr>
<tr>
<td>'taskId'</td>
<td>The name for the task, as determined by the configured 'TaskNameResolver'.</td>
</tr>
<tr>
<td>'startTime'</td>
<td>The time the task was started, as indicated by the 'SmartLifecycle#start' call.</td>
</tr>
<tr>
<td>'endTime'</td>
<td>The time the task was completed, as indicated by the 'ApplicationReadyEvent'.</td>
</tr>
<tr>
<td>'exitMessage'</td>
<td>Any information available at the time of exit. This can programmatically be set by a 'TaskExecutionListener'.</td>
</tr>
<tr>
<td>'errorMessage'</td>
<td>If an exception is the cause of the end of the task (as indicated by an 'ApplicationFailedEvent'), the stack trace for that exception is stored here.</td>
</tr>
<tr>
<td>'arguments'</td>
<td>A 'List' of the string command line arguments as they were passed into the executable boot application.</td>
</tr>
</tbody>
</table>
When a task completes, it tries to return an exit code to the OS. If we take a look at our <<getting-started-developing-first-task,original example>>, we can see that we are not controlling that aspect of our application. So, if an exception is thrown, the JVM returns a code that may or may not be of any use to you in debugging.

Consequently, Spring Boot provides an interface, `ExitCodeExceptionMapper`, that lets you map uncaught exceptions to exit codes. Doing so lets you indicate, at the level of exit codes, what went wrong. Also, by mapping exit codes in this manner, Spring Cloud Task records the returned exit code.

If the task terminates with a SIG-INT or a SIG-TERM, the exit code is zero unless otherwise specified within the code.

NOTE: While the task is running, the exit code is stored as a null in the repository. Once the task completes, the appropriate exit code is stored based on the guidelines described earlier in this section.

[[features-configuration]]

== Configuration

Spring Cloud Task provides a ready-to-use configuration, as defined in the `DefaultTaskConfigurer` and `SimpleTaskConfiguration` classes. This section walks through the defaults and how to customize Spring Cloud Task for your needs.

[[features-data-source]]

=== DataSource

Spring Cloud Task uses a datasource for storing the results of task executions. By default, we provide an in-memory instance of H2 to provide a simple method of bootstrapping development. However, in a production environment, you probably want to configure your own `DataSource`.

If your application uses only a single `DataSource` and that serves as both your business schema and the task repository, all you need to do is provide any `DataSource` (the easiest way to do so is through Spring Boot's configuration conventions). This `DataSource` is automatically used by Spring Cloud Task for the repository.

If your application uses more than one `DataSource`, you need to configure the task repository with the appropriate `DataSource`. This customization can be done through an implementation of `TaskConfigurer`.

[[features-table-prefix]]

=== Table Prefix

One modifiable property of `TaskRepository` is the table prefix for the task tables.
By default, they are all prefaced with 'TASK_'. 'TASK_EXECUTION' and 'TASK_EXECUTION_PARAMS' are two examples. However, there are potential reasons to modify this prefix. If the schema name needs to be prepended to the table names or if more than one set of task tables is needed within the same schema, you must change the table prefix. You can do so by setting the `spring.cloud.task.tablePrefix` to the prefix you need, as follows:

```
'spring.cloud.task.tablePrefix=yourPrefix'
```

By using the `spring.cloud.task.tablePrefix`, a user assumes the responsibility to create the task tables that meet both the criteria for the task table schema but with modifications that are required for a user's business needs.

You can utilize the Spring Cloud Task Schema DDL as a guide when creating your own Task DDL as seen [here](https://github.com/spring-cloud/spring-cloud-task/tree/master/spring-cloud-task-core/src/main/resources/org/springframework/cloud/task).

[[features-table-initialization]]

--- Enable/Disable table initialization

In cases where you are creating the task tables and do not wish for Spring Cloud Task to create them at task startup, set the `spring.cloud.task.initialize-enabled` property to `false`, as follows:

```
'spring.cloud.task.initialize-enabled=false'
```

It defaults to `true`.

NOTE: The property `spring.cloud.task.initialize.enable` has been deprecated.

[[features-generated_task_id]]

--- Externally Generated Task ID

In some cases, you may want to allow for the time difference between when a task is requested and when the infrastructure actually launches it. Spring Cloud Task lets you create a 'TaskExecution' when the task is requested. Then pass the execution ID of the generated 'TaskExecution' to the task so that it can update the 'TaskExecution' through the task's lifecycle.

A 'TaskExecution' can be created by calling the `createTaskExecution` method on an implementation of the 'TaskRepository' that references the datastore that holds the 'TaskExecution' objects.

In order to configure your Task to use a generated 'TaskExecutionId', add the following property:
Spring Cloud Task lets you store an external task ID for each `TaskExecution`. An example of this would be a task ID provided by Cloud Foundry when a task is launched on the platform. In order to configure your Task to use a generated `TaskExecutionId`, add the following property:

`spring.cloud.task.executionid=yourtaskId`

Spring Cloud Task lets you store a parent task ID for each `TaskExecution`. An example of this would be a task that executes another task or tasks and you want to record which task launched each of the child tasks. In order to configure your Task to set a parent `TaskExecutionId` add the following property on the child task:

`spring.cloud.task.parent-execution-id=<parentExecutionTaskId>`

The `TaskConfigurer` is a strategy interface that lets you customize the way components of Spring Cloud Task are configured. By default, we provide the `DefaultTaskConfigurer` that provides logical defaults: `Map`-based in-memory components (useful for development if no `DataSource` is provided) and JDBC based components (useful if there is a `DataSource` available).

The `TaskConfigurer` lets you configure three main components:

<table>
<thead>
<tr>
<th>Component</th>
<th>Description</th>
<th>Default (provided by <code>DefaultTaskConfigurer</code>)</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>TaskRepository</code></td>
<td>The implementation of the <code>TaskRepository</code> to be used.</td>
<td><code>SimpleTaskRepository</code></td>
</tr>
<tr>
<td><code>TaskExplorer</code></td>
<td>The implementation of the <code>TaskExplorer</code> (a component for read-only access to the task repository) to be used.</td>
<td><code>SimpleTaskExplorer</code></td>
</tr>
</tbody>
</table>
You can customize any of the components described in the preceding table by creating a custom implementation of the `TaskConfigurer` interface. Typically, extending the `DefaultTaskConfigurer` (which is provided if a `TaskConfigurer` is not found) and overriding the required getter is sufficient. However, implementing your own from scratch may be required.

NOTE: Users should not directly use getter methods from a `TaskConfigurer` directly unless they are using it to supply implementations to be exposed as Spring Beans.

### Task Name

In most cases, the name of the task is the application name as configured in Spring Boot. However, there are some cases where you may want to map the run of a task to a different name. Spring Cloud Data Flow is an example of this (because you probably want the task to be run with the name of the task definition). Because of this, we offer the ability to customize how the task is named, through the `TaskNameResolver` interface.

By default, Spring Cloud Task provides the `SimpleTaskNameResolver`, which uses the following options (in order of precedence):

1. A Spring Boot property (configured in any of the ways Spring Boot allows) called `spring.cloud.task.name`.
2. The application name as resolved using Spring Boot's rules (obtained through `ApplicationContext#getId`).

### Task Execution Listener

`TaskExecutionListener` lets you register listeners for specific events that occur during the task lifecycle. To do so, create a class that implements the `TaskExecutionListener` interface. The class that implements the `TaskExecutionListener` interface is notified of the following events:

* `onTaskStartup`: Prior to storing the `TaskExecution` into the `TaskRepository`.
* `onTaskEnd`: Prior to updating the `TaskExecution` entry in the `TaskRepository` and marking the final state of the task.
* `onTaskFailed`: Prior to the `onTaskEnd` method being invoked when an unhandled exception is thrown by the task.
Spring Cloud Task also lets you add `TaskExecution` Listeners to methods within a bean by using the following method annotations:

* `@BeforeTask`: Prior to the storing the `TaskExecution` into the `TaskRepository`
* `@AfterTask`: Prior to the updating of the `TaskExecution` entry in the `TaskRepository` marking the final state of the task.
* `@FailedTask`: Prior to the `@AfterTask` method being invoked when an unhandled exception is thrown by the task.

The following example shows the three annotations in use:

```java
public class MyBean {

@BeforeTask
public void methodA(TaskExecution taskExecution) {
}

@AfterTask
public void methodB(TaskExecution taskExecution) {
}

@FailedTask
public void methodC(TaskExecution taskExecution, Throwable throwable) {
}
}
```

NOTE: Inserting an `ApplicationListener` earlier in the chain than `TaskLifecycleListener` exists may cause unexpected effects.

[[features-task-execution-listener-Exceptions]]

Exceptions Thrown by Task Execution Listener

If an exception is thrown by a `TaskExecutionListener` event handler, all listener processing for that event handler stops. For example, if three `onTaskStartup` listeners have started and the first `onTaskStartup` event handler throws an exception, the other two `onTaskStartup` methods are not called. However, the other event handlers (`onTaskEnd` and `onTaskFailed`) for the `TaskExecutionListeners` are called.

The exit code returned when a exception is thrown by a `TaskExecutionListener`
event handler is the exit code that was reported by the
If no `ExitCodeEvent` is emitted, the Exception thrown is evaluated to see
if it is of type
https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-application-exit[ExitCodeGenerator].
If so, it returns the exit code from the `ExitCodeGenerator`. Otherwise, `1` is returned.

In the case that an exception is thrown in an `onTaskStartup` method, the exit code
for the application will be `1`.
If an exception is thrown in either a `onTaskEnd` or `onTaskFailed`
method, the exit code for the application will be the one established using the rules
enumerated above.

NOTE: In the case of an exception being thrown in a `onTaskStartup`, `onTaskEnd`, or
`onTaskFailed`
you can not override the exit code for the application using
`ExitCodeExceptionMapper`.

[[features-task-execution-listener-exit-messages]]
==== Exit Messages

You can set the exit message for a task programmatically by using a
`TaskExecutionListener`. This is done by setting the `TaskExecution's` `exitMessage`,
which then gets passed into the `TaskExecutionListener`. The following example shows
a method that is annotated with the `@AfterTask` `ExecutionListener`:

[source,java]
@AfterTask
public void afterMe(TaskExecution taskExecution) {
    taskExecution.setExitMessage("AFTER EXIT MESSAGE");
}

An `ExitMessage` can be set at any of the listener events (`onTaskStartup`,
`onTaskFailed`, and `onTaskEnd`). The order of precedence for the three listeners follows:

. `onTaskEnd`
. `onTaskFailed`
. `onTaskStartup`

For example, if you set an `exitMessage` for the `onTaskStartup` and `onTaskFailed`
listeners and the task ends without failing, the `exitMessage` from the
`onTaskStartup`
is stored in the repository. Otherwise, if a failure occurs, the `exitMessage` from
the `onTaskFailed` is stored. Also if you set the `exitMessage` with an
`onTaskEnd` listener, the `exitMessage` from the `onTaskEnd` supersedes
Restricting Spring Cloud Task Instances

Spring Cloud Task lets you establish that only one task with a given task name can be run at a time. To do so, you need to establish the `<features-task-name, task name>` and set `spring.cloud.task.single-instance-enabled=true` for each task execution. While the first task execution is running, any other time you try to run a task with the same `<features-task-name, task name>` and `spring.cloud.task.single-instance-enabled=true`, the task fails with the following error message: 'Task with name "application" is already running.' The default value for `spring.cloud.task.single-instance-enabled` is `false`. The following example shows how to set `spring.cloud.task.single-instance-enabled` to `true`:

'spring.cloud.task.single-instance-enabled=true or false'

To use this feature, you must add the following Spring Integration dependencies to your application:

[source,xml]
<dependency>
  <groupId>org.springframework.integration</groupId>
  <artifactId>spring-integration-core</artifactId>
</dependency>
<dependency>
  <groupId>org.springframework.integration</groupId>
  <artifactId>spring-integration-jdbc</artifactId>
</dependency>

NOTE: The exit code for the application will be 1 if the task fails because this feature is enabled and another task is running with the same task name.

Disabling Spring Cloud Task Auto Configuration

In cases where Spring Cloud Task should not be auto configured for an implementation, you can disable Task’s auto configuration. This can be done either by adding the following annotation to your Task application:

```
@EnableAutoConfiguration(exclude={SimpleTaskAutoConfiguration.class})
```

You may also disable Task auto configuration by setting the `spring.cloud.task.autoconfiguration.enabled` property to `false`.
If the application requires the `ApplicationContext` to be closed at the completion of a task (all `*Runner#run` methods have been called and the task repository has been updated), set the property `spring.cloud.task.closecontextEnabled` to `true`.

Another case to close the context is when the Task Execution completes however the application does not terminate. In these cases the context is held open because a thread has been allocated (for example: if you are using a TaskExecutor). In these cases set the `spring.cloud.task.closecontextEnabled` property to `true` when launching your task.

This will close the application's context once the task is complete. Thus allowing the application to terminate.

Spring Boot provides facilities for the execution of batch jobs within an über-jar. Spring Boot's support of this functionality lets a developer execute multiple batch jobs within that execution. Spring Cloud Task provides the ability to associate the execution of a job (a job execution) with a task's execution so that one can be traced back to the other.

Spring Cloud Task achieves this functionality by using the `TaskBatchExecutionListener`.

By default, this listener is auto configured in any context that has both a Spring Batch Job configured (by having a bean of type `Job` defined in the context) and the `spring-cloud-task-batch` jar on the classpath. The listener is injected into all jobs that meet those conditions.
To prevent the listener from being injected into any batch jobs within the current context, you can disable the autoconfiguration by using standard Spring Boot mechanisms.

To only have the listener injected into particular jobs within the context, override the `batchTaskExecutionListenerBeanPostProcessor` and provide a list of job bean IDs, as shown in the following example:

``java
public TaskBatchExecutionListenerBeanPostProcessor batchTaskExecutionListenerBeanPostProcessor() { TaskBatchExecutionListenerBeanPostProcessor postProcessor = new TaskBatchExecutionListenerBeanPostProcessor();

    postProcessor.setJobNames(Arrays.asList(new String[] { "job1", "job2" }));

    return postProcessor;
}
```
NOTE: You can find a sample batch application in the samples module of the Spring Cloud Task Project, https://github.com/spring-cloud/spring-cloud-task/tree/master/spring-cloud-task-samples/batch-job[here].

[[batch-partitioning]]
== Remote Partitioning

Spring Cloud Deployer provides facilities for launching Spring Boot-based applications on most cloud infrastructures. The `DeployerPartitionHandler` and `DeployerStepExecutionHandler` delegate the launching of worker step executions to Spring Cloud Deployer.

To configure the `DeployerStepExecutionHandler`, you must provide a `Resource` representing the Spring Boot über-jar to be executed, a `TaskLauncher`, and a `JobExplorer`. You can configure any environment properties as well as the max number of workers to be executing at once, the interval to poll for the results (defaults to 10 seconds), and a timeout (defaults to -1 or no timeout). The following example shows how configuring this `PartitionHandler` might look:

[source,java]

```java
@Bean public PartitionHandler partitionHandler(TaskLauncher taskLauncher, JobExplorer jobExplorer) throws Exception {

    MavenProperties mavenProperties = new MavenProperties();
    mavenProperties.setRemoteRepositories(new HashMap<>(Collections.singletonMap("springRepo", new MavenProperties.RemoteRepository(repository))));

    Resource resource = MavenResource.parse(String.format("%s:%s:%s",
            "io.spring.cloud",
            "partitioned-batch-job",
            "1.1.0.RELEASE"), mavenProperties);

    DeployerPartitionHandler partitionHandler = new DeployerPartitionHandler(taskLauncher, jobExplorer, resource, "workerStep");
```
List<String> commandLineArgs = new ArrayList<>(3);
commandLineArgs.add("--spring.profiles.active=worker");
commandLineArgs.add("--spring.cloud.task.initialize.enable=false");
commandLineArgs.add("--spring.batch.initializer.enabled=false");

partitionHandler.setCommandLineArgsProvider(
    new PassThroughCommandLineArgsProvider(commandLineArgs));
partitionHandler.setEnvironmentVariablesProvider(new NoOpEnvironmentVariablesProvider());
partitionHandler.setMaxWorkers(2);
partitionHandler.setApplicationName("PartitionedBatchJobTask");

return partitionHandler;

NOTE: When passing environment variables to partitions, each partition may
be on a different machine with different environment settings.
Consequently, you should pass only those environment variables that are required.

Notice in the example above that we have set the maximum number of workers to 2.
Setting the maximum of workers establishes the maximum number of partitions that should be running at one time.

The 'Resource' to be executed is expected to be a Spring Boot über-jar with a
'DeployerStepExecutionHandler' configured as a 'CommandLineRunner' in the current context.
The repository enumerated in the preceding example should be the remote repository in
which the über-jar is located. Both the manager and worker are expected to have visibility
into the same data store being used as the job repository and task repository. Once the
underlying infrastructure has bootstrapped the Spring Boot jar and Spring Boot has launched the 'DeployerStepExecutionHandler', the step handler executes the requested 'Step'. The following example shows how to configure the 'DeployerStepExecutionHandler':

[source,java]

@Bean public DeployerStepExecutionHandler stepExecutionHandler(JobExplorer jobExplorer) {
    DeployerStepExecutionHandler handler = new DeployerStepExecutionHandler(this.context,
        jobExplorer, this.jobRepository);

    return handler;
}
NOTE: You can find a sample remote partition application in the samples module of the Spring Cloud Task project, https://github.com/spring-cloud/spring-cloud-task/tree/master/spring-cloud-task-samples/partitioned-batch-job.

=== Notes on Developing a Batch-partitioned application for the Kubernetes Platform

* When deploying partitioned apps on the Kubernetes platform, you must use the following dependency for the Spring Cloud Kubernetes Deployer:

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-deployer-kubernetes</artifactId>
</dependency>
```

* The application name for the task application and its partitions need to follow the following regex pattern: `^[a-z0-9]([-a-z0-9]*[a-z0-9])$`. Otherwise, an exception is thrown.

=== Notes on Developing a Batch-partitioned Application for the Cloud Foundry Platform

* When deploying partitioned apps on the Cloud Foundry platform, you must use the following dependencies for the Spring Cloud Foundry Deployer:

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-deployer-cloudfoundry</artifactId>
</dependency>
<dependency>
  <groupId>io.projectreactor</groupId>
  <artifactId>reactor-core</artifactId>
  <version>3.1.5.RELEASE</version>
</dependency>
<dependency>
  <groupId>io.projectreactor.ipc</groupId>
  <artifactId>reactor-netty</artifactId>
  <version>0.7.5.RELEASE</version>
</dependency>
```
* When configuring the partition handler, Cloud Foundry Deployment environment variables need to be established so that the partition handler can start the partitions. The following list shows the required environment variables:
  - `spring_cloud_deployer_cloudfoundry_url`
  - `spring_cloud_deployer_cloudfoundry_org`
  - `spring_cloud_deployer_cloudfoundry_space`
  - `spring_cloud_deployer_cloudfoundry_domain`
  - `spring_cloud_deployer_cloudfoundry_username`
  - `spring_cloud_deployer_cloudfoundry_password`
  - `spring_cloud_deployer_cloudfoundry_services`
  - `spring_cloud_deployer_cloudfoundry_taskTimeout`

An example set of deployment environment variables for a partitioned task that uses a `mysql` database service might resemble the following:

```bash
spring_cloud_deployer_cloudfoundry_url=https://api.local.pcfdev.io
spring_cloud_deployer_cloudfoundry_org=pcfdev-org
spring_cloud_deployer_cloudfoundry_space=pcfdev-space
spring_cloud_deployer_cloudfoundry_domain=local.pcfdev.io
spring_cloud_deployer_cloudfoundry_username=admin
spring_cloud_deployer_cloudfoundry_password=admin
spring_cloud_deployer_cloudfoundry_services=mysql
spring_cloud_deployer_cloudfoundry_taskTimeout=300
```

NOTE: When using PCF-Dev, the following environment variable is also required:
`spring_cloud_deployer_cloudfoundry_skipSslValidation=true`

== Batch Informational Messages

Spring Cloud Task provides the ability for batch jobs to emit informational messages. The ""\<<stream.adoc#stream-integration-batch-events>>" section covers this feature in detail.

== Batch Job Exit Codes

As discussed <<features.adoc#features-lifecycle-exit-codes,earlier>>, Spring Cloud Task applications support the ability to record the exit code of a task execution. However, in cases where you run a Spring Batch Job within a task, regardless of how the Batch Job Execution completes, the result of the task is always zero when using the default Batch/Boot behavior. Keep in mind that a task is a boot application and that the exit code
returned from the task is the same as a boot application. To override this behavior and allow the task to return an exit code other than zero when a batch job returns an
https://docs.spring.io/spring-batch/4.0.x/reference/html/step.html#batchStatusVsExitStatus[BatchStatus] of 'FAILED', set 'spring.cloud.task.batch.fail-on-job-failure' to 'true'. Then the exit code
can be 1 (the default) or be based on the

This functionality uses a new 'CommandLineRunner' that replaces the one provided by Spring Boot. By default, it is configured with the same order. However, if you want to customize the order in which the 'CommandLineRunner' is run, you can set its order by setting the 'spring.cloud.task.batch.commandLineRunnerOrder' property. To have your task return the exit code based on the result of the batch job execution, you need to write your own 'CommandLineRunner'.

//TODO Great place for a example showing how a custom CommandLineRunner

[[batch-job-starter]]
= Single Step Batch Job Starter

[[partintro]]
--
This section goes into how to develop a Spring Batch 'Job' with a single 'Step' by using the starter included in Spring Cloud Task. This starter lets you use configuration to define an 'ItemReader', an 'ItemWriter', or a full single-step Spring Batch 'Job'. For more about Spring Batch and its capabilities, see the https://spring.io/projects/spring-batch[Spring Batch documentation].

--

To obtain the starter for Maven, add the following to your build:

====
[source,xml]

<dependency>  
  <groupId>org.springframework.cloud</groupId>  
  <artifactId>spring-cloud-starter-single-step-batch-job</artifactId>  
  <version>2.3.0</version> 
</dependency>
To obtain the starter for Gradle, add the following to your build:

```groovy
compile "org.springframework.cloud:spring-cloud-starter-single-step-batch-job:2.3.0"
```

---

#### Defining a Job

You can use the starter to define as little as an `ItemReader` or an `ItemWriter` or as much as a full `Job`. In this section, we define which properties are required to be defined to configure a `Job`.

#### Properties

To begin, the starter provides a set of properties that let you configure the basics of a `Job` with one `Step`:

**.Job Properties**

<table>
<thead>
<tr>
<th>Property</th>
<th>Type</th>
<th>Default Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>spring.batch.job.jobName</code></td>
<td><code>String</code></td>
<td><code>null</code></td>
<td>The name of the job.</td>
</tr>
<tr>
<td><code>spring.batch.job.stepName</code></td>
<td><code>String</code></td>
<td><code>null</code></td>
<td>The name of the step.</td>
</tr>
<tr>
<td><code>spring.batch.job.chunkSize</code></td>
<td><code>Integer</code></td>
<td><code>null</code></td>
<td>The number of items to be processed per transaction.</td>
</tr>
</tbody>
</table>

With the above properties configured, you have a `Job` with a single, chunk-based step. This chunk-based step reads, processes, and writes `Map<String, Object>` instances as the items. However, the step does not yet do anything. You need to configure an
ItemReader, an optional ItemProcessor, and an ItemWriter to give it something to do. To configure one of these, you can either use properties and configure one of the options that has provided autoconfiguration or you can configure your own with the standard Spring configuration mechanisms.

NOTE: If you configure your own, the input and output types must match the others in the step.
The ItemReader implementations and ItemWriter implementations in this starter all use a Map<String, Object> as the input and the output item.

[[item-readers]]
== Autoconfiguration for ItemReader Implementations

This starter provides autoconfiguration for four different ItemReader implementations:
AmqpItemReader, FlatFileItemReader, JdbcCursorItemReader, and KafkaItemReader.
In this section, we outline how to configure each of these by using the provided autoconfiguration.

[[amqpitemreader]]
=== AmqpItemReader

You can read from a queue or topic with AMQP by using the AmqpItemReader. The autoconfiguration for this ItemReader implementation is dependent upon two sets of configuration. The first is the configuration of an AmqpTemplate. You can either configure this yourself or use the autoconfiguration provided by Spring Boot. See the https://docs.spring.io/spring-boot/docs/2.4.x/reference/htmlsingle/#boot-features-amqp [Spring Boot AMQP documentation].
Once you have configured the AmqpTemplate, you can enable the batch capabilities to support it by setting the following properties:

<table>
<thead>
<tr>
<th>AmqpItemReader Properties</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>spring.batch.job.amqpitemreader.enabled</code></td>
</tr>
<tr>
<td><code>spring.batch.job.amqpitemreader.jsonConverterEnabled</code></td>
</tr>
</tbody>
</table>
For more information, see the https://docs.spring.io/spring-batch/docs/4.3.x/api/org/springframework/batch/item/amqp/AmqpItemReader.html[`AmqpItemReader` documentation].

`FlatFileItemReader` lets you read from flat files (such as CSVs and other file formats). To read from a file, you can provide some components yourself through normal Spring configuration (`LineTokenizer`, `RecordSeparatorPolicy`, `FieldSetMapper`, `LineMapper`, or `SkippedLinesCallback`). You can also use the following properties to configure the reader:

<table>
<thead>
<tr>
<th>Property</th>
<th>Type</th>
<th>Default Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>spring.batch.job.flatfileitemreader.saveState</code></td>
<td><code>boolean</code></td>
<td><code>true</code></td>
<td>Determines if the state should be saved for restarts.</td>
</tr>
<tr>
<td><code>spring.batch.job.flatfileitemreader.name</code></td>
<td><code>String</code></td>
<td><code>null</code></td>
<td>Name used to provide unique keys in the <code>ExecutionContext</code>.</td>
</tr>
<tr>
<td><code>spring.batch.job.flatfileitemreader.maxItemcount</code></td>
<td><code>int</code></td>
<td><code>Integer.MAX_VALUE</code></td>
<td>Maximum number of items to be read from the file.</td>
</tr>
<tr>
<td><code>spring.batch.job.flatfileitemreader.currentItemCount</code></td>
<td><code>int</code></td>
<td><code>0</code></td>
<td>Number of items that have already been read. Used on restarts.</td>
</tr>
<tr>
<td><code>spring.batch.job.flatfileitemreader.comments</code></td>
<td><code>List&lt;String&gt;</code></td>
<td>empty List</td>
<td>A list of Strings that indicate commented lines (lines to be ignored) in the file.</td>
</tr>
<tr>
<td><code>spring.batch.job.flatfileitemreader.resource</code></td>
<td><code>Resource</code></td>
<td><code>null</code></td>
<td>The resource to be read.</td>
</tr>
<tr>
<td><code>spring.batch.job.flatfileitemreader.strict</code></td>
<td><code>boolean</code></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
If set to `true`, the reader throws an exception if the resource is not found.

- `spring.batch.job.flatfileitemreader.encoding`:
  - `String`
  - `FlatFileItemReader.DEFAULT_CHARSET`
  - Encoding to be used when reading the file.

- `spring.batch.job.flatfileitemreader.linesToSkip`:
  - `int`
  - `0`
  - Indicates the number of lines to skip at the start of a file.

- `spring.batch.job.flatfileitemreader.delimited`:
  - `boolean`
  - `false`
  - Indicates whether the file is a delimited file (CSV and other formats). Only one of this property or `spring.batch.job.flatfileitemreader.fixedLength` can be `true` at the same time.

- `spring.batch.job.flatfileitemreader.delimiter`:
  - `String`
  - `DelimitedLineTokenizer.DELIMITER_COMMA`
  - If reading a delimited file, indicates the delimiter to parse on.

- `spring.batch.job.flatfileitemreader.quoteCharacter`:
  - `char`
  - `DelimitedLineTokenizer.DEFAULT_QUOTE_CHARACTER`
  - Used to determine the character used to quote values.

- `spring.batch.job.flatfileitemreader.includedFields`:
  - `List<Integer>`
  - `empty list`
  - A list of indices to determine which fields in a record to include in the item.

- `spring.batch.job.flatfileitemreader.fixedLength`:
  - `boolean`
  - `false`
  - Indicates if a file's records are parsed by column numbers. Only one of this property or `spring.batch.job.flatfileitemreader.delimited` can be `true` at the same time.

- `spring.batch.job.flatfileitemreader.ranges`:
  - `List<Range>`
  - `empty list`
  - List of column ranges by which to parse a fixed width record. See the https://docs.spring.io/spring-batch/docs/4.3.x/api/org/springframework/batch/item/file/transform/Range.html[Range documentation].

- `spring.batch.job.flatfileitemreader.names`
List of names for each field parsed from a record. These names are the keys in the `Map<String, Object>` in the items returned from this `ItemReader`.

If set to `true`, the mapping fails if the fields cannot be mapped.

See the https://docs.spring.io/spring-batch/docs/4.3.x/api/org/springframework/batch/item/file/FlatFileItemReader.html[FlatFileItemReader documentation].

The `JdbcCursorItemReader` runs a query against a relational database and iterates over the resulting cursor (`ResultSet`) to provide the resulting items. This autoconfiguration lets you provide a `PreparedStatementSetter`, a `RowMapper`, or both. You can also use the following properties to configure a `JdbcCursorItemReader`:

<table>
<thead>
<tr>
<th>Property</th>
<th>Type</th>
<th>Default Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>saveState</code></td>
<td><code>boolean</code></td>
<td><code>true</code></td>
<td>Determines whether the state should be saved for restarts.</td>
</tr>
<tr>
<td><code>name</code></td>
<td><code>String</code></td>
<td><code>null</code></td>
<td>Name used to provide unique keys in the <code>ExecutionContext</code>.</td>
</tr>
<tr>
<td><code>maxItemCount</code></td>
<td><code>int</code></td>
<td><code>Integer.MAX_VALUE</code></td>
<td>Maximum number of items to be read from the file.</td>
</tr>
<tr>
<td><code>currentItemCount</code></td>
<td><code>int</code></td>
<td><code>0</code></td>
<td>Number of items that have already been read. Used on restarts.</td>
</tr>
<tr>
<td><code>fetchSize</code></td>
<td><code>int</code></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
A hint to the driver to indicate how many records to retrieve per call to the database system. For best performance, you usually want to set it to match the chunk size.

`spring.batch.job.jdbcCursorItemReader.maxRows`
`int`

Maximum number of items to read from the database.

`spring.batch.job.jdbcCursorItemReader.queryTimeout`
`int`

Number of milliseconds for the query to timeout.

`spring.batch.job.jdbcCursorItemReader.ignoreWarnings`
`boolean`
`true`
Determines whether the reader should ignore SQL warnings when processing.

`spring.batch.job.jdbcCursorItemReader.verifyCursorPosition`
`boolean`
`true`
Indicates whether the cursor's position should be verified after each read to verify that the `RowMapper` did not advance the cursor.

`spring.batch.job.jdbcCursorItemReader.driverSupportsAbsolute`
`boolean`
`false`
Indicates whether the driver supports absolute positioning of a cursor.

`spring.batch.job.jdbcCursorItemReader.useSharedExtendedConnection`
`boolean`
`false`
Indicates whether the connection is shared with other processing (and is therefore part of a transaction).

`spring.batch.job.jdbcCursorItemReader.sql`
`String`
`null`
SQL query from which to read.

See the https://docs.spring.io/spring-batch/docs/4.3.x/api/org/springframework/batch/item/database/JdbcCursorItemReader.html[`JdbcCursorItemReader` documentation].
To configure a `KafkaItemReader`, two pieces of configuration are required. First, configuring Kafka with Spring Boot's Kafka autoconfiguration is required (see the https://docs.spring.io/spring-boot/docs/2.4.x/reference/htmlsingle/#boot-features-kafka [Spring Boot Kafka documentation]). Once you have configured the Kafka properties from Spring Boot, you can configure the `KafkaItemReader` itself by setting the following properties:

<table>
<thead>
<tr>
<th><code>KafkaItemReader</code> Properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>Property</td>
</tr>
<tr>
<td><code>spring.batch.job.kafkaitemreader.name</code></td>
</tr>
<tr>
<td><code>spring.batch.job.kafkaitemreader.topic</code></td>
</tr>
<tr>
<td><code>spring.batch.job.kafkaitemreader.partitions</code></td>
</tr>
<tr>
<td><code>spring.batch.job.kafkaitemreader.pollTimeOutInSeconds</code></td>
</tr>
<tr>
<td><code>spring.batch.job.kafkaitemreader.saveState</code></td>
</tr>
</tbody>
</table>

See the https://docs.spring.io/spring-batch/docs/4.3.x/api/org/springframework/batch/item/kafka/KafkaItemReader.html [KafkaItemReader documentation].

[[item-processors]]

== ItemProcessor Configuration

The single-step batch job autoconfiguration accepts an `ItemProcessor` if one is available within the `ApplicationContext`. If one is found of the correct type (`ItemProcessor<Map<String, Object>, Map<String, Object>>`), it is autowired into the step.
== Autoconfiguration for ItemWriter implementations

This starter provides autoconfiguration for `ItemWriter` implementations that match the supported `ItemReader` implementations: `AmqpItemWriter`, `FlatFileItemWriter`, `JdbcItemWriter`, and `KafkaItemWriter`. This section covers how to use autoconfiguration to configure a supported `ItemWriter`.

=== AmqpItemWriter

To write to a RabbitMQ queue, you need two sets of configuration. First, you need an `AmqpTemplate`. The easiest way to get this is by using Spring Boot's RabbitMQ autoconfiguration. See the https://docs.spring.io/spring-boot/docs/2.4.x/reference/htmlsingle/#boot-features-amqp [Spring Boot RabbitMQ documentation].

Once you have configured the `AmqpTemplate`, you can configure the `AmqpItemWriter` by setting the following properties:

`.AmqpItemWriter` Properties
<table>
<thead>
<tr>
<th>Property</th>
<th>Type</th>
<th>Default Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>spring.batch.job.amqpitemwriter.enabled</code></td>
<td><code>boolean</code></td>
<td><code>false</code></td>
<td>If <code>true</code>, the autoconfiguration runs.</td>
</tr>
<tr>
<td><code>spring.batch.job.amqpitemwriter.jsonConverterEnabled</code></td>
<td><code>boolean</code></td>
<td><code>true</code></td>
<td>Indicates whether <code>Jackson2JsonMessageConverter</code> should be registered to convert messages.</td>
</tr>
</tbody>
</table>

=== FlatFileItemWriter

To write a file as the output of the step, you can configure `FlatFileItemWriter`. Autoconfiguration accepts components that have been explicitly configured (such as `LineAggregator`, `FieldExtractor`, `FlatFileHeaderCallback`, or a `FlatFileFooterCallback`) and components that have been configured by setting the following properties specified:

`.FlatFileItemWriter` Properties
<table>
<thead>
<tr>
<th>Property</th>
<th>Type</th>
<th>Default Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>spring.batch.job.flatfileitemwriter.resource</code></td>
<td><code>Resource</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>null</code></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>The resource to be read.</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| `spring.batch.job.flatfileitemwriter.delimited` |
| `boolean` |
| `false` |
| Indicates whether the output file is a delimited file. If `true`, `spring.batch.job.flatfileitemwriter.formatted` must be `false`. |

| `spring.batch.job.flatfileitemwriter.formatted` |
| `boolean` |
| `false` |
| Indicates whether the output file is a formatted file. If `true`, `spring.batch.job.flatfileitemwriter.delimited` must be `false`. |

| `spring.batch.job.flatfileitemwriter.format` |
| `String` |
| `null` |
| The format used to generate the output for a formatted file. The formatting is performed by using `String.format`. |

| `spring.batch.job.flatfileitemwriter.locale` |
| `Locale` |
| `Locale.getDefault()` |
| The `Locale` to be used when generating the file. |

| `spring.batch.job.flatfileitemwriter.maximumLength` |
| `int` |
| `0` |
| Max length of the record. If 0, the size is unbounded. |

| `spring.batch.job.flatfileitemwriter.minimumLength` |
| `int` |
| `0` |
| The minimum record length. |

| `spring.batch.job.flatfileitemwriter.delimiter` |
| `String` |
| `\`, `\` |
| The `String` used to delimit fields in a delimited file. |

| `spring.batch.job.flatfileitemwriter.encoding` |
| `String` |
| `FlatFileItemReader.DEFAULT_CHARSET` |
| Encoding to use when writing the file. |

| `spring.batch.job.flatfileitemwriter.forceSync` |
| `boolean` |
| `false` |
| Indicates whether a file should be force-synced to the disk on flush. |
List of names for each field parsed from a record. These names are the keys in the `Map<String, Object>` for the items received by this `ItemWriter`.

Indicates whether a file should be appended to if the output file is found.

What `String` to use to separate lines in the output file.

Name used to provide unique keys in the `ExecutionContext`.

Determines whether the state should be saved for restarts.

If set to `true`, an empty file (there is no output) is deleted when the job completes.

If set to `true` and a file is found where the output file should be, it is deleted before the step begins.

Indicates whether the reader is a transactional queue (indicating that the items read are returned to the queue upon a failure).

See the [https://docs.spring.io/spring-batch/docs/4.3.x/api/org/springframework/batch/item/file/FlatFileItemWriter.html][FlatFileItemWriter documentation].

[[jdbcitemwriter]]

--- JdbcBatchItemWriter
To write the output of a step to a relational database, this starter provides the ability to autoconfigure a `JdbcBatchItemWriter`. The autoconfiguration lets you provide your own `ItemPreparedStatementSetter` or `ItemSqlParameterSourceProvider` and configuration options by setting the following properties:

```
`JdbcBatchItemWriter` Properties
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Property</td>
</tr>
<tr>
<td>---</td>
</tr>
<tr>
<td><code>spring.batch.job.jdbcbatchitemwriter.name</code></td>
</tr>
<tr>
<td><code>spring.batch.job.jdbcbatchitemwriter.sql</code></td>
</tr>
<tr>
<td><code>spring.batch.job.jdbcbatchitemwriter.assertUpdates</code></td>
</tr>
</tbody>
</table>

See the https://docs.spring.io/spring-batch/docs/4.3.x/api/org/springframework/batch/item/database/JdbcBatchItemWriter.html[`JdbcBatchItemWriter` documentation].
```

[[kafkaitemwriter]]

### KafkaItemWriter

To write step output to a Kafka topic, you need `KafkaItemWriter`. This starter provides autoconfiguration for a `KafkaItemWriter` by using facilities from two places. The first is Spring Boot's Kafka autoconfiguration. (See the https://docs.spring.io/spring-boot/docs/2.4.x/reference/htmlsingle/#boot-features-kafka[Spring Boot Kafka documentation].) Second, this starter lets you configure two properties on the writer.

```
`KafkaItemWriter` Properties
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Property</td>
</tr>
<tr>
<td>---</td>
</tr>
<tr>
<td><code>spring.batch.job.kafkaitemwriter.topic</code></td>
</tr>
</tbody>
</table>
```
| `spring.batch.job.kafkaitemwriter.delete` |
| `boolean` |
| `false` |
| Whether the items being passed to the writer are all to be sent as delete events to the topic. |

For more about the configuration options for the `KafkaItemWriter`, see the https://docs.spring.io/spring-batch/docs/4.3.x/api/org/springframework/batch/item/kafka/KafkaItemWriter.html[`KafkaItemWriter` documentation].

[[stream-integration]]
= Spring Cloud Stream Integration

[[partintro]]
--
A task by itself can be useful, but integration of a task into a larger ecosystem lets it be useful for more complex processing and orchestration. This section covers the integration options for Spring Cloud Task with Spring Cloud Stream.
--

[[stream-integration-launching-sink]]
== Launching a Task from a Spring Cloud Stream

You can launch tasks from a stream. To do so, create a sink that listens for a message that contains a `TaskLaunchRequest` as its payload. The `TaskLaunchRequest` contains:

* `uri`: To the task artifact that is to be executed.
* `applicationName`: The name that is associated with the task. If no applicationName is set, the `TaskLaunchRequest` generates a task name comprised of the following: `Task-<UUID>`.  
* `commandLineArguments`: A list containing the command line arguments for the task.  
* `environmentProperties`: A map containing the environment variables to be used by the task.  
* `deploymentProperties`: A map containing the properties that are used by the deployer to deploy the task.

NOTE: If the payload is of a different type, the sink throws an exception.

For example, a stream can be created that has a processor that takes in data from an HTTP source and creates a `GenericMessage` that contains the `TaskLaunchRequest` and sends the message to its output channel. The task sink would then receive the message from its input channel and then launch the task.

To create a taskSink, you need only create a Spring Boot application that includes the
EnableTaskLauncher` annotation, as shown in the following example:

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

The https://github.com/spring-cloud/spring-cloud-task/tree/master/spring-cloud-task-samples[samples module] of the Spring Cloud Task project contains a sample Sink and Processor. To install these samples into your local maven repository, run a maven build from the `spring-cloud-task-samples` directory with the `skipInstall` property set to `false`, as shown in the following example:

```
mvn clean install
```

NOTE: The `maven.remoteRepositories.springRepo.url` property must be set to the location of the remote repository in which the über-jar is located. If not set, there is no remote repository, so it relies upon the local repository only.

[[stream-integration-launching-sink-dataflow]]

### Spring Cloud Data Flow

To create a stream in Spring Cloud Data Flow, you must first register the Task Sink Application we created. In the following example, we are registering the Processor and Sink sample applications by using the Spring Cloud Data Flow shell:

```bash
app register --name taskSink --type sink --uri maven://io.spring.cloud:tasksink:<version> app register --name taskProcessor --type processor --uri maven:io.spring.cloud:taskprocessor:<version>
```
The following example shows how to create a stream from the Spring Cloud Data Flow shell:

[source,bash]
stream create foo --definition "http --server.port=9000|taskProcessor|taskSink" --deploy

[[stream-integration-events]]
== Spring Cloud Task Events

Spring Cloud Task provides the ability to emit events through a Spring Cloud Stream channel when the task is run through a Spring Cloud Stream channel. A task listener is used to publish the `TaskExecution` on a message channel named `task-events`. This feature is autowired into any task that has `spring-cloud-stream`, `spring-cloud-stream-binder`, and a defined task on its classpath.

NOTE: To disable the event emitting listener, set the `spring.cloud.task.events.enabled` property to `false`.

With the appropriate classpath defined, the following task emits the `TaskExecution` as an event on the `task-events` channel (at both the start and the end of the task):

[source, java]

@SpringBootApplication public class TaskEventsApplication {

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

    @Configuration
    public static class TaskConfiguration {

```
@Bean
public CommandLineRunner commandLineRunner() {
    return new CommandLineRunner() {
        @Override
        public void run(String... args) throws Exception {
            System.out.println("The CommandLineRunner was executed");
        }
    };
}

NOTE: A binder implementation is also required to be on the classpath.

NOTE: A sample task event application can be found in the samples module of the Spring Cloud Task Project, https://github.com/spring-cloud/spring-cloud-task/tree/master/spring-cloud-task-samples/task-events[here].

[[stream-integration-disable-task-events]]
=== Disabling Specific Task Events

To disable task events, you can set the ‘spring.cloud.task.events.enabled’ property to 'false'.

[[stream-integration-batch-events]]
== Spring Batch Events

When executing a Spring Batch job through a task, Spring Cloud Task can be configured to emit informational messages based on the Spring Batch listeners available in Spring Batch. Specifically, the following Spring Batch listeners are autoconfigured into each batch job and emit messages on the associated Spring Cloud Stream channels when run through Spring Cloud Task:

* `JobExecutionListener` listens for `job-execution-events`
* `StepExecutionListener` listens for `step-execution-events`
* `ChunkListener` listens for `chunk-events`
* `ItemReadListener` listens for `item-read-events`
* `ItemProcessListener` listens for `item-process-events`
* `ItemWriteListener` listens for `item-write-events`
* `SkipListener` listens for `skip-events`

These listeners are autoconfigured into any `AbstractJob` when the appropriate beans (a `Job` and a `TaskLifecycleListener`) exist in the context. Configuration to listen to these events is handled the same way binding to any other Spring...
Cloud Stream channel is done. Our task (the one running the batch job) serves as a ‘Source’, with the listening applications serving as either a ‘Processor’ or a ‘Sink’.

An example could be to have an application listening to the ‘job-execution-events’ channel for the start and stop of a job. To configure the listening application, you would configure the input to be ‘job-execution-events’ as follows:

`spring.cloud.stream.bindings.input.destination=job-execution-events`

NOTE: A binder implementation is also required to be on the classpath.

NOTE: A sample batch event application can be found in the samples module of the Spring Cloud Task Project, https://github.com/spring-cloud/spring-cloud-task/tree/master/spring-cloud-task-samples/batch-events[here].

=== Sending Batch Events to Different Channels

One of the options that Spring Cloud Task offers for batch events is the ability to alter the channel to which a specific listener can emit its messages. To do so, use the following configuration: `spring.cloud.stream.bindings.<the channel>.destination=<new destination>`.

For example, if `StepExecutionListener` needs to emit its messages to another channel called `my-step-execution-events` instead of the default `step-execution-events`, you can add the following configuration:

`spring.cloud.stream.bindings.step-execution-events.destination=my-step-execution-events`

=== Disabling Batch Events

To disable the listener functionality for all batch events, use the following configuration:

`spring.cloud.task.batch.events.enabled=false`

To disable a specific batch event, use the following configuration:

`spring.cloud.task.batch.events.<batch event listener>.enabled=false`

The following listing shows individual listeners that you can disable:

[source,bash]

```bash
spring.cloud.task.batch.events.job-execution.enabled=false
spring.cloud.task.batch.events.step-execution.enabled=false
spring.cloud.task.batch.events.chunk.enabled=false
spring.cloud.task.batch.events.item-read.enabled=false
```
== Emit Order for Batch Events
By default, batch events have `Ordered.LOWEST_PRECEDENCE`. To change this value (for example, to 5), use the following configuration:

```bash
spring.cloud.task.batch.events.job-execution-order=5 spring.cloud.task.batch.events.step-execution-order=5 spring.cloud.task.batch.events.chunk-order=5 spring.cloud.task.batch.events.item-read-order=5 spring.cloud.task.batch.events.item-process-order=5 spring.cloud.task.batch.events.item-write-order=5 spring.cloud.task.batch.events.skip-order=5
```

[[appendix]]
= Appendices

[[appendix-task-repository-schema]]
== Task Repository Schema

[[partintro]]
--
This appendix provides an ERD for the database schema used in the task repository.
--

image::task_schema.png[]

== Table Information
--

<table>
<thead>
<tr>
<th>.TASK_EXECUTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stores the task execution information.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Required</th>
<th>Type</th>
<th>Field Length</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>TASK_EXECUTION_ID</td>
<td>TRUE</td>
<td>BIGINT</td>
<td>X</td>
<td>Spring Cloud Task Framework at app startup establishes the next available id as obtained from the 'TASK_SEQ'. Or if the record is created outside of task then the value must be populated at record creation time.</td>
</tr>
<tr>
<td>START_TIME</td>
<td>FALSE</td>
<td>DATETIME</td>
<td>X</td>
<td>Spring Cloud Task Framework at app startup establishes the value.</td>
</tr>
<tr>
<td>END_TIME</td>
<td>FALSE</td>
<td>DATETIME</td>
<td>X</td>
<td>Spring Cloud Task Framework at app exit establishes the value.</td>
</tr>
<tr>
<td>TASK_NAME</td>
<td>FALSE</td>
<td>VARCHAR</td>
<td>100</td>
<td>Spring Cloud Task Framework at app startup will set this to &quot;Application&quot; unless user establish the name using the</td>
</tr>
<tr>
<td>EXIT_CODE</td>
<td>FALSE</td>
<td>INTEGER</td>
<td>X</td>
<td>Follows Spring Boot defaults unless overridden by the user as discussed <a href="https://docs.spring.io/spring-cloud-task/docs/current/reference/#features-lifecycle-exit-codes">https://docs.spring.io/spring-cloud-task/docs/current/reference/#features-lifecycle-exit-codes</a></td>
</tr>
<tr>
<td>-------------</td>
<td>-------</td>
<td>---------</td>
<td>---</td>
<td>----------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>EXIT_MESSAGE</td>
<td>FALSE</td>
<td>VARCHAR</td>
<td>2500</td>
<td>User Defined as discussed <a href="https://docs.spring.io/spring-cloud-task/docs/current/reference/#features-task-execution-listener-exit-messages">https://docs.spring.io/spring-cloud-task/docs/current/reference/#features-task-execution-listener-exit-messages</a></td>
</tr>
<tr>
<td>ERROR_MESSAGE</td>
<td>FALSE</td>
<td>VARCHAR</td>
<td>2500</td>
<td>Spring Cloud Task Framework at app exit establishes the value.</td>
</tr>
<tr>
<td>LAST_UPDATED</td>
<td>TRUE</td>
<td>DATETIME</td>
<td>X</td>
<td>Spring Cloud Task Framework at app startup establishes the value. Or if the record is created outside of task then the value must be populated at record creation time.</td>
</tr>
<tr>
<td>EXTERNAL_EXECUTION_ID</td>
<td>FALSE</td>
<td>VARCHAR</td>
<td>250</td>
<td>If the <code>spring.cloud.task.external-execution-id</code> property is set then Spring Cloud Task Framework at app startup will set this to the value specified. More information can be found &lt;&lt;features-external_task_id,here&gt;&gt;</td>
</tr>
<tr>
<td>PARENT_TASK_EXECUTION_ID</td>
<td>FALSE</td>
<td>BIGINT</td>
<td>X</td>
<td>If the <code>spring.cloud.task.parent-execution-id</code> property is set then Spring Cloud Task Framework at app startup will set this to the value specified. More information can be found &lt;&lt;features-parent_task_id,here&gt;&gt;</td>
</tr>
</tbody>
</table>

---

**.TASK_EXECUTION_PARAMS**
Stores the parameters used for a task execution

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Required</th>
<th>Type</th>
<th>Field Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>TASK_EXECUTION_ID</td>
<td>TRUE</td>
<td>BIGINT</td>
<td>X</td>
</tr>
<tr>
<td>TASK_PARAM</td>
<td>FALSE</td>
<td>VARCHAR</td>
<td>2500</td>
</tr>
</tbody>
</table>

---

**.TASK_TASK_BATCG**
Used to link the task execution to the batch execution.

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Required</th>
<th>Type</th>
<th>Field Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>TASK_EXECUTION_ID</td>
<td>TRUE</td>
<td>BIGINT</td>
<td>X</td>
</tr>
<tr>
<td>JOB_EXECUTION_ID</td>
<td>TRUE</td>
<td>BIGINT</td>
<td>X</td>
</tr>
</tbody>
</table>
Used for the ‘single-instance-enabled’ feature discussed.

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Required</th>
<th>Type</th>
<th>Field Length</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOCK_KEY</td>
<td>TRUE</td>
<td>CHAR</td>
<td>36</td>
<td>UUID for the lock</td>
</tr>
<tr>
<td>REGION</td>
<td>TRUE</td>
<td>VARCHAR</td>
<td>100</td>
<td>User can establish a group of locks using this field.</td>
</tr>
<tr>
<td>CLIENT_ID</td>
<td>TRUE</td>
<td>CHAR</td>
<td>36</td>
<td>The task execution id that contains the name of the app to lock.</td>
</tr>
<tr>
<td>CREATED_DATE</td>
<td>TRUE</td>
<td>DATETIME</td>
<td>X</td>
<td>The date that the entry was created</td>
</tr>
</tbody>
</table>

NOTE: The DDL for setting up tables for each database type can be found https://github.com/spring-cloud/spring-cloud-task/tree/master/spring-cloud-task-core/src/main/resources/org/springframework/cloud/task[here].

== Building This Documentation

This project uses Maven to generate this documentation. To generate it for yourself, run the following command: `$ ./mvnw clean package -P full`.

== Running a Task App on Cloud Foundry

The simplest way to launch a Spring Cloud Task application as a task on Cloud Foundry is to use Spring Cloud Data Flow. Via Spring Cloud Data Flow you can register your task application, create a definition for it and then launch it. You then can track the task execution(s) via a RESTful API, the Spring Cloud Data Flow Shell, or the UI. To learn out to get started installing Data Flow follow the instructions in the https://docs.spring.io/spring-cloud-dataflow/docs/current/reference/htmlsingle/#getting-started[Getting Started] section of the reference documentation. For info on how to register and launch tasks, see the https://docs.spring.io/spring-cloud-dataflow/docs/current/reference/htmlsingle/#_the_lifecycle_of_a_task[Lifecycle of a Task] documentation.
Spring Cloud Vault Config provides client-side support for externalized configuration in a distributed system.

With https://www.vaultproject.io[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, Couchbase, MongoDB, Consul, AWS and more.

== New & Noteworthy

This section briefly covers items that are new and noteworthy in the latest releases.

[[new-in-3.0.0]]

=== New in Spring Cloud Vault 3.0
* Migration of `PropertySource` initialization from Spring Cloud's Bootstrap Context to Spring Boot's <<vault.configdata,ConfigData API>>.
* Support for the <<vault.config.backends.couchbase>> backend.
* Configuration of keystore/truststore types through `spring.cloud.vault.ssl.key-store-type=...`/`spring.cloud.vault.ssl.trust-store-type=...` including PEM support.
* Support for `ReactiveDiscoveryClient` by configuring a `ReactiveVaultEndpointProvider`.

== 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

NOTE: 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: https://learn.hashicorp.com/vault

*Install Vault*

[source,bash]

```bash
$ wget releases.hashicorp.com/vault/$(vault_version)/vault_${vault_version}_${platform}.zip
$ unzip vault_${vault_version}_${platform}.zip
```
NOTE: These steps can be achieved by downloading and running https://github.com/spring-cloud/spring-cloud-vault/blob/master/src/test/bash/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.


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`.

[[quickstart.vault.start]]

*Start Vault server*

Next create a config file along the lines of:

[source]

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


[source,bash]

$ 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.

NOTE: 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.

[source,bash]

```bash
$ export VAULT_ADDR="https://localhost:8200" $ export VAULT_SKIP_VERIFY=true # Don't do this for production $ vault init
```

You should see something like:

[source,bash]

```bash
Key 1: 7149c6a2e16b8833f6eb1e76df03e47f6113a3288b3093af5033d44f0e70fe701 Key 2: 901c534c7988c18c20435a85213c683bdcf0e6fc82e82893779f152978c18c02 Key 3: 03ff3948575b1165a20c20ee7c3e6ed04f4cbe0e82dbff5be49c63f98bc03a03 Key 4: 216ae5cc3ddaf93ceb8e1d15bb9fc3176653f5b738f5f3d1ee00cd7dcccbe926e04 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.

[source,bash]

```bash
$ 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.

== 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:

```xml
<parent>  
  <groupId>org.springframework.boot</groupId>  
  <artifactId>spring-boot-starter-parent</artifactId>  
  <version>2.4.0.RELEASE</version>  
  <relativePath /> <!-- lookup parent from repository - - </parent>  

<dependencies>  
  <dependency>  
    <groupId>org.springframework.cloud</groupId>  
    <artifactId>spring-cloud-starter-vault-config</artifactId>  
    <version>3.1.3-SNAPSHOT</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:

```java
@SpringBootApplication @RestController public class Application {
```
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 `application.properties`, for example

```yaml
```

* `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
* `spring.config.import` mounts Vault as `PropertySource` using all enabled secret backends (key-value enabled by default)

Enabling further integrations requires additional dependencies and configuration. Depending on how you have set up Vault you might need additional configuration like `{docs}`#vault.config.ssl[SSL] and `{docs}`#vault.config.authentication[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`).

NOTE: With Spring Cloud Vault 3.0 and Spring Boot 2.4, the bootstrap context initialization (`bootstrap.yml`, `bootstrap.properties`) of property sources was deprecated.
Instead, Spring Cloud Vault favors Spring Boot's Config Data API which allows importing configuration from Vault. With Spring Boot Config Data approach, you need to set the `spring.config.import` property in order to bind to Vault. You can read more about it in the <<vault.configdata.locations, Config Data Locations section>>.
You can enable the bootstrap context either by setting the configuration property `spring.cloud.bootstrap.enabled=true` or by including the dependency `org.springframework.cloud:spring-cloud-starter-bootstrap`.

=== Authentication


Spring Cloud Vault supports multiple {docs}#vault.config.authentication[authentication mechanisms] to authenticate applications with Vault.

For a quickstart, use the root token printed by the <<quickstart.vault.start,Vault initialization>>.

.application.yml
====
[source,yaml]

```
spring.cloud.vault: token: 19aefa97-cccc-bbbb-aaaa-225940e63d76 spring.config.import: vault://
```

FFFF

WARNING: 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.

[[vault.configdata]]
== ConfigData API

Spring Boot provides since version 2.4 a ConfigData API that allows the declaration of configuration sources and importing these as property sources.

Spring Cloud Vault uses as of version 3.0 the ConfigData API to mount Vault's secret backends as property sources.
In previous versions, the Bootstrap context was used. The ConfigData API is much more flexible as it allows specifying which configuration systems to import and in which order.

NOTE: You can enable the deprecated bootstrap context either by setting the configuration property `spring.cloud.bootstrap.enabled=true` or by including the dependency `org.springframework.cloud:spring-cloud-starter-bootstrap`.

[[vault.configdata.locations]]

### ConfigData Locations

You can mount Vault configuration through one or more `PropertySource` that are materialized from Vault.

Spring Cloud Vault supports two config locations:

* `vault://` (default location)
* `vault://<context-path>` (contextual location)

Using the default location mounts property sources for all enabled `<<vault.config.backends>>`. Without further configuration, Spring Cloud Vault mounts the key-value backend at `/secret/${spring.application.name}`.

Each activated profile adds another context path following the form `/secret/${spring.application.name}/${profile}`.

Adding further modules to the classpath, such as `spring-cloud-config-databases`, provides additional secret backend configuration options which get mounted as property sources if enabled.

If you want to control which context paths are mounted from Vault as `PropertySource`, you can either use a contextual location (`vault:///my/context/path`) or configure a `<<vault.config.backends.configurer,`VaultConfigurer`>>`.

Contextual locations are specified and mounted individually. Spring Cloud Vault mounts each location as a unique `PropertySource`.

You can mix the default locations with contextual locations (or other config systems) to control the order of property sources.

This approach is useful in particular if you want to disable the default key-value path computation and mount each key-value backend yourself instead.

.application.yml

```yaml
spring.config.import: vault://first/context/path, vault://other/path, vault://
```

Spring Cloud Vault requires infrastructure classes to interact with Vault. When not using the ConfigData API (meaning that you haven't specified `spring.config.import=vault://` or a contextual Vault path), Spring Cloud Vault defines its beans through `VaultAutoConfiguration` and `VaultReactiveAutoConfiguration`.

Spring Boot bootstraps the application before a Spring Context is available. Therefore `VaultConfigDataLoader` registers beans itself to propagate these later on into the application context.

You can customize the infrastructure used by Spring Cloud Vault by registering custom instances using the `Bootstrapper` API:

```java
InstanceSupplier<RestTemplateBuilder> builderSupplier = ctx → RestTemplateBuilder .builder() .requestFactory(ctx.get(ClientFactoryWrapper.class).getClientHttpRequestFactory()) .defaultHeader("X-Vault-Namespace", "my-namespace");

SpringApplication application = new SpringApplication(MyApplication.class);
application.addBootstrapper(registry ← registry.register(RestTemplateBuilder.class, builderSupplier));
```
See also <<vault.config.backends.configurer>> and the source of `VaultConfigDataLoader` for customization hooks.

[[vault.config.authentication]]
== 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.

[[vault.config.authentication.token]]
=== 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].

NOTE: 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.

```yaml
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: https://www.vaultproject.io/docs/concepts/tokens.html

```yaml
[[vault.config.authentication.vault-agent]]
=== 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.
```

```

```

```yaml
.spring.cloud.vault: authentication: NONE
```
* `authentication` setting this value to `NONE` disables `ClientAuthentication` and `SessionManager`.

See also: https://www.vaultproject.io/docs/agent/index.html[Vault Documentation: Agent]

```yaml
[[vault.config.authentication.appid]]
```

### AppId authentication

Vault supports [AppId](https://www.vaultproject.io/docs/auth/app-id.html) 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.

```yaml
dot application.yml using SHA256 IP-Address UserId's
```

The corresponding command to generate the IP address UserId from a command line is:

```bash
$ echo -n 192.168.99.1 | sha256sum
```

```yaml
spring.cloud.vault: authentication: APPID app-id: user-id: IP_ADDRESS
```
NOTE: 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).

.application.yml using SHA256 Mac-Address UserId's

```yaml
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
```
NOTE: 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.

.application.yml
====
[source,yaml]

spring.cloud.vault: authentication: APPID app-id: user-id: com.example.MyUserIdMechanism

====

.MyUserIdMechanism.java
====
[source,yaml]

public class MyUserIdMechanism implements AppIdUserIdMechanism {

    @Override
    public String createUserId() {
        String userId = ...
        return userId;
    }
}
See also: https://www.vaultproject.io/docs/auth/app-id.html[Vault Documentation: Using the App ID auth backend]

=== AppRole authentication

https://www.vaultproject.io/docs/auth/app-id.html[AppRole] is intended for machine authentication, like the deprecated (since Vault 0.6.1) <<vault.config.authentication.appid>>. 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.

.application.yml with AppRole authentication properties

```yaml
spring.cloud.vault: authentication: APPROLE app-role: role-id: bde2076b-cccb-3cf0-d57e-bca7b1e83a52
```
The following scenarios are supported along the required configuration details:

### Configuration

<table>
<thead>
<tr>
<th><em>Method</em></th>
<th><em>RoleId</em></th>
<th><em>SecretId</em></th>
<th><em>RoleName</em></th>
<th><em>Token</em></th>
</tr>
</thead>
<tbody>
<tr>
<td>Provided RoleId/SecretId</td>
<td>Provided</td>
<td>Provided</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Provided RoleId without SecretId</td>
<td>Provided</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Provided RoleId, Pull SecretId</td>
<td>Provided</td>
<td>Provided</td>
<td>Provided</td>
<td>Provided</td>
</tr>
<tr>
<td>Pull RoleId, provided SecretId</td>
<td>Provided</td>
<td>Provided</td>
<td>Provided</td>
<td>Provided</td>
</tr>
<tr>
<td>Pull Mode</td>
<td></td>
<td></td>
<td>Provided</td>
<td>Provided</td>
</tr>
<tr>
<td>Wrapped</td>
<td></td>
<td></td>
<td></td>
<td>Provided</td>
</tr>
<tr>
<td>Wrapped RoleId, provided SecretId</td>
<td>Provided</td>
<td></td>
<td></td>
<td>Provided</td>
</tr>
<tr>
<td>Provided RoleId, wrapped SecretId</td>
<td></td>
<td>Provided</td>
<td></td>
<td>Provided</td>
</tr>
</tbody>
</table>

### Pull/Push/Wrapped Matrix

<table>
<thead>
<tr>
<th><em>RoleId</em></th>
<th><em>SecretId</em></th>
<th><em>Supported</em></th>
</tr>
</thead>
<tbody>
<tr>
<td>Provided</td>
<td>Provided</td>
<td></td>
</tr>
<tr>
<td>Provided</td>
<td>Pull</td>
<td></td>
</tr>
<tr>
<td>Provided</td>
<td>Wrapped</td>
<td></td>
</tr>
<tr>
<td>Provided</td>
<td>Absent</td>
<td></td>
</tr>
<tr>
<td>Pull</td>
<td>Provided</td>
<td></td>
</tr>
<tr>
<td>Pull</td>
<td>Pull</td>
<td></td>
</tr>
<tr>
<td>Pull</td>
<td>Wrapped</td>
<td></td>
</tr>
<tr>
<td>Pull</td>
<td>Absent</td>
<td></td>
</tr>
<tr>
<td>Wrapped</td>
<td>Provided</td>
<td></td>
</tr>
<tr>
<td>Wrapped</td>
<td>Pull</td>
<td></td>
</tr>
<tr>
<td>Wrapped</td>
<td>Wrapped</td>
<td></td>
</tr>
<tr>
<td>Wrapped</td>
<td>Absent</td>
<td></td>
</tr>
</tbody>
</table>

NOTE: You can use still all combinations of push/pull/wrapped modes by providing a configured `AppRoleAuthentication` bean within the context.
Spring Cloud Vault cannot derive all possible AppRole combinations from the configuration properties.

IMPORTANT: 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`.

.application.yml with all AppRole authentication properties

[source,yaml]
spring.cloud.vault: authentication: APPROLE app-role: role-id: bde2076b-ccc9-b3f0-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: https://www.vaultproject.io/docs/auth/approle.html

[[vault.config.authentication.awsec2]]

```yaml
spring.cloud.vault: authentication: AWS_EC2
```

The https://www.vaultproject.io/docs/auth/aws-ec2.html
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.

```
.application.yml using AWS-EC2 Authentication
```

```yaml
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.

```
.application.yml with configured role

spring.cloud.vault: authentication: AWS_EC2 aws-ec2: role: application-server
```

```
.application.yml with all AWS EC2 authentication properties

```
* `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: https://www.vaultproject.io/docs/auth/aws.html[Vault Documentation: Using the aws auth backend]

```
[[vault.config.authentication.awsiam]]
```

--- AWS-IAM authentication

The https://www.vaultproject.io/docs/auth/aws-ec2.html[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.

```
.application.yml with required AWS-IAM Authentication properties
```

```
[source,yaml]
spring.cloud.vault: authentication: AWS_IAM
```

```
.application.yml with all AWS-IAM Authentication properties
```

```
[source,yaml]
* `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-name` 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: https://www.vaultproject.io/docs/auth/aws.html[Vault Documentation: Using the aws auth backend]

```yaml
[spring.cloud.vault: authentication: AZURE_MSI]
  azure-msi: role: my-dev-role
```

====

**Azure MSI authentication**

The https://www.vaultproject.io/docs/auth/azure.html[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.

```yaml
[spring.cloud.vault: authentication: AZURE_MSI]
  azure-msi: role: my-dev-role
  azure-path: azure
  metadata-service: 169.254.169.254/metadata/instance...
  identity-token-service: 169.254.169.254/
```
* `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
* `metadata-service` sets the URI at which to access the instance metadata service
* `identity-token-service` sets the URI at which to access the identity token service

Azure MSI authentication obtains environmental details about the virtual machine (subscription Id, resource group, VM name) from the instance metadata service. The Vault server has Resource Id defaults to `https://vault.hashicorp.com`. To change this, set `spring.cloud.vault.azure-msi.identity-token-service` accordingly.

See also:

* https://www.vaultproject.io/docs/auth/azure.html
* https://docs.microsoft.com/en-us/azure/virtual-machines/windows/instance-metadata-service

```yaml
[[vault.config.authentication.clientcert]]
```
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*

NOTE: Response Wrapping for token creation requires Vault 0.6.0 or higher.

```
$ 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
```

```yaml
spring.cloud.vault: authentication: CUBBYHOLE token: 397ccb93-ff6c-b17b-9389-380b01ca2645
```
See also:
* https://www.vaultproject.io/docs/concepts/tokens.html
* https://www.vaultproject.io/docs/secrets/cubbyhole/index.html
* https://www.vaultproject.io/docs/concepts/response-wrapping.html

```yaml
[[vault.config.authentication.gcp-gce]]
```

### GCP-GCE authentication


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 https://cloud.google.com/compute/docs/instances/verifying-instance-identity[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.

```yaml
.application.yml with required GCP-GCE Authentication properties
====
[source,yaml]
```

```
spring.cloud.vault: authentication: GCP_GCE gcp-gce: role: my-dev-role
```

```yaml
.application.yml with all GCP-GCE Authentication properties
====
[source,yaml]
```

```
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:

* https://www.vaultproject.io/docs/auth/gcp.html [Vault Documentation: Using the GCP auth backend]

```
[vault.config.authentication.gcpiam]
```

### GCP-IAM authentication

The [gcp](https://www.vaultproject.io/docs/auth/gcp.html) 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](https://cloud.google.com/iam/reference/rest/v1/projects.serviceAccounts/signJwt). 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](https://cloud.google.com/docs/authentication/production), 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`.

```
.springframework.cloud.vault: authentication: GCP_IAM gcp-iam: role: my-dev-role
```

```
[source,yaml]
```

```
spring.cloud.vault: authentication: GCP_IAM gcp-iam: role: my-dev-role
```

```
[source,yaml]
```
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.

NOTE: 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:

* https://www.vaultproject.io/docs/auth/gcp.html [Vault Documentation: Using the GCP auth backend]

[[vault.authentication.gcpiam]]
[[vault.config.authentication.kubernetes]]

--- 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`.

`.application.yml with all Kubernetes authentication properties

---

```
```
* `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:

* https://www.vaultproject.io/docs/auth/kubernetes.html
* https://kubernetes.io/docs/tasks/configure-pod-container/configure-service-account/

[[vault.config.authentication.pcf]]

### Pivotal CloudFoundry authentication

The https://www.vaultproject.io/docs/auth/pcf.html[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.

**.application.yml with required PCF Authentication properties**

```yaml
---
[source,yaml]
```

```yaml
spring.cloud.vault: authentication: PCF pcf: role: my-dev-role
```

**.application.yml with all PCF Authentication properties**

```yaml
---
[source,yaml]
```

```yaml
```

* `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.

NOTE: PCF authentication requires BouncyCastle (bcpkix-jdk15on) to be on the classpath for RSA PSS signing.

See also: https://www.vaultproject.io/docs/auth/pcf.html[Vault Documentation: Using the pcf auth backend]

```
[[vault.config.acl]]
== ACL Requirements

This section explains which paths are accessed by Spring Vault so you can derive your policy declarations from the required capabilities.

|===
|Capability |Associated HTTP verbs
|create    |`POST`/`PUT`
|read      |`GET`
|update    |`POST`/`PUT`
|delete    |`DELETE`
|list      |`LIST` (`GET`)  
|===

See also https://www.vaultproject.io/guides/identity/policies.

=== Authentication

Login: `POST auth/$authMethod/login`

=== KeyValue Mount Discovery

`GET sys/internal/ui/mounts/$mountPath`

=== SecretLeaseContainer

`SecretLeaseContainer` uses different paths depending on the configured lease endpoint.

`LeaseEndpoints.Legacy`
* Revocation: `PUT sys/revoke`
* Renewal: `PUT sys/renew`

`LeaseEndpoints.Leases` (`SysLeases`)

* Revocation: `PUT sys/leases/revoke`
* Renewal: `PUT sys/leases/renew`

=== Session Management

* Token lookup: `GET auth/token/lookup-self`
* Renewal: `POST auth/token/renew-self`
* Revoke: `POST auth/token/revoke-self`

[[vault.config.backends]]
== Secret Backends

[[vault.config.backends.kv]]
[[vault.config.backends.generic]]
[[vault.config.backends.kv.versioned]]

=== 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).

NOTE: 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.

```yaml
```
* `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

NOTE: The key-value secret backend can be operated in versioned (v2) and non-versioned (v1) modes.

See also:

* https://www.vaultproject.io/docs/secrets/kv/kv-v2.html[Vault Documentation: Using the KV Secrets Engine - Version 2 (versioned key-value backend)]

[[vault.config.backends.consul]]

=== Consul

Spring Cloud Vault can obtain credentials for HashiCorp Consul. The Consul integration requires the `spring-cloud-vault-config-consul` dependency.

.pom.xml

```xml
<dependencies>
    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-vault-config-consul</artifactId>
        <version>3.1.3-SNAPSHOT</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: https://www.vaultproject.io/docs/secrets/consul/index.html[Vault Documentation: Setting up Consul with Vault]

```
[[vault.config.backends.rabbitmq]]
=== RabbitMQ

Spring Cloud Vault can obtain credentials for RabbitMQ.

The RabbitMQ integration requires the `spring-cloud-vault-config-rabbitmq` dependency.

```
<dependencies>    <dependency>        <groupId>org.springframework.cloud</groupId>        <artifactId>spring-cloud-vault-config-rabbitmq</artifactId>        <version>3.1.3-SNAPSHOT</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`.

```yaml
```

* `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: https://www.vaultproject.io/docs/secrets/rabbitmq/index.html[Vault Documentation: Setting up RabbitMQ with Vault]

```xml
<dependencies>
  <dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-vault-config-aws</artifactId>
    <version>3.1.3-SNAPSHOT</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=...`.

Supported AWS credential Types:

* iam_user (Defaults)
* assumed_role (STS)
* federation_token (STS)

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`.

For STS security token, you can configure the property name by setting `spring.cloud.vault.aws.session-token-key-property`. The security token is stored under `cloud.aws.credentials.sessionToken` (defaults).

Example: iam_user

Example: assumed_role (STS)

Example: federation_token (STS)

* `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
* `session-token-key-property` sets the property name in which the AWS STS security token is stored.
* `credential-type` sets the AWS credential type to use for this backend. Defaults to `iam_user`
* `ttl` sets the ttl for the STS token when using `assumed_role` or `federation_token`. Defaults to the ttl specified by the vault role. Min/Max values are also limited to what AWS would support for STS.
* `role-arn` sets the IAM role to assume if more than one are configured for the vault role when using `assumed_role`.

See also: https://www.vaultproject.io/docs/secrets/aws/index.html[Vault Documentation: Setting up AWS with Vault]

[[vault.config.backends.database-backends]]
== 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:

* `<<vault.config.backends.database>>`
* `<<vault.config.backends.cassandra>>`
* `<<vault.config.backends.couchbase>>`
* `<<vault.config.backends.elasticsearch>>`
* `<<vault.config.backends.mongodb>>`
* `<<vault.config.backends.mysql>>`
* `<<vault.config.backends.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`.

```
<dependencies>
  <dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-vault-config-databases</artifactId>
    <version>3.1.3-SNAPSHOT</version>
  </dependency>
</dependencies>
```
NOTE: 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.

[[vault.config.backends.database]]

== Database

Spring Cloud Vault can obtain credentials for any database listed at https://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'.

====

[source,yaml]

* `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: https://www.vaultproject.io/docs/secrets/databases/index.html[Vault Documentation: Database Secrets backend]

WARNING: 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.

```yaml
[[vault.config.backends.cassandra]]
=== Apache Cassandra

NOTE: 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`.

```yaml
```
* `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: https://www.vaultproject.io/docs/secrets/cassandra/index.html

[[vault.config.backends.couchbase]]

[== Couchbase Database ==]

Spring Cloud Vault can obtain credentials for Couchbase.
The integration can be enabled by setting
`spring.cloud.vault.couchbase.enabled=true` (default `false`) and providing the role
name with `spring.cloud.vault.couchbase.role=…`.

Username and password are available from `spring.couchbase.username`
and `spring.couchbase.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.couchbase.username-property` and
`spring.cloud.vault.couchbase.password-property`.

[==]

```yaml
```
* `enabled` setting this value to `true` enables the Couchbase backend config usage
* `role` sets the role name of the Couchbase role definition
* `backend` sets the path of the Couchbase mount to use
* `username-property` sets the property name in which the Couchbase username is stored
* `password-property` sets the property name in which the Couchbase password is stored

See also: https://github.com/hashicorp/vault-plugin-database-couchbase

```
[[vault.config.backends.elasticsearch]]
```
* `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: https://www.vaultproject.io/docs/secrets/databases/elasticdb[ Vault Documentation: Setting up Elasticsearch with Vault]

```
[[vault.config.backends.elasticsearch]]

### MongoDB

NOTE: 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`.

```

```
* `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: https://www.vaultproject.io/docs/secrets/mongodb/index.html

[[vault.config.backends.mongodb]]

=== MySQL

NOTE: 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`.

====

[source,yaml]

```yaml
```
* `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: https://www.vaultproject.io/docs/secrets/mysql/index.html

[[vault.config.backends.postgresql]]

```yaml
```
* `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: https://www.vaultproject.io/docs/secrets/postgresql/index.html

```java
public class CustomizationBean implements VaultConfigurer {

@Override
public void addSecretBackends(SecretBackendConfigurer configurer) {

  configurer.add("secret/my-application");
}
```
configurer.registerDefaultKeyValueSecretBackends(false);
configurer.registerDefaultDiscoveredSecretBackends(true);
}
}

[source,java]

SpringApplication application = new SpringApplication(MyApplication.class);
application.addBootstrapper(VaultBootstrapper.fromConfigurer(new CustomizationBean()));

====

[[vault.config.backends.custom]]
== Custom Secret Backend Implementations

Spring Cloud Vault ships with secret backend support for the most common backend integrations.
You can integrate with any kind of backend by providing an implementation that describes how to obtain data from the backend you want to use and how to surface data provided by that backend by providing a `PropertyTransformer`.

Adding a custom implementation for a backend requires implementation of two interfaces:

* `org.springframework.cloud.vault.config.VaultSecretBackendDescriptor`
* `org.springframework.cloud.vault.config.SecretBackendMetadataFactory`

`VaultSecretBackendDescriptor` is typically an object that holds configuration data, such as `VaultDatabaseProperties`. Spring Cloud Vault requires that your type is annotated with `@ConfigurationProperties` to materialize the class from the configuration.

`SecretBackendMetadataFactory` accepts `VaultSecretBackendDescriptor` to create the actual `SecretBackendMetadata` object which holds the context path within your Vault server, any path variables required to resolve parametrized context paths and `PropertyTransformer`.

Both, `VaultSecretBackendDescriptor` and `SecretBackendMetadataFactory` types must be registered in `spring.factories` which is an extension mechanism provided by Spring, similar to Java's ServiceLoader.

== 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 `application.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.

```yaml
spring.cloud.vault.discovery: enabled: true service-id: my-vault-service
```

[[vault.config.fail-fast]]
== 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.

```yaml
spring.cloud.vault: fail-fast: true
```
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: https://www.vaultproject.io/docs/enterprise/namespaces/index.html[Vault
Enterprise: Namespaces]

---

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.

---

```
```

* `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
* `trust-store-type` sets the trust-store type. Supported values are all supported `KeyStore` types including `PEM`.
* `enabled-protocols` sets the list of enabled SSL/TLS protocols (since 3.0.2).
* `enabled-cipher-suites` sets the list of enabled SSL/TLS cipher suites (since 3.0.2).

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.

[[vault-lease-renewal]]
== 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.

NOTE: 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.

[[source,yaml]]
```
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: https://www.vaultproject.io/docs/concepts/lease.html

```yaml
spring.cloud.vault: session.lifecycle: enabled: true refresh-before-expiry: 10s expiry-threshold: 20s
```

---

* `enabled` controls whether session lifecycle management is enabled to renew session tokens. Enabled by default.
* `refresh-before-expiry` controls the point in time when the session token gets renewed. The refresh time is calculated by subtracting `refresh-before-expiry` from the token expiry time. Defaults to `5 seconds`.

* `expiry-threshold` sets the expiry threshold. The threshold represents a minimum TTL duration to consider a session token as valid. Tokens with a shorter TTL are considered expired and are not used anymore. Should be greater than `refresh-before-expiry` to prevent token expiry. Defaults to `7 seconds`.

See also: https://www.vaultproject.io/api-docs/auth/token#renew-a-token-self[Vault Documentation: Token Renewal]

<table>
<thead>
<tr>
<th>Name</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>spring.cloud.vault.app-id.app-id-path</td>
<td><code>app-id</code></td>
<td>Mount path of the AppId authentication backend.</td>
</tr>
<tr>
<td>spring.cloud.vault.app-id.network-interface</td>
<td></td>
<td>Network interface hint for the &quot;MAC_ADDRESS&quot; UserId mechanism.</td>
</tr>
</tbody>
</table>
|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.
|spring.cloud.vault.authentication | |
|spring.cloud.vault.aws-ec2.aws-ec2-path | `aws-ec2` | Mount path of the AWS-EC2 authentication backend.
|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.credential-type | aws credential type
|spring.cloud.vault.aws.enabled | `false` | Enable aws backend usage.
|spring.cloud.vault.aws.role | Role name for credentials.
|spring.cloud.vault.aws.role-arn | Role arn for assumed_role in case we have multiple roles associated with the vault role. @since 3.0.2
|spring.cloud.vault.aws.secret-key-property | `cloud.aws.credentials.secretKey` | Target property for the obtained secret key.
|spring.cloud.vault.aws.session-token-key-property | `cloud.aws.credentials.sessionToken` | Target property for the obtained secret key.
|spring.cloud.vault.aws.ttl | `0` | TTL for sts tokens. Defaults to whatever the vault Role may have for Max. Also limited to what AWS supports to be the max for STS. @since 3.0.2
|spring.cloud.vault.azure-msi.azure-path | `azure` | Mount path of the Azure MSI authentication backend.
|spring.cloud.vault.azure-msi.identity-token-service | Identity token service URI. @since 3.0
|spring.cloud.vault.azure-msi.metadata-service | Instance metadata service URI. @since 3.0
|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.
<table>
<thead>
<tr>
<th>Property Name</th>
<th>Default Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>spring.cloud.vault.cassandra.role</td>
<td></td>
<td>Role name for credentials.</td>
</tr>
<tr>
<td>spring.cloud.vault.cassandra.role</td>
<td><code>false</code></td>
<td>Enable static role usage. @since 2.2</td>
</tr>
<tr>
<td>spring.cloud.vault.cassandra.username-property</td>
<td><code>spring.data.cassandra.username</code></td>
<td>Target property for the obtained username.</td>
</tr>
<tr>
<td>spring.cloud.vault.config.lifecycle.enabled</td>
<td><code>true</code></td>
<td>Enable lifecycle management.</td>
</tr>
<tr>
<td>spring.cloud.vault.config.lifecycle.expiry-threshold</td>
<td></td>
<td>The expiry threshold. {link Lease} is renewed the given {link Duration} before it expires. @since 2.2</td>
</tr>
<tr>
<td>spring.cloud.vault.config.lifecycle.lease-endpoints</td>
<td></td>
<td>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</td>
</tr>
<tr>
<td>spring.cloud.vault.config.lifecycle.min-renewal</td>
<td></td>
<td>The time period that is at least required before renewing a lease. @since 2.2</td>
</tr>
<tr>
<td>spring.cloud.vault.connection-timeout</td>
<td><code>5000</code></td>
<td>Connection timeout.</td>
</tr>
<tr>
<td>spring.cloud.vault.consul.backend</td>
<td><code>consul</code></td>
<td>Consul backend path.</td>
</tr>
<tr>
<td>spring.cloud.vault.consul.enabled</td>
<td><code>false</code></td>
<td>Enable consul backend usage.</td>
</tr>
<tr>
<td>spring.cloud.vault.consul.role</td>
<td></td>
<td>Role name for credentials.</td>
</tr>
<tr>
<td>spring.cloud.vault.consul.token-property</td>
<td><code>spring.cloud.consul.token</code></td>
<td>Target property for the obtained token.</td>
</tr>
<tr>
<td>spring.cloud.vault.couchbase.backend</td>
<td><code>database</code></td>
<td>Couchbase backend path.</td>
</tr>
<tr>
<td>spring.cloud.vault.couchbase.enabled</td>
<td><code>false</code></td>
<td>Enable couchbase backend usage.</td>
</tr>
<tr>
<td>spring.cloud.vault.couchbase.password-property</td>
<td><code>spring.couchbase.password</code></td>
<td>Target property for the obtained password.</td>
</tr>
<tr>
<td>spring.cloud.vault.couchbase.role</td>
<td></td>
<td>Role name for credentials.</td>
</tr>
<tr>
<td>spring.cloud.vault.couchbase.username-property</td>
<td><code>spring.couchbase.username</code></td>
<td>Target property for the obtained username.</td>
</tr>
<tr>
<td>spring.cloud.vault.database.backend</td>
<td><code>database</code></td>
<td>Database backend path.</td>
</tr>
<tr>
<td>spring.cloud.vault.database.enabled</td>
<td><code>false</code></td>
<td>Enable database backend usage.</td>
</tr>
<tr>
<td>spring.cloud.vault.database.password-property</td>
<td><code>spring.datasource.password</code></td>
<td>Target property for the obtained password.</td>
</tr>
<tr>
<td>spring.cloud.vault.database.role</td>
<td></td>
<td>Role name for credentials.</td>
</tr>
<tr>
<td>spring.cloud.vault.database.username-property</td>
<td><code>spring.datasource.username</code></td>
<td>Target property for the obtained username.</td>
</tr>
<tr>
<td>spring.cloud.vault.discovery.enabled</td>
<td><code>false</code></td>
<td>Flag to indicate that Vault server discovery is enabled (vault server URL will be looked up via discovery).</td>
</tr>
<tr>
<td>spring.cloud.vault.elasticsearch.backend</td>
<td><code>database</code></td>
<td>Database backend path.</td>
</tr>
<tr>
<td>spring.cloud.vault.elasticsearch.enabled</td>
<td><code>false</code></td>
<td>Enable elasticsearch backend usage.</td>
</tr>
<tr>
<td>spring.cloud.vault.elasticsearch.password-property</td>
<td><code>spring.elasticsearch.rest.password</code></td>
<td>Target property for the obtained password.</td>
</tr>
<tr>
<td>spring.cloud.vault.elasticsearch.role</td>
<td></td>
<td>Role name for credentials.</td>
</tr>
<tr>
<td>spring.cloud.vault.elasticsearch.static-role</td>
<td><code>false</code></td>
<td>Enable static role usage.</td>
</tr>
<tr>
<td>Property</td>
<td>Description</td>
<td></td>
</tr>
<tr>
<td>-----------</td>
<td>-------------</td>
<td></td>
</tr>
<tr>
<td><code>spring.cloud.vault.elasticsearch.username-property</code></td>
<td>Target property for the obtained username.</td>
<td></td>
</tr>
<tr>
<td><code>spring.cloud.vault.enabled</code></td>
<td><code>true</code></td>
<td>Enable Vault config server.</td>
</tr>
<tr>
<td><code>spring.cloud.vault.fail-fast</code></td>
<td><code>false</code></td>
<td>Fail fast if data cannot be obtained from Vault.</td>
</tr>
<tr>
<td><code>spring.cloud.vault.gcp-gce.gcp-path</code></td>
<td><code>gcp</code></td>
<td>Mount path of the Kubernetes authentication backend.</td>
</tr>
<tr>
<td><code>spring.cloud.vault.gcp-gce.role</code></td>
<td></td>
<td>Name of the role against which the login is being attempted.</td>
</tr>
<tr>
<td><code>spring.cloud.vault.gcp-gce.service-account</code></td>
<td></td>
<td>Optional service account id. Using the default id if left unconfigured.</td>
</tr>
<tr>
<td><code>spring.cloud.vault.gcp-iam.credentials.encoded-key</code></td>
<td></td>
<td>The base64 encoded contents of an OAuth2 account private key in JSON format.</td>
</tr>
<tr>
<td><code>spring.cloud.vault.gcp-iam.credentials.location</code></td>
<td></td>
<td>Location of the OAuth2 credentials private key. Since this is a Resource, the private key can be in a multitude of locations, such as a local file system, classpath, URL, etc.</td>
</tr>
<tr>
<td><code>spring.cloud.vault.gcp-iam.gcp-path</code></td>
<td><code>gcp</code></td>
<td>Mount path of the Kubernetes authentication backend.</td>
</tr>
<tr>
<td><code>spring.cloud.vault.gcp-iam.jwt-validity</code></td>
<td><code>15m</code></td>
<td>Validity of the JWT token.</td>
</tr>
<tr>
<td><code>spring.cloud.vault.gcp-iam.project-id</code></td>
<td></td>
<td>Overrides the GCP project Id.</td>
</tr>
<tr>
<td><code>spring.cloud.vault.gcp-iam.role</code></td>
<td></td>
<td>Name of the role against which the login is being attempted.</td>
</tr>
<tr>
<td><code>spring.cloud.vault.gcp-iam.service-account-id</code></td>
<td></td>
<td>Overrides the GCP service account Id.</td>
</tr>
<tr>
<td><code>spring.cloud.vault.host</code></td>
<td><code>localhost</code></td>
<td>Vault server host.</td>
</tr>
<tr>
<td><code>spring.cloud.vault.kubernetes.kubernetes-path</code></td>
<td><code>kubernetes</code></td>
<td>Mount path of the Kubernetes authentication backend.</td>
</tr>
<tr>
<td><code>spring.cloud.vault.kubernetes.role</code></td>
<td></td>
<td>Name of the role against which the login is being attempted.</td>
</tr>
<tr>
<td><code>spring.cloud.vault.kubernetes.service-account-token-file</code></td>
<td><code>/var/run/secrets/kubernetes.io/serviceaccount/token</code></td>
<td>Path to the service account token file.</td>
</tr>
<tr>
<td><code>spring.cloud.vault.kv.application-name</code></td>
<td><code>application</code></td>
<td>Application name to be used for the context.</td>
</tr>
<tr>
<td><code>spring.cloud.vault.kv.backend</code></td>
<td><code>secret</code></td>
<td>Name of the default backend.</td>
</tr>
<tr>
<td><code>spring.cloud.vault.kv.default-context</code></td>
<td><code>application</code></td>
<td>Name of the default context.</td>
</tr>
<tr>
<td><code>spring.cloud.vault.kv.enabled</code></td>
<td><code>true</code></td>
<td>Enable the key-value backend.</td>
</tr>
<tr>
<td><code>spring.cloud.vault.kv.profile-separator</code></td>
<td><code>/</code></td>
<td>Profile-separator to combine application name and profile.</td>
</tr>
<tr>
<td><code>spring.cloud.vault.kv.profiles</code></td>
<td></td>
<td>List of active profiles. @since 3.0</td>
</tr>
<tr>
<td><code>spring.cloud.vault.mongodb.backend</code></td>
<td><code>mongodb</code></td>
<td>MongoDB backend path.</td>
</tr>
<tr>
<td><code>spring.cloud.vault.mongodb.enabled</code></td>
<td><code>false</code></td>
<td>Enable mongodb backend usage.</td>
</tr>
<tr>
<td><code>spring.cloud.vault.mongodb.password-property</code></td>
<td><code>spring.data.mongodb.password</code></td>
<td>Target property for the obtained password.</td>
</tr>
<tr>
<td><code>spring.cloud.vault.mongodb.role</code></td>
<td></td>
<td>Role name for credentials.</td>
</tr>
<tr>
<td><code>spring.cloud.vault.mongodb.static-role</code></td>
<td><code>false</code></td>
<td>Enable static role usage. @since 2.2</td>
</tr>
<tr>
<td><code>spring.cloud.vault.mongodb.username-property</code></td>
<td><code>spring.data.mongodb.username</code></td>
<td></td>
</tr>
<tr>
<td>Property</td>
<td>Value</td>
<td>Description</td>
</tr>
<tr>
<td>----------------------------------------------</td>
<td>----------------------</td>
<td>--------------------------------------------------</td>
</tr>
<tr>
<td>spring.cloud.vault.mysql.backend</td>
<td><code>mysql</code></td>
<td>mysql backend path.</td>
</tr>
<tr>
<td>spring.cloud.vault.mysql.enabled</td>
<td><code>false</code></td>
<td>Enable mysql backend usage.</td>
</tr>
<tr>
<td>spring.cloud.vault.mysql.password-property</td>
<td><code>{spring.datasource.password}</code></td>
<td>Target property for the obtained username.</td>
</tr>
<tr>
<td>spring.cloud.vault.mysql.role</td>
<td><code>false</code></td>
<td>Role name for credentials.</td>
</tr>
<tr>
<td>spring.cloud.vault.mysql.username-property</td>
<td><code>{spring.datasource.username}</code></td>
<td>Target property for the obtained username.</td>
</tr>
<tr>
<td>spring.cloud.vault.namespace</td>
<td><code>false</code></td>
<td>Vault namespace (requires Vault Enterprise).</td>
</tr>
<tr>
<td>spring.cloud.vault.pcf.instance-certificate</td>
<td><code>false</code></td>
<td>Path to the instance certificate (PEM). Defaults to <code>{CF_INSTANCE_CERT}</code> env variable.</td>
</tr>
<tr>
<td>spring.cloud.vault.pcf.instance-key</td>
<td><code>false</code></td>
<td>Path to the instance key (PEM). Defaults to <code>{CF_INSTANCE_KEY}</code> env variable.</td>
</tr>
<tr>
<td>spring.cloud.vault.pcf.pcf-path</td>
<td><code>pcf</code></td>
<td>Mount path of the Kubernetes authentication backend.</td>
</tr>
<tr>
<td>spring.cloud.vault.pcf.role</td>
<td><code>false</code></td>
<td>Name of the role against which the login is being attempted.</td>
</tr>
<tr>
<td>spring.cloud.vault.postgresql.backend</td>
<td><code>postgresql</code></td>
<td>postgresql backend path.</td>
</tr>
<tr>
<td>spring.cloud.vault.postgresql.enabled</td>
<td><code>false</code></td>
<td>Enable postgresql backend usage.</td>
</tr>
<tr>
<td>spring.cloud.vault.postgresql.password-property</td>
<td><code>{spring.datasource.password}</code></td>
<td>Target property for the obtained username.</td>
</tr>
<tr>
<td>spring.cloud.vault.postgresql.role</td>
<td><code>false</code></td>
<td>Role name for credentials.</td>
</tr>
<tr>
<td>spring.cloud.vault.postgresql.username-property</td>
<td><code>{spring.datasource.username}</code></td>
<td>Target property for the obtained username.</td>
</tr>
<tr>
<td>spring.cloud.vault.rabbitmq.backend</td>
<td><code>rabbitmq</code></td>
<td>rabbitmq backend path.</td>
</tr>
<tr>
<td>spring.cloud.vault.rabbitmq.enabled</td>
<td><code>false</code></td>
<td>Enable rabbitmq backend usage.</td>
</tr>
<tr>
<td>spring.cloud.vault.rabbitmq.password-property</td>
<td><code>{spring.rabbitmq.password}</code></td>
<td>Target property for the obtained password.</td>
</tr>
<tr>
<td>spring.cloud.vault.rabbitmq.role</td>
<td><code>false</code></td>
<td>Role name for credentials.</td>
</tr>
<tr>
<td>spring.cloud.vault.rabbitmq.username-property</td>
<td><code>{spring.rabbitmq.username}</code></td>
<td>Target property for the obtained username.</td>
</tr>
<tr>
<td>spring.cloud.vault.read-timeout</td>
<td><code>15000</code></td>
<td>Read timeout.</td>
</tr>
<tr>
<td>spring.cloud.vault.scheme</td>
<td><code>https</code></td>
<td>Protocol scheme. Can be either &quot;http&quot; or &quot;https&quot;.</td>
</tr>
<tr>
<td>spring.cloud.vault.session.lifecycle.enabled</td>
<td><code>true</code></td>
<td>Enable session lifecycle management.</td>
</tr>
<tr>
<td>spring.cloud.vault.session.lifecycle.expiry-threshold</td>
<td><code>7s</code></td>
<td>The expiry threshold for a <code>{LoginToken}</code>. The threshold represents a minimum TTL duration to consider a login token as valid. Tokens with a shorter TTL are considered expired and are not used anymore. Should be greater than <code>{refreshBeforeExpiry}</code> to prevent token expiry.</td>
</tr>
<tr>
<td>spring.cloud.vault.session.lifecycle.refresh-before-expiry</td>
<td><code>5s</code></td>
<td>The time period that is at least required before renewing the <code>{LoginToken}</code>.</td>
</tr>
<tr>
<td>spring.cloud.vault.ssl.cert-auth-path</td>
<td><code>cert</code></td>
<td>Mount path of the TLS cert authentication backend.</td>
</tr>
<tr>
<td>spring.cloud.vault.ssl.enabled-cipher-suites</td>
<td><code>false</code></td>
<td>List of enabled SSL/TLS cipher suites. @since 3.0.2</td>
</tr>
<tr>
<td>spring.cloud.vault.ssl.enabled-protocols</td>
<td><code>false</code></td>
<td>List of enabled SSL/TLS protocol. @since 3.0.2</td>
</tr>
<tr>
<td>spring.cloud.vault.ssl.key-store</td>
<td><code>false</code></td>
<td>Trust store that holds certificates and private keys.</td>
</tr>
</tbody>
</table>
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.
== Quick Start

This quick start walks through using Spring Cloud Zookeeper for Service Discovery and Distributed Configuration.

First, run Zookeeper on your machine. Then you can access it and use it as a Service Registry and Configuration source with Spring Cloud Zookeeper.

=== Discovery Client Usage

To use these features in an application, you can build it as a Spring Boot application that depends on `spring-cloud-zookeeper-core` and `spring-cloud-zookeeper-discovery`. The most convenient way to add the dependency is with a Spring Boot starter: `org.springframework.cloud:spring-cloud-starter-zookeeper-discovery`. We recommend using dependency management and 'spring-boot-starter-parent'. The following example shows a typical Maven configuration:

[source,xml,indent=0]
.pom.xml

```
<project>  
  <parent>  
    <groupId>org.springframework.boot</groupId>  
    <artifactId>spring-boot-starter-parent</artifactId>  
    <version>2.4.5-SNAPSHOT</version>  
    <relativePath/> <!-- lookup parent from repository -->  
  </parent>
```

The following example shows a typical Gradle setup:

[source,groovy,indent=0]
.build.gradle

plugins { id 'org.springframework.boot' version "$2.4.5-SNAPSHOT" id 'io.spring.dependency-management' version "${spring-dependency-management-version}" id 'java' }

repositories { mavenCentral() }

WARNING: Depending on the version you are using, you might need to adjust Apache Zookeeper version used in your project. You can read more about it in the Install Zookeeper section.

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

```java
@SpringBootApplication @RestController public class Application {

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

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

When this HTTP server runs, it connects to Zookeeper, which runs on the default local port (2181). To modify the startup behavior, you can change the location of Zookeeper by using `application.properties`, as shown in the following example:

```properties
spring: cloud: zookeeper: connect-string: localhost:2181
```

You can now use `DiscoveryClient`, `@LoadBalanced RestTemplate`, or `@LoadBalanced WebClient.Builder` to retrieve services and instances data from Zookeeper, as shown in the following example:

```java
@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; }
```
To use these features in an application, you can build it as a Spring Boot application that depends on `spring-cloud-zookeeper-core` and `spring-cloud-zookeeper-config`. The most convenient way to add the dependency is with a Spring Boot starter: `org.springframework.cloud:spring-cloud-starter-zookeeper-config`. We recommend using dependency management and `spring-boot-starter-parent`. The following example shows a typical Maven configuration:

```xml
<project>
  <parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.4.5-SNAPSHOT</version>
    <relativePath>!-- lookup parent from repository --!</relativePath>
  </parent>
  
  <dependencies>
    <dependency>
      <groupId>org.springframework.cloud</groupId>
      <artifactId>spring-cloud-starter-zookeeper-config</artifactId>
    </dependency>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-test</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>
  
  <build>
    <plugins>
      <plugin>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-maven-plugin</artifactId>
      </plugin>
    </plugins>
  </build>
</project>
```
The following example shows a typical Gradle setup:

```groovy
plugins { id 'org.springframework.boot' version '2.4.5-SNAPSHOT' id 'io.spring.dependency-management' version '${spring-dependency-management-version}' id 'java' }

repositories { mavenCentral() }

dependencies { implementation 'org.springframework.cloud:spring-cloud-starter-zookeeper-config' testImplementation 'org.springframework.boot:spring-boot-starter-test' } dependencyManagement {
    imports { mavenBom "org.springframework.cloud:spring-cloud-dependencies:${springCloudVersion}" }
}

WARNING: Depending on the version you are using, you might need to adjust Apache Zookeeper version used in your project. You can read more about it in the <<spring-cloud-zookeeper-install,Install Zookeeper section>>.

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

```java
@SpringBootApplication @RestController public class Application {

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

    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }
}
```
The application retrieves configuration data from Zookeeper.

WARNING: If you use Spring Cloud Zookeeper Config, you need to set the 'spring.config.import' property in order to bind to Zookeeper. You can read more about it in the <<config-data-import,Spring Boot Config Data Import section>>.

[[spring-cloud-zookeeper-install]]
== 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
[source,xml,indent=0]

<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>  
<groupId>org.apache.zookeeper</groupId>  
<artifactId>zookeeper</artifactId>  
</dependency>  
<version>3.4.12</version>  
<exclusions>  
<exclusion>  
<groupId>org.slf4j</groupId>  
<artifactId>slf4j-log4j12</artifactId>  
</exclusion>  
</exclusions>  
</dependency>

.gradle
[source,groovy,indent=0]
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. https://curator.apache.org/curator (A Java library for Zookeeper) provides Service Discovery through a https://curator.apache.org/curator-x-discovery/Service Discovery Extension. Spring Cloud Zookeeper uses this extension for service registration and discovery.

--- Activating

Including a dependency on `org.springframework.cloud:spring-cloud-starter-zookeeper-discovery` enables autoconfiguration that sets up Spring Cloud Zookeeper Discovery.

NOTE: For web functionality, you still need to include `org.springframework.boot:spring-boot-starter-web`.

CAUTION: When working with version 3.4 of Zookeeper you need to change the way you include the dependency as described <<spring-cloud-zookeeper-install,here>>.

--- 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:

```java
@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);
    }
```

```java```
NOTE: 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:

[source,yml,indent=0]
.application.yml

```yaml
spring: cloud: zookeeper: connect-string: localhost:2181
```
CAUTION: If you use <<spring-cloud-zookeeper-config,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`.

---

Using the DiscoveryClient

Spring Cloud has support for
(a REST client builder),
https://github.com/spring-cloud/spring-cloud-netflix/blob/master/docs/src/main/asciidoc/spring-cloud.RestTemplate] and
https://cloud.spring.io/spring-cloud-commons/reference/html/#loadbalanced-webclient[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:

```java
@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; }
```
== Using Spring Cloud Zookeeper with Spring Cloud Components

Feign, Spring Cloud Gateway and Spring Cloud LoadBalancer all work with Spring Cloud Zookeeper.

=== 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.

TIP: If you were previously using the StickyRule in Zookeeper, its replacement in the current stack is the `SameInstancePreferenceServiceInstanceListSupplier` in SC LoadBalancer. You can read on how to set it up in the [Spring Cloud Commons documentation](https://docs.spring.io/spring-cloud-commons/docs/current/reference/html/#spring-cloud-loadbalancer).

== 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:

```java
@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); }
```
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 localhost:8081/service-registry status=OUT_OF_SERVICE
```

NOTE: The preceding example uses the `http` command from https://httpie.org.

---

=== 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

https://github.com/spring-cloud/spring-cloud-netflix/blob/master/docs/src/main/asciidoc/spring-cloud-netflix.adoc#spring-cloud-feign

(Feign)

(a REST client builder),


(Spring 'RestTemplate') and
You can also use the Zookeeper Dependency Watchers functionality to control and monitor the state of your 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`).

Consider the following example of dependency representation:

```yaml
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`.

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:

```java
@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 <<spring-cloud-zookeeper-dependencies-setting-up-aliases, 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:
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 <<spring-cloud-zookeeper-dependency-watcher-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`
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'.

[[spring-cloud-zookeeper-dependency-watcher-presence-checker]]

=== 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.

. If the dependency is marked as 'required' and is not in Zookeeper, when your application boots, it throws an exception and shuts down.
. 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.

[[spring-cloud-zookeeper-config]]

== Distributed Configuration with Zookeeper

Zookeeper provides a https://zookeeper.apache.org/doc/current/zookeeperOver.html#sc_dataModelNameSpace[hierarchical namespace] that lets clients store arbitrary data, such as configuration data. Spring Cloud Zookeeper Config is an alternative to the https://github.com/spring-cloud/spring-cloud-config[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.

--- Activating

Including a dependency on `org.springframework.cloud:spring-cloud-starter-zookeeper-config` enables autoconfiguration that sets up Spring Cloud Zookeeper Config.

CAUTION: When working with version 3.4 of Zookeeper you need to change the way you include the dependency as described <<spring-cloud-zookeeper-install,here>>.

[[config-data-import]]

--- Spring Boot Config Data Import

Spring Boot 2.4 introduced a new way to import configuration data via the `spring.config.import` property. This is now the default way to get configuration from Zookeeper.

To optionally connect to Zookeeper for configuration set the following in `application.properties`:

```
.sspring.config.import=optional:zookeeper:
```
This will connect to Zookeeper at the default location of "localhost:2181". Removing the `optional:` prefix will cause Zookeeper Config to fail if it is unable to connect to Zookeeper. To change the connection properties of Zookeeper Config either set 'spring.cloud.zookeeper.connect-string' or add the connect string to the 'spring.config.import' statement such as, 'spring.config.import=optional:zookeeper:myhost:2818'. The location in the import property has precedence over the 'connect-string' property.

Zookeeper Config will try to load values from four automatic contexts based on `spring.cloud.zookeeper.config.name` (which defaults to the value of the `spring.application.name` property) and `spring.cloud.zookeeper.config.default-context` (which defaults to 'application'). If you want to specify the contexts rather than using the computed ones, you can add that information to the 'spring.config.import' statement.

```
.application.properties
```

```properties
spring.config.import=optional:zookeeper:myhost:2181/contextone;/context/two
```

This will optionally load configuration only from '/contextone' and '/context/two'.

NOTE: A 'bootstrap' file (properties or yaml) is *not* needed for the Spring Boot Config Data method of import via 'spring.config.import'.

```yaml
=== Customizing
Zookeeper Config may be customized by setting the following properties:
```

```yaml
```

* `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.

CAUTION: If you have set `spring.cloud.bootstrap.enabled=true` or `spring.config.use-legacy-processing=true`, or included `spring-cloud-starter-bootstrap`, then the above values will need to be placed in `bootstrap.yml` instead of `application.yml`.

### 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:

```java
@BootstrapConfiguration public class CustomCuratorFrameworkConfig {

    @Bean
    public CuratorFramework curatorFramework() {
        CuratorFramework curator = new CuratorFramework();
        curator.addAuthInfo("digest", "user:password".getBytes());
        return curator;
    }
}
```


Alternatively, you can add your credentials from a class that depends on the existing `CuratorFramework` bean, as shown in the following example:

```java
@BootstrapConfiguration public class DefaultCuratorFrameworkConfig {
```

```java
    public CuratorFramework curatorFramework() {
        CuratorFramework curator = new CuratorFramework();
        curator.addAuthInfo("digest", "user:password".getBytes());
        return curator;
    }

    @Bean
    public DefaultCuratorFrameworkConfig() {
        return new 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
```

---
<table>
<thead>
<tr>
<th>Name</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>eureka.client.eureka-connection-idle-timeout-seconds</td>
<td><code>30</code></td>
<td>Indicates how much time (in seconds) that the HTTP connections to eureka server can stay idle before it can be closed. In the AWS environment, it is recommended that the values is 30 seconds or less, since the firewall cleans up the connection information after a few mins leaving the connection hanging in limbo.</td>
</tr>
<tr>
<td>eureka.client.eureka-server-connect-timeout-seconds</td>
<td><code>5</code></td>
<td>Indicates how long to wait (in seconds) before a connection to eureka server needs to timeout. Note that the connections in the client are pooled by org.apache.http.client.HttpClient and this setting affects the actual connection creation and also the wait time to get the connection from the pool.</td>
</tr>
<tr>
<td>eureka.client.eureka-server-d-n-s-name</td>
<td>``</td>
<td>Gets the DNS name to be queried to get the list of eureka servers. This information is not required if the contract returns the service urls by implementing serviceUrls. The DNS mechanism is used when useDnsForFetchingServiceUrls is set to true and the eureka client expects the DNS to configured a certain way so that it can fetch changing eureka servers dynamically. The changes are effective at runtime.</td>
</tr>
<tr>
<td>eureka.client.eureka-server-port</td>
<td>``</td>
<td>Gets the port to be used to construct the service url to contact eureka server when the list of eureka servers come from the DNS. This information is not required if the contract returns the service urls eurekaServerServiceUrls(String). The DNS mechanism is used when useDnsForFetchingServiceUrls is set to true and the eureka client expects the DNS to configured a certain way so that it can fetch changing eureka servers dynamically. The</td>
</tr>
<tr>
<td>Property</td>
<td>Default</td>
<td>Description</td>
</tr>
<tr>
<td>----------</td>
<td>---------</td>
<td>-------------</td>
</tr>
<tr>
<td>eureka.client.eureka-server-read-timeout-seconds</td>
<td><code>8</code></td>
<td>Indicates how long to wait (in seconds) before a read from eureka server needs to timeout.</td>
</tr>
<tr>
<td>eureka.client.eureka-server-total-connections</td>
<td><code>200</code></td>
<td>Gets the total number of connections that is allowed from eureka client to all eureka servers.</td>
</tr>
<tr>
<td>eureka.client.eureka-server-total-connections-per-host</td>
<td><code>50</code></td>
<td>Gets the total number of connections that is allowed from eureka client to a eureka server host.</td>
</tr>
<tr>
<td>eureka.client.eureka-server-url-context</td>
<td><code>''</code></td>
<td>Gets the URL context to be used to construct the service url to contact eureka server when the list of eureka servers come from the DNS. This information is not required if the contract returns the service urls from eurekaServerServiceUrls. The DNS mechanism is used when useOnsForFetchingServiceUrls is set to true and the eureka client expects the DNS to configured a certain way so that it can fetch changing eureka servers dynamically. The changes are effective at runtime.</td>
</tr>
<tr>
<td>eureka.client.eureka-service-url-poll-interval-seconds</td>
<td><code>0</code></td>
<td>Indicates how often(in seconds) to poll for changes to eureka server information. Eureka servers could be added or removed and this setting controls how soon the eureka clients should know about it.</td>
</tr>
<tr>
<td>eureka.client.prefer-same-zone-eureka</td>
<td><code>true</code></td>
<td>Indicates whether or not this instance should try to use the eureka server in the same zone for latency and/or other reason. Ideally eureka clients are configured to talk to servers in the same zone. The changes are effective at runtime at the next registry fetch cycle as specified by registryFetchIntervalSeconds.</td>
</tr>
<tr>
<td>eureka.client.register-with-eureka</td>
<td><code>true</code></td>
<td>Indicates whether or not this instance should register its information with eureka server for discovery by others. In some cases, you do not want your instances to be discovered whereas you just want do discover other instances.</td>
</tr>
<tr>
<td>eureka.server.peer-eureka-nodes-update-interval-ms</td>
<td><code>0</code></td>
<td></td>
</tr>
<tr>
<td>eureka.server.peer-eureka-status-refresh-time-interval-ms</td>
<td><code>0</code></td>
<td></td>
</tr>
<tr>
<td>feign.autoconfiguration.jackson.enabled</td>
<td><code>false</code></td>
<td>If true, PageJacksonModule and SortJacksonModule bean will be provided for Jackson page decoding.</td>
</tr>
<tr>
<td>feign.circuitbreaker.enabled</td>
<td><code>false</code></td>
<td>If true, an OpenFeign client will be wrapped with a Spring Cloud CircuitBreaker circuit breaker.</td>
</tr>
<tr>
<td>feign.circuitbreaker.group.enabled</td>
<td><code>false</code></td>
<td>If true, an OpenFeign client will be wrapped with a Spring Cloud CircuitBreaker circuit breaker with with group.</td>
</tr>
<tr>
<td>feign.client.config</td>
<td><code>''</code></td>
<td></td>
</tr>
<tr>
<td>feign.client.decode-slash</td>
<td><code>true</code></td>
<td>Feign clients do not encode slash <code>\x60;/\x60;</code> characters by default. To change this behavior, set the <code>\x60;decodeSlash\x60;</code> to <code>\x60;false\x60;</code>.</td>
</tr>
<tr>
<td>feign.client.default-config</td>
<td><code>default</code></td>
<td></td>
</tr>
<tr>
<td>feign.client.default-to-properties</td>
<td><code>true</code></td>
<td></td>
</tr>
<tr>
<td>feign.client.refresh-enabled</td>
<td><code>false</code></td>
<td>Enables options value refresh capability for Feign.</td>
</tr>
<tr>
<td>feign.compression.request.enabled</td>
<td><code>false</code></td>
<td>Enables the request sent by Feign to be compressed.</td>
</tr>
<tr>
<td>feign.compression.request.mime-types</td>
<td><code>['text/xml, application/xml, application/json']</code></td>
<td>The list of supported mime types.</td>
</tr>
<tr>
<td>feign.compression.request.min-request-size</td>
<td><code>2048</code></td>
<td>The minimum threshold content size.</td>
</tr>
<tr>
<td>feign.compression.response.enabled</td>
<td><code>false</code></td>
<td>Enables the response from Feign to be compressed.</td>
</tr>
<tr>
<td>feign.compression.response.useGzipDecoder</td>
<td>'false'</td>
<td>Enables the default gzip decoder to be used.</td>
</tr>
<tr>
<td>feign.encoder.charset-from-content-type</td>
<td>'false'</td>
<td>Indicates whether the charset should be derived from the {code Content-Type} header.</td>
</tr>
<tr>
<td>feign.httpclient.connection-timeout</td>
<td>'2000'</td>
<td></td>
</tr>
<tr>
<td>feign.httpclient.connection-timer-repeat</td>
<td>'3000'</td>
<td></td>
</tr>
<tr>
<td>feign.httpclient.disable-ssl-validation</td>
<td>'false'</td>
<td></td>
</tr>
<tr>
<td>feign.httpclient.enabled</td>
<td>'true'</td>
<td>Enables the use of the Apache HTTP Client by Feign.</td>
</tr>
<tr>
<td>feign.httpclient.follow-redirects</td>
<td>'true'</td>
<td></td>
</tr>
<tr>
<td>feign.httpclient.hc5.enabled</td>
<td>'false'</td>
<td>Enables the use of the Apache HTTP Client 5 by Feign.</td>
</tr>
<tr>
<td>feign.httpclient.hc5.pool-concurrency-policy</td>
<td>''</td>
<td>Pool concurrency policies.</td>
</tr>
<tr>
<td>feign.httpclient.hc5.pool-reuse-policy</td>
<td>''</td>
<td>Pool connection re-use policies.</td>
</tr>
<tr>
<td>feign.httpclient.hc5.socket-timeout</td>
<td>'5'</td>
<td>Default value for socket timeout.</td>
</tr>
<tr>
<td>feign.httpclient.hc5.socket-timeout-unit</td>
<td>''</td>
<td>Default value for socket timeout unit.</td>
</tr>
<tr>
<td>feign.httpclient.max-connections</td>
<td>'200'</td>
<td></td>
</tr>
<tr>
<td>feign.httpclient.max-connections-per-route</td>
<td>'50'</td>
<td></td>
</tr>
<tr>
<td>feign.httpclient.time-to-live</td>
<td>'900'</td>
<td></td>
</tr>
<tr>
<td>feign.httpclient.time-to-live-unit</td>
<td>''</td>
<td></td>
</tr>
<tr>
<td>feign.metrics.enabled</td>
<td>'true'</td>
<td>Enables metrics capability for Feign.</td>
</tr>
<tr>
<td>feign.okhttp.enabled</td>
<td>'false'</td>
<td>Enables the use of the OK HTTP Client by Feign.</td>
</tr>
<tr>
<td>ribbon.eureka.enabled</td>
<td>'true'</td>
<td>Enables the use of Eureka with Ribbon.</td>
</tr>
<tr>
<td>spring.cloud.bus.ack.destination-service</td>
<td>''</td>
<td>Service that wants to listen to acks. By default null (meaning all services).</td>
</tr>
<tr>
<td>spring.cloud.bus.ack.enabled</td>
<td>'true'</td>
<td>Flag to switch off acks (default on).</td>
</tr>
<tr>
<td>spring.cloud.bus.content-type</td>
<td>''</td>
<td>The bus mime-type.</td>
</tr>
<tr>
<td>spring.cloud.bus.destination</td>
<td>''</td>
<td>Name of Spring Cloud Stream destination for messages.</td>
</tr>
<tr>
<td>spring.cloud.bus.enabled</td>
<td>'true'</td>
<td>Flag to indicate that the bus is enabled.</td>
</tr>
<tr>
<td>spring.cloud.bus.env.enabled</td>
<td>'true'</td>
<td>Flag to switch off environment change events (default on).</td>
</tr>
<tr>
<td>spring.cloud.bus.id</td>
<td>'application'</td>
<td>The identifier for this application instance.</td>
</tr>
<tr>
<td>spring.cloud.bus.refresh.enabled</td>
<td>'true'</td>
<td>Flag to switch off refresh events (default on).</td>
</tr>
<tr>
<td>spring.cloud.bus.trace.enabled</td>
<td>'false'</td>
<td>Flag to switch on tracing of acks (default off).</td>
</tr>
<tr>
<td>spring.cloud.cloudfoundry.discovery.default-server-port</td>
<td>'80'</td>
<td>Port to use when no port is defined by service discovery.</td>
</tr>
<tr>
<td>spring.cloud.cloudfoundry.discovery.enabled</td>
<td>'true'</td>
<td>Flag to indicate that discovery is enabled.</td>
</tr>
<tr>
<td>spring.cloud.cloudfoundry.discovery.heartbeat-frequency</td>
<td>'5000'</td>
<td>Frequency in milliseconds of poll for heart beat. The client will poll on this frequency and broadcast a list of service ids.</td>
</tr>
<tr>
<td>spring.cloud.cloudfoundry.discovery.internal-domain</td>
<td>'apps.internal'</td>
<td>Default internal domain when configured to use Native DNS service discovery.</td>
</tr>
<tr>
<td>spring.cloud.cloudfoundry.discovery.order</td>
<td>'0'</td>
<td>Order of the discovery client used by {code CompositeDiscoveryClient}; for sorting available clients.</td>
</tr>
</tbody>
</table>
| spring.cloud.cloudfoundry.discovery.use-container-ip | 'false' | Whether to resolve hostname when BOSH DNS is used. In order to use this feature,
spring.cloud.cloudfoundry.discovery.use-dns 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` | |
|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). |
|spring.cloud.compatibility-verifier.compatible-boot-versions | `` | Default accepted versions for the Spring Boot dependency. You can set `{@code x}` for the patch version if you don’t want to specify a concrete value. Example: `{@code 3.4.x}` |
|spring.cloud.compatibility-verifier.enabled | `false` | Enables creation of Spring Cloud compatibility verification. |
|spring.cloud.config.allow-override | `true` | Flag to indicate that `{@link #isOverrideSystemProperties() systemPropertiesOverride}` 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 `quot;masterquot;` for a git based server). |
|spring.cloud.config.media-type | `` | The Accept header media type to send to config server. |
|spring.cloud.config.name | `` | Name of application used to fetch remote properties. |
|spring.cloud.config.override-none | `false` | Flag to indicate that when `{@link #setAllowOverride(boolean) allowOverride}` 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.tls | `''` | TLS properties. |
|spring.cloud.config.token | `''` | Security Token passed thru to underlying environment repository. |
|spring.cloud.config.username | `''` | The username to use (HTTP Basic) when contacting the remote server. |
|spring.cloud.consul.config.acl-token | `''` | |
|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` | |
|spring.cloud.consul.config.enabled | `true` | |
|spring.cloud.consul.config.fail-fast | `true` | Throw exceptions during config lookup if true, otherwise, log warnings. |
|spring.cloud.consul.config.format | `''` | |
|spring.cloud.consul.config.name | `''` | Alternative to `spring.application.name` to use in looking up values in consul KV. |
|spring.cloud.consul.config.prefix | `''` | |
|spring.cloud.consul.config.prefixes | `''` | |
|spring.cloud.consul.config.profile-separator | `,'` | |
|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.acl-token | `''` | |
|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 to query for in server list. This allows looking up services in another datacenters.
spring.cloud.consul.discovery.default-query-tag | "" | Tag to query for in service list if one is not listed in serverListQueryTags. Multiple tags can be specified with a comma separated value.
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.deregister | "true" | Disable automatic deregistration 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" |
spring.cloud.consul.discovery.heartbeat.interval-ratio | "" | spring.cloud.consul.discovery.heartbeat.reregister-service-on-failure | "false" |
spring.cloud.consul.discovery.heartbeat.ttl | "30s" | 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" | 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 CompositeDiscoveryClient; for sorting available clients. |
| spring.cloud.consul.discovery.port | `""` | Port to register the service under (defaults to listening port). |
| spring.cloud.consul.discovery.prefer-agent-address | `false` | Source of how we will determine the address to use. |
| spring.cloud.consul.discovery.prefer-ip-address | `false` | Use ip address rather than hostname during registration. |
| spring.cloud.consul.discovery.query-passing | `false` | Add the `passing` parameter to `/v1/health/service/serviceName`. This pushes health check passing to the server. |
| spring.cloud.consul.discovery.register | `true` | Register as a service in consul. |
| spring.cloud.consul.discovery.register-health-check | `true` | Register health check in consul. Useful during development of a service. |
| spring.cloud.consul.discovery.scheme | `http` | Whether to register an http or https service. |
| spring.cloud.consul.discovery.server-list-query-tags | `""` | Map of serviceId's tag to query for in server list. This allows filtering services by one more tags. Multiple tags can be specified with a comma separated value. |
| spring.cloud.consul.discovery.service-name | `""` | Service name. |
| spring.cloud.consul.discovery.tags | `""` | Tags to use when registering service. |
| spring.cloud.consul.enabled | `true` | Is spring consul enabled. |
| spring.cloud.consul.host | `localhost` | Consul agent hostname. Defaults to localhost. |
| spring.cloud.consul.path | `""` | Custom path if consul is under non-root. |
| spring.cloud.consul.port | `8500` | Consul agent port. Defaults to localhost:8500. |
| spring.cloud.consul.retry.enabled | `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.multiplier | `1.1` | Multiplier for next interval. |
| 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.decrypt-environment-post-processor.enabled | `true` | Enable the DecryptEnvironmentPostProcessor. |
| spring.cloud.discovery.client.composite-indicator.enabled | `true` | Enables
discovery client composite health indicator.

```yaml
| spring.cloud.discovery.client.health-indicator.enabled | 'true' |
| spring.cloud.discovery.client.health-indicator.include-description | 'false' |
| spring.cloud.discovery.client.health-indicator.use-services-query | 'true' |
Whether or not the indicator should use `{@link DiscoveryClient#getServices}` to check its health. When set to `{@code false}` the indicator instead uses the lighter `{@link DiscoveryClient#probe()}`. This can be helpful in large deployments where the number of services returned makes the operation unnecessarily heavy.

```yaml
| spring.cloud.discovery.client.simple.instances | '' |
| spring.cloud.discovery.client.simple.order | '' |
| spring.cloud.discovery.enabled | 'true' |
| spring.cloud.features.enabled | 'true' |
| spring.cloud.gateway.default-filters | '' |
| spring.cloud.gateway.enabled | 'true' |
| spring.cloud.gateway.discovery.locator.enabled | 'false' |
| spring.cloud.gateway.discovery.locator.filters | '' |
| spring.cloud.gateway.discovery.locator.include-expression | 'true' |
| spring.cloud.gateway.discovery.locator.lower-case-service-id | 'false' |
| spring.cloud.gateway.discovery.locator.prioritize-own-service | 'true' |
| spring.cloud.gateway.discovery.locator.route-id-prefix | '' |
| spring.cloud.gateway.discovery.locator.url-expression | '@%x27;lb://@%x27;+serviceId' |
| spring.cloud.gateway.enabled | 'true' |
| spring.cloud.gateway.fail-on-route-definition-error | 'true' |
| spring.cloud.gateway.filter.add-request-header.enabled | 'true' |
| spring.cloud.gateway.filter.add-request-parameter.enabled | 'true' |
| spring.cloud.gateway.filter.add-response-header.enabled | 'true' |
| spring.cloud.gateway.filter.circuit-breaker.enabled | 'true' |
| spring.cloud.gateway.filter.dedupe-response-header.enabled | 'true' |
| spring.cloud.gateway.filter.fallback-headers.enabled | 'true' |
| spring.cloud.gateway.filter.hystrix.enabled | 'true' |
| spring.cloud.gateway.filter.map-request-header.enabled | 'true' |
| spring.cloud.gateway.filter.modify-request-body.enabled | 'true' |
```

List of filter definitions that are applied to every route.

Flag that enables DiscoveryClient gateway integration.

SpEL expression that will evaluate whether to include a service in gateway integration or not, defaults to: true.

SpEL expression that create the uri for each route, defaults to: `@%x27;lb:////@%x27;+serviceId`.

Enables gateway functionality.

Option to fail on route definition errors, defaults to true. Otherwise, a warning is logged.

Enables the add-request-header filter.

Enables the add-request-parameter filter.

Enables the add-response-header filter.

Enables the circuit-breaker filter.

Enables the dedupe-response-header filter.

Enables the fallback-headers filter.

Enables the hystrix filter.

Enables the map-request-header filter.

Enables the modify-request-body filter.
<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>spring.cloud.gateway.filter.modify-response-body.enabled</td>
<td><code>true</code></td>
</tr>
<tr>
<td>spring.cloud.gateway.filter.prefix-path.enabled</td>
<td><code>true</code></td>
</tr>
<tr>
<td>spring.cloud.gateway.filter.preserve-host-header.enabled</td>
<td><code>true</code></td>
</tr>
<tr>
<td>spring.cloud.gateway.filter.redirect-to.enabled</td>
<td><code>true</code></td>
</tr>
<tr>
<td>spring.cloud.gateway.filter.remove-hop-by-hop.headers</td>
<td><code>''</code></td>
</tr>
<tr>
<td>spring.cloud.gateway.filter.remove-hop-by-hop.order</td>
<td><code>''</code></td>
</tr>
<tr>
<td>spring.cloud.gateway.filter.remove-request-header.enabled</td>
<td><code>true</code></td>
</tr>
<tr>
<td>spring.cloud.gateway.filter.remove-request-parameter.enabled</td>
<td><code>true</code></td>
</tr>
<tr>
<td>spring.cloud.gateway.filter.remove-response-header.enabled</td>
<td><code>true</code></td>
</tr>
<tr>
<td>spring.cloud.gateway.filter.request-header-size.enabled</td>
<td><code>true</code></td>
</tr>
<tr>
<td>spring.cloud.gateway.filter.request-header-to-request-uri.enabled</td>
<td><code>true</code></td>
</tr>
<tr>
<td>spring.cloud.gateway.filter.request-rate-limiter.deny-empty-key</td>
<td><code>true</code></td>
</tr>
<tr>
<td>spring.cloud.gateway.filter.request-rate-limiter.empty-key-status-code</td>
<td><code>''</code></td>
</tr>
<tr>
<td>spring.cloud.gateway.filter.request-rate-limiter.enabled</td>
<td><code>true</code></td>
</tr>
<tr>
<td>spring.cloud.gateway.filter.request-size.enabled</td>
<td><code>true</code></td>
</tr>
<tr>
<td>spring.cloud.gateway.filter.retry.enabled</td>
<td><code>true</code></td>
</tr>
<tr>
<td>spring.cloud.gateway.filter.rewrite-location-response-header.enabled</td>
<td><code>true</code></td>
</tr>
<tr>
<td>spring.cloud.gateway.filter.rewrite-location.enabled</td>
<td><code>true</code></td>
</tr>
<tr>
<td>spring.cloud.gateway.filter.rewrite-path.enabled</td>
<td><code>true</code></td>
</tr>
<tr>
<td>spring.cloud.gateway.filter.rewrite-response-header.enabled</td>
<td><code>true</code></td>
</tr>
<tr>
<td>spring.cloud.gateway.filter.save-session.enabled</td>
<td><code>true</code></td>
</tr>
<tr>
<td>spring.cloud.gateway.filter.secure-headers.content-type-options</td>
<td><code>nosniff</code></td>
</tr>
<tr>
<td>spring.cloud.gateway.filter.secure-headers.disable</td>
<td><code>''</code></td>
</tr>
<tr>
<td>spring.cloud.gateway.filter.secure-headers.download-options</td>
<td><code>noopen</code></td>
</tr>
<tr>
<td>spring.cloud.gateway.filter.secure-headers.enabled</td>
<td><code>true</code></td>
</tr>
<tr>
<td>spring.cloud.gateway.filter.secure-headers.frame-options</td>
<td><code>DENY</code></td>
</tr>
<tr>
<td>spring.cloud.gateway.filter.secure-headers.permitted-cross-domain-policies</td>
<td><code>none</code></td>
</tr>
<tr>
<td>spring.cloud.gateway.filter.secure-headers.referrer-policy</td>
<td><code>no-referrer</code></td>
</tr>
<tr>
<td>spring.cloud.gateway.filter.secure-headers.strict-transport-security</td>
<td><code>max-age=631138519</code></td>
</tr>
<tr>
<td>spring.cloud.gateway.filter.secure-headers.xss-protection-header</td>
<td><code>1; mode=block</code></td>
</tr>
<tr>
<td>spring.cloud.gateway.filter.set-path.enabled</td>
<td><code>true</code></td>
</tr>
<tr>
<td>spring.cloud.gateway.filter.set-request-header.enabled</td>
<td><code>true</code></td>
</tr>
<tr>
<td>spring.cloud.gateway.filter.set-request-host-header.enabled</td>
<td><code>true</code></td>
</tr>
<tr>
<td>spring.cloud.gateway.filter.set-response-header.enabled</td>
<td><code>true</code></td>
</tr>
<tr>
<td>spring.cloud.gateway.filter.set-status.enabled</td>
<td><code>true</code></td>
</tr>
<tr>
<td>spring.cloud.gateway.filter.strip-prefix.enabled</td>
<td><code>true</code></td>
</tr>
<tr>
<td>spring.cloud.gateway.forwarded.enabled</td>
<td><code>true</code></td>
</tr>
<tr>
<td>spring.cloud.gateway.global-filter.adapt-cached-body.enabled</td>
<td><code>true</code></td>
</tr>
<tr>
<td>spring.cloud.gateway.global-filter.forward-path.enabled</td>
<td><code>true</code></td>
</tr>
<tr>
<td>spring.cloud.gateway.global-filter.forward-routing.enabled</td>
<td><code>true</code></td>
</tr>
<tr>
<td>spring.cloud.gateway.global-filter.load-balancer-client.enabled</td>
<td><code>true</code></td>
</tr>
<tr>
<td>spring.cloud.gateway.global-filter.netty-routing.enabled</td>
<td><code>true</code></td>
</tr>
<tr>
<td>spring.cloud.gateway.global-filter.netty-write-response.enabled</td>
<td><code>true</code></td>
</tr>
<tr>
<td>spring.cloud.gateway.global-filter.reactive-load-balancer-client.enabled</td>
<td><code>true</code></td>
</tr>
<tr>
<td>spring.cloud.gateway.global-filter.remove-cached-body.enabled</td>
<td><code>true</code></td>
</tr>
<tr>
<td>spring.cloud.gateway.global-filter.route-to-request-url.enabled</td>
<td><code>true</code></td>
</tr>
<tr>
<td>spring.cloud.gateway.global-filter.websocket-routing.enabled</td>
<td><code>true</code></td>
</tr>
<tr>
<td>spring.cloud.gateway.globalcors.add-to-simple-url-handler-mapping</td>
<td><code>false</code></td>
</tr>
<tr>
<td>spring.cloud.gateway.globalcors.cors-configurations</td>
<td>``</td>
</tr>
<tr>
<td>spring.cloud.gateway.httpclient.compression</td>
<td><code>false</code></td>
</tr>
<tr>
<td>spring.cloud.gateway.httpclient.connect-timeout</td>
<td>``</td>
</tr>
<tr>
<td>spring.cloud.gateway.httpclient.max-header-size</td>
<td>``</td>
</tr>
<tr>
<td>spring.cloud.gateway.httpclient.max-initial-line-length</td>
<td>``</td>
</tr>
<tr>
<td>spring.cloud.gateway.httpclient.pool.acquire-timeout</td>
<td>``</td>
</tr>
</tbody>
</table>
| spring.cloud.gateway.httpclient.pool.eviction-interval | `0` | Perform regular
<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>spring.cloud.gateway.httpclient.pool.max-connections</code></td>
<td><code>Duration.ZERO</code> only for type FIXED, the maximum number of connections before starting pending acquisition on existing ones.</td>
</tr>
<tr>
<td><code>spring.cloud.gateway.httpclient.pool.max-idle-time</code></td>
<td>Time in millis after which the channel will be closed. If NULL, there is no max idle time.</td>
</tr>
<tr>
<td><code>spring.cloud.gateway.httpclient.pool.max-life-time</code></td>
<td>Duration after which the channel will be closed. If NULL, there is no max life time.</td>
</tr>
<tr>
<td><code>spring.cloud.gateway.httpclient.pool.name</code></td>
<td><code>proxy</code></td>
</tr>
<tr>
<td><code>spring.cloud.gateway.httpclient.pool.type</code></td>
<td><code>ELASTIC</code></td>
</tr>
<tr>
<td><code>spring.cloud.gateway.httpclient.proxy.host</code></td>
<td><code>null</code></td>
</tr>
<tr>
<td><code>spring.cloud.gateway.httpclient.proxy.non-proxy-hosts-pattern</code></td>
<td><code>null</code></td>
</tr>
<tr>
<td><code>spring.cloud.gateway.httpclient.proxy.password</code></td>
<td><code>null</code></td>
</tr>
<tr>
<td><code>spring.cloud.gateway.httpclient.proxy.port</code></td>
<td><code>null</code></td>
</tr>
<tr>
<td><code>spring.cloud.gateway.httpclient.proxy.type</code></td>
<td><code>null</code></td>
</tr>
<tr>
<td><code>spring.cloud.gateway.httpclient.proxy.username</code></td>
<td><code>null</code></td>
</tr>
<tr>
<td><code>spring.cloud.gateway.httpclient.response-timeout</code></td>
<td><code>null</code></td>
</tr>
<tr>
<td><code>spring.cloud.gateway.httpclient.ssl.close-notify-flush-timeout</code></td>
<td><code>3000ms</code></td>
</tr>
<tr>
<td><code>spring.cloud.gateway.httpclient.ssl.close-notify-read-timeout</code></td>
<td><code>0</code></td>
</tr>
<tr>
<td><code>spring.cloud.gateway.httpclient.ssl.default-configuration-type</code></td>
<td><code>null</code></td>
</tr>
<tr>
<td><code>spring.cloud.gateway.httpclient.ssl.handshake-timeout</code></td>
<td><code>10000ms</code></td>
</tr>
<tr>
<td><code>spring.cloud.gateway.httpclient.ssl.key-password</code></td>
<td><code>null</code></td>
</tr>
<tr>
<td><code>spring.cloud.gateway.httpclient.ssl.key-store</code></td>
<td><code>null</code></td>
</tr>
<tr>
<td><code>spring.cloud.gateway.httpclient.ssl.key-store-password</code></td>
<td><code>null</code></td>
</tr>
<tr>
<td><code>spring.cloud.gateway.httpclient.ssl.key-store-provider</code></td>
<td><code>null</code></td>
</tr>
<tr>
<td><code>spring.cloud.gateway.httpclient.ssl.key-store-type</code></td>
<td><code>JKS</code></td>
</tr>
<tr>
<td><code>spring.cloud.gateway.httpclient.ssl.trusted-x509-certificates</code></td>
<td><code>null</code></td>
</tr>
<tr>
<td><code>spring.cloud.gateway.httpclient.ssl.use-insecure-trust-manager</code></td>
<td><code>false</code></td>
</tr>
<tr>
<td><code>spring.cloud.gateway.httpclient.websocket.max-frame-payload-length</code></td>
<td><code>null</code></td>
</tr>
<tr>
<td><code>spring.cloud.gateway.httpclient.websocket.proxy-ping</code></td>
<td><code>true</code></td>
</tr>
<tr>
<td>Configuration Key</td>
<td>Default/Value</td>
</tr>
<tr>
<td>-----------------------------------------</td>
<td>-----------------</td>
</tr>
<tr>
<td>spring.cloud.gateway.httpclient.wiretap</td>
<td><code>false</code></td>
</tr>
<tr>
<td>spring.cloud.gateway.httpsserver.wiretap</td>
<td><code>false</code></td>
</tr>
<tr>
<td>spring.cloud.gateway.loadbalancer.use404</td>
<td><code>false</code></td>
</tr>
<tr>
<td>spring.cloud.gateway.metrics.enabled</td>
<td><code>false</code></td>
</tr>
<tr>
<td>spring.cloud.gateway.metrics.prefix</td>
<td><code>spring.cloud.gateway</code></td>
</tr>
<tr>
<td>spring.cloud.gateway.metrics.tags</td>
<td>``</td>
</tr>
<tr>
<td>spring.cloud.gateway.predicate.after.enabled</td>
<td><code>true</code></td>
</tr>
<tr>
<td>spring.cloud.gateway.predicate.before.enabled</td>
<td><code>true</code></td>
</tr>
<tr>
<td>spring.cloud.gateway.predicate.between.enabled</td>
<td><code>true</code></td>
</tr>
<tr>
<td>spring.cloud.gateway.predicate.cloud-foundry-route-service.enabled</td>
<td><code>true</code></td>
</tr>
<tr>
<td>spring.cloud.gateway.predicate.cookie.enabled</td>
<td><code>true</code></td>
</tr>
<tr>
<td>spring.cloud.gateway.predicate.header.enabled</td>
<td><code>true</code></td>
</tr>
<tr>
<td>spring.cloud.gateway.predicate.host.enabled</td>
<td><code>true</code></td>
</tr>
<tr>
<td>spring.cloud.gateway.predicate.method.enabled</td>
<td><code>true</code></td>
</tr>
<tr>
<td>spring.cloud.gateway.predicate.path.enabled</td>
<td><code>true</code></td>
</tr>
<tr>
<td>spring.cloud.gateway.predicate.query.enabled</td>
<td><code>true</code></td>
</tr>
<tr>
<td>spring.cloud.gateway.predicate.read-body.enabled</td>
<td><code>true</code></td>
</tr>
<tr>
<td>spring.cloud.gateway.predicate.remote-addr.enabled</td>
<td><code>true</code></td>
</tr>
<tr>
<td>spring.cloud.gateway.predicate.weight.enabled</td>
<td><code>true</code></td>
</tr>
<tr>
<td>spring.cloud.gateway.redis-rate-limiter.burst-capacity-header</td>
<td><code>X-RateLimit-Burst-Capacity</code></td>
</tr>
<tr>
<td>spring.cloud.gateway.redis-rate-limiter.config</td>
<td>``</td>
</tr>
<tr>
<td>spring.cloud.gateway.redis-rate-limiter.include-headers</td>
<td><code>true</code></td>
</tr>
<tr>
<td>spring.cloud.gateway.redis-rate-limiter.remaining-header</td>
<td><code>X-RateLimit-Remaining</code></td>
</tr>
<tr>
<td>spring.cloud.gateway.redis-rate-limiter.replenish-rate-header</td>
<td><code>X-RateLimit-Replenish-Rate</code></td>
</tr>
<tr>
<td>spring.cloud.gateway.redis-rate-limiter.requested-tokens-header</td>
<td><code>X-RateLimit-Requested-Tokens</code></td>
</tr>
<tr>
<td>spring.cloud.gateway.routes</td>
<td>``</td>
</tr>
<tr>
<td>spring.cloud.gateway.set-status.original-status-header-name</td>
<td>``</td>
</tr>
<tr>
<td>spring.cloud.gateway.streaming-media-types</td>
<td>``</td>
</tr>
</tbody>
</table>
| 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 HttpClient factory beans. 
| spring.cloud.httpclientfactories.ok.enabled | `true` | Enables creation of OK HttpClient factory beans. 
| spring.cloud.hypermedia.refresh.fixed-delay | `5000` | 
| spring.cloud.hypermedia.refresh.initial-delay | `10000` | 
| 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. 
| 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 | `''` | 
| spring.cloud.kubernetes.client.apiVersion | `v1` | Kubernetes API Version 
| spring.cloud.kubernetes.client.ca-cert-data | `''` | 
| spring.cloud.kubernetes.client.ca-cert-file | `''` | 
| spring.cloud.kubernetes.client.caCertData | `''` | Kubernetes API CACertData 
| spring.cloud.kubernetes.client.caCertFile | `''` | Kubernetes API CACertFile 
| spring.cloud.kubernetes.client.client-cert-data | `''` | 
| spring.cloud.kubernetes.client.client-cert-file | `''` |
| spring.cloud.kubernetes.client.client-key-algo | `RSA` | Kubernetes API ClientKeyAlgo
| spring.cloud.kubernetes.client.client-key-data | `` | Kubernetes API ClientKeyData
| spring.cloud.kubernetes.client.client-key-file | `` | Kubernetes API ClientKeyFile
| spring.cloud.kubernetes.client.client-key-passphrase | `changeit` | Kubernetes API ClientKeyPassphrase
| spring.cloud.kubernetes.client.connection-timeout | `` | Connection timeout
| spring.cloud.kubernetes.client.connectionTimeout | `10s` | Connection timeout
| spring.cloud.kubernetes.client.http-proxy | `` | Kubernetes API HTTP Proxy
| spring.cloud.kubernetes.client.https-proxy | `` | Kubernetes API HTTPS Proxy
| spring.cloud.kubernetes.client.logging-interval | `` | Logging interval
| spring.cloud.kubernetes.client.loggingInterval | `20s` | Logging interval
| spring.cloud.kubernetes.client.master-url | `https://kubernetes.default.svc` | Kubernetes API Master Node URL
| spring.cloud.kubernetes.client.namespace | `true` | Kubernetes Namespace
| spring.cloud.kubernetes.client.no-proxy | `` | Kubernetes API No Proxy
| spring.cloud.kubernetes.client.oauth-token | `` | Kubernetes API OAuth Token
| spring.cloud.kubernetes.client.password | `` | Kubernetes API Password
| spring.cloud.kubernetes.client.proxy-password | `` | Kubernetes API Proxy Password
| spring.cloud.kubernetes.client.proxy-username | `` | Kubernetes API Proxy Username
| spring.cloud.kubernetes.client.request-timeout | `` | Request timeout
| spring.cloud.kubernetes.client.requestTimeout | `10s` | Request timeout
| spring.cloud.kubernetes.client.rolling-timeout | `` | Rolling timeout
| spring.cloud.kubernetes.client.rollingTimeout | `900s` | Rolling timeout
| spring.cloud.kubernetes.client.service-account-namespace-path | `'/var/run/secrets/kubernetes.io/serviceaccount/namespace'` | Kubernetes API Service Account Namespace Path
| spring.cloud.kubernetes.client.trust-certs | `false` | Kubernetes API Trust Certificates
| spring.cloud.kubernetes.client.username | `` | Kubernetes API Username
| spring.cloud.kubernetes.client.watch-reconnect-interval | `` | Kubernetes API Watch Reconnect Interval
| spring.cloud.kubernetes.client.watch-reconnect-limit | `-1` | Kubernetes API Watch Reconnect Limit
| spring.cloud.kubernetes.client.watchReconnectInterval | `1s` | Kubernetes API Watch Reconnect Interval limit
| spring.cloud.kubernetes.client.watchReconnectLimit | `-1` | Kubernetes API Watch Reconnect Limit
| spring.cloud.kubernetes.config.enable-api | `true` | Enable the ConfigMap property source locator.
| spring.cloud.kubernetes.config.enabled | `true` | Enable the ConfigMap property source locator.
| spring.cloud.kubernetes.config.name | `` | Kubernetes API Config Source Name
| spring.cloud.kubernetes.config.namespace | `` | Kubernetes API Config Source Namespace
| spring.cloud.kubernetes.config.paths | `` | Kubernetes API Config Source Paths
| spring.cloud.kubernetes.config.sources | `` | Kubernetes API Config Source Sources
| spring.cloud.kubernetes.discovery.all-namespaces | `false` | If discovering all namespaces.
<table>
<thead>
<tr>
<th>Property</th>
<th>Default Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>spring.cloud.kubernetes.discovery.cache-loading-timeout-seconds</code></td>
<td><code>60</code></td>
<td>Timeout for initializing discovery cache, will abort the application if exceeded.</td>
</tr>
<tr>
<td><code>spring.cloud.kubernetes.discovery.enabled</code></td>
<td><code>true</code></td>
<td>If Kubernetes Discovery is enabled.</td>
</tr>
<tr>
<td><code>spring.cloud.kubernetes.discovery.filter</code></td>
<td><code>null</code></td>
<td>SpEL expression to filter services AFTER they have been retrieved from the Kubernetes API server.</td>
</tr>
<tr>
<td><code>spring.cloud.kubernetes.discovery.include-not-ready-addresses</code></td>
<td><code>false</code></td>
<td>If endpoint addresses not marked <code>ready</code> by the k8s api server should be discovered.</td>
</tr>
<tr>
<td><code>spring.cloud.kubernetes.discovery.known-secure-ports</code></td>
<td><code>null</code></td>
<td>Set the port numbers that are considered secure and use HTTPS.</td>
</tr>
<tr>
<td><code>spring.cloud.kubernetes.discovery.metadata.add-annotations</code></td>
<td><code>true</code></td>
<td>When set, the Kubernetes annotations of the services will be included as metadata of the returned ServiceInstance.</td>
</tr>
<tr>
<td><code>spring.cloud.kubernetes.discovery.metadata.add-labels</code></td>
<td><code>true</code></td>
<td>When set, the Kubernetes labels of the services will be included as metadata of the returned ServiceInstance.</td>
</tr>
<tr>
<td><code>spring.cloud.kubernetes.discovery.metadata.add-ports</code></td>
<td><code>true</code></td>
<td>When set, any named Kubernetes service ports will be included as metadata of the returned ServiceInstance.</td>
</tr>
<tr>
<td><code>spring.cloud.kubernetes.discovery.metadata.annotations-prefix</code></td>
<td><code>null</code></td>
<td>When addAnnotations is set, then this will be used as a prefix to the key names in the metadata map.</td>
</tr>
<tr>
<td><code>spring.cloud.kubernetes.discovery.metadata.labels-prefix</code></td>
<td><code>null</code></td>
<td>When addLabels is set, then this will be used as a prefix to the key names in the metadata map.</td>
</tr>
<tr>
<td><code>spring.cloud.kubernetes.discovery.metadata.ports-prefix</code></td>
<td><code>port.</code></td>
<td>When addPorts is set, then this will be used as a prefix to the key names in the metadata map.</td>
</tr>
<tr>
<td><code>spring.cloud.kubernetes.discovery.order</code></td>
<td><code>null</code></td>
<td></td>
</tr>
<tr>
<td><code>spring.cloud.kubernetes.discovery.primary-port-name</code></td>
<td><code>null</code></td>
<td>If set then the port with a given name is used as primary when multiple ports are defined for a service.</td>
</tr>
<tr>
<td><code>spring.cloud.kubernetes.discovery.service-labels</code></td>
<td><code>null</code></td>
<td>If set, then only the services matching these labels will be fetched from the Kubernetes API server.</td>
</tr>
<tr>
<td><code>spring.cloud.kubernetes.discovery.service-name</code></td>
<td><code>unknown</code></td>
<td>The service name of the local instance.</td>
</tr>
<tr>
<td><code>spring.cloud.kubernetes.discovery.wait-cache-ready</code></td>
<td><code>true</code></td>
<td></td>
</tr>
<tr>
<td><code>spring.cloud.kubernetes.enabled</code></td>
<td><code>true</code></td>
<td>Whether to enable Kubernetes integration.</td>
</tr>
<tr>
<td><code>spring.cloud.kubernetes.leader.auto-startup</code></td>
<td><code>true</code></td>
<td>Should leader election be started automatically on startup. Default: true</td>
</tr>
<tr>
<td><code>spring.cloud.kubernetes.leader.config-map-name</code></td>
<td><code>leaders</code></td>
<td>Kubernetes ConfigMap where leaders information will be stored. Default: leaders</td>
</tr>
<tr>
<td><code>spring.cloud.kubernetes.leader.enabled</code></td>
<td><code>true</code></td>
<td>Should leader election be enabled. Default: true</td>
</tr>
<tr>
<td><code>spring.cloud.kubernetes.leader.id-prefix</code></td>
<td><code>leader.id.</code></td>
<td>Leader id property prefix for the ConfigMap. Default: leader.id.</td>
</tr>
<tr>
<td><code>spring.cloud.kubernetes.leader.namespace</code></td>
<td><code>null</code></td>
<td>Kubernetes namespace where the leaders ConfigMap and candidates are located.</td>
</tr>
<tr>
<td><code>spring.cloud.kubernetes.leader.publish-failed-events</code></td>
<td><code>false</code></td>
<td>Enable/disable publishing events in case leadership acquisition fails. Default: false</td>
</tr>
<tr>
<td><code>spring.cloud.kubernetes.leader.role</code></td>
<td><code>null</code></td>
<td>Role for which leadership this candidate will compete.</td>
</tr>
<tr>
<td><code>spring.cloud.kubernetes.leader.update-period</code></td>
<td><code>60000ms</code></td>
<td>Leadership status check period. Default: 60s</td>
</tr>
</tbody>
</table>
|spring.cloud.kubernetes.loadbalancer.cluster-domain | `cluster.local` | cluster domain.
|spring.cloud.kubernetes.loadbalancer.enabled | `true` | Load balancer enabled, default true.
|spring.cloud.kubernetes.loadbalancer.mode | `` | {@link KubernetesLoadBalancerMode} setting load balancer server list with ip of pod or service name. Default value is POD.
|spring.cloud.kubernetes.loadbalancer.port-name | `http` | Service port name.
|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 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` | Enables the Secrets property source locator.
|spring.cloud.kubernetes.secrets.labels | `` |
|spring.cloud.kubernetes.secrets.name | `` |
|spring.cloud.kubernetes.secrets.namespace | `` |
|spring.cloud.kubernetes.secrets.paths | `` |
|spring.cloud.kubernetes.secrets.sources | `` | 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 &lt;code&gt;StringToDurationConverter&lt;/code&gt;. @see &lt;a href="#x3D; "&gt;https://github.com/spring-projects/spring-boot/blob/master/spring-boot-project/spring-boot/src/main/java/org/springframework/boot/convert/StringToDurationConverter.java"&gt;&lt;/a&gt;.
|spring.cloud.loadbalancer.configurations | `default` | Enables a predefined LoadBalancer configuration.
|spring.cloud.loadbalancer.enabled | `true` | Enables Spring Cloud LoadBalancer.
|spring.cloud.loadbalancer.eureka.approximate-zone-from-hostname |  'false' | Used to determine whether we should try to get the `zone` value from host name.
|spring.cloud.loadbalancer.health-check.initial-delay |  '0' | 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 |  '' | Indicates whether the instances should be refetched by the HealthCheckServiceInstanceListSupplier. This can be used if the instances can be updated and the underlying delegate does not provide an ongoing flux.
|spring.cloud.loadbalancer.health-check.refetch-instances |  'false' | Indicates whether the instances should be refetched by the HealthCheckServiceInstanceListSupplier. This can be used if the instances can be updated and the underlying delegate does not provide an ongoing flux.
|spring.cloud.loadbalancer.health-check.refetch-instances-interval |  '25s' | Interval for refetching available service instances.
|spring.cloud.loadbalancer.health-check.repeat-health-check |  'true' | Indicates whether health checks should keep repeating. It might be useful to set it to `false` if periodically refetching the instances, as every refetch will also trigger a health check.
|spring.cloud.loadbalancer.hint |  '' | Allows setting the value of the `hint` that is passed on to the LoadBalancer request and can subsequently be used in ReactiveLoadBalancer implementations.
|spring.cloud.loadbalancer.hint-header-name |  'X-SC-LB-Hint' | Allows setting the name of the header used for passing the hint for hint-based service instance filtering.
|spring.cloud.loadbalancer.retry.avoid-previous-instance |  'true' | Enables wrapping ServiceInstanceListSupplier beans with RetryAwareServiceInstanceListSupplier; if Spring-Retry is in the classpath.
|spring.cloud.loadbalancer.retry.backoff.enabled |  'false' | Indicates whether Reactor Retry backoffs should be applied.
|spring.cloud.loadbalancer.retry.backoff.jitter |  '0.5' | Used to set the jitter of the RetryBackoffSpec.
|spring.cloud.loadbalancer.retry.backoff.max-backoff |  '' | Used to set the maxBackoff of the RetryBackoffSpec.
|spring.cloud.loadbalancer.retry.backoff.min-backoff |  '5ms' | Used to set the minBackoff of the RetryBackoffSpec.
|spring.cloud.loadbalancer.retry.enabled |  'true' | Enables LoadBalancer retries.
|spring.cloud.loadbalancer.retry.max-retries-on-next-service-instance |  '1' | Number of retries to be executed on the next `ServiceInstance`. A `ServiceInstance` is chosen before each retry call.
|spring.cloud.loadbalancer.retry.max-retries-on-same-service-instance |  '0' | Number of retries to be executed on the same `ServiceInstance`.
|spring.cloud.loadbalancer.retry.retry-on-all-operations |  'false' | Indicates retries should be attempted on operations other than `GET`.
|spring.cloud.loadbalancer.retry.retryable-status-codes |  '' | A `Set` of status codes that should trigger a retry.
|spring.cloud.loadbalancer.ribbon.enabled |  'true' | Causes RibbonLoadBalancerClient 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.sticky-session.add-service-instance-cookie |  'false' | Indicates whether a cookie with the newly selected instance should be added by SC
LoadBalancer.

| spring.cloud.loadbalancer.sticky-session.instance-id-cookie-name | `sc-lb-instance-id` | The name of the cookie holding the preferred instance id. |
| spring.cloud.loadbalancer.zone | `''` | Spring Cloud LoadBalancer zone. |
| spring.cloud.refresh.additional-property-sources-to-retain | `''` | Additional property sources to retain during a refresh. Typically only system property sources are retained. This property allows property sources, such as property sources created by EnvironmentPostProcessors to be retained as well. |
| 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 AutoServiceRegistration. 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.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 binding; `spring.cloud.stream.bindings.input.contentType=text/plain`. For example; spring.cloud.stream.bindings.rabbit1.environment. . . , spring.cloud.stream.bindings.rabbit2.environment. . . |
| 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-destination-cache-size | `10` | The maximum size of Least Recently Used (LRU) cache of dynamic destinations. Once this size is reached, new destinations will trigger the removal of old destinations. Default: 10 |
| 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` | |
| spring.cloud.stream.function.bindings | `''` | |
| spring.cloud.stream.function.definition | `''` | |
| 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 |
| 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 |
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 &quot;spring.cloud.stream.bindings.foo.consumer.instance-index-list&quot; where &amp;x27;foo&amp;x27; is the name of the binding. This setting will override the one set in &amp;x27;spring.cloud.stream.instance-index&amp;x27;

```
|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 &quot;spring.cloud.stream.bindings.foo.consumer.instance-index-list&quot; where &amp;x27;foo&amp;x27; is the name of the binding. This setting will override the one set in &amp;x27;spring.cloud.stream.instance-index&amp;x27;
```

Message header names that will NOT be copied from the inbound message.

```
|spring.cloud.stream.metrics.export-properties | '' | List of properties that are going to be appended to each message. This gets populate 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 an unique value per application. Defaults to:
```

```
|spring.application.name:=${vcap.application.name:=${spring.config.name:application}}
```

```
|spring.cloud.stream.metrics.meter-filter | '' | Pattern to control the meters that will be captured. By default all meters will be captured. For example, &amp;x27;spring.integration.*&amp;x27; will only capture metric information for meters whose name starts with &amp;x27;spring.integration&amp;x27;.
```

```
|spring.cloud.stream.metrics.properties | '' | Application properties that should be added to the metrics payload. For example: &amp;x60;spring.application**&amp;x60;
```

```
|spring.cloud.stream.metrics.schedule-interval | '60s' | Interval expressed as Duration for scheduling metrics snapshots publishing. Defaults to 60 seconds
```

```
|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.pollable-source | 'none' | A semi-colon delimited list of binding names of pollable sources. Binding names follow the same naming convention as functions. For example, name &amp;x27;pollable-source&amp;x27; will be accessible as &amp;x27;foo&amp;x27;
```

```
|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.poller.time-unit | '' | The TimeUnit to apply to delay values.
```

```
|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 &gt; Stream)
```

```
|spring.cloud.task.batch.command-line-runner-order | '0' | The order for the {code CommandLineRunner} used to run batch jobs when {code spring.cloud.task.batch.fail-on-
<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>spring.cloud.task.batch.events.chunk-enabled</td>
<td><code>true</code>.</td>
<td>This property is used to determine if a task should listen for batch chunk events.</td>
</tr>
<tr>
<td>spring.cloud.task.batch.events.enabled</td>
<td><code>true</code>.</td>
<td>This property is used to determine if a task should listen for batch events.</td>
</tr>
<tr>
<td>spring.cloud.task.batch.events.item-process.enabled</td>
<td><code>true</code>.</td>
<td>This property is used to determine if a task should listen for batch item processed events.</td>
</tr>
<tr>
<td>spring.cloud.task.batch.events.item-read.enabled</td>
<td><code>true</code>.</td>
<td>This property is used to determine if a task should listen for batch item read events.</td>
</tr>
<tr>
<td>spring.cloud.task.batch.events.job-execution.enabled</td>
<td><code>true</code>.</td>
<td>This property is used to determine if a task should listen for batch job execution events.</td>
</tr>
<tr>
<td>spring.cloud.task.batch.events.skip.enabled</td>
<td><code>true</code>.</td>
<td>This property is used to determine if a task should listen for batch skip events.</td>
</tr>
<tr>
<td>spring.cloud.task.batch.events.step-execution.enabled</td>
<td><code>true</code>.</td>
<td>This property is used to determine if a task should listen for batch step execution events.</td>
</tr>
<tr>
<td>spring.cloud.task.batch.fail-on-job-failure</td>
<td><code>false</code>.</td>
<td>This property is used to determine if a task should return with a non zero exit code if a batch job fails.</td>
</tr>
<tr>
<td>spring.cloud.task.batch.fail-on-job-failure-poll-interval</td>
<td><code>5000</code>.</td>
<td>Fixed delay in milliseconds that Spring Cloud Task will wait when checking if <code>{@link org.springframework.batch.core.JobExecution}s have completed, when </code>spring.cloud.task.batch.failOnJobFailure` is set to true. Defaults to 5000.</td>
</tr>
<tr>
<td>spring.cloud.task.batch.job-names</td>
<td><code>@#{job1,job2}@#{job3}</code>.</td>
<td>Comma-separated list of job names to execute on startup (for instance, <code>@#{job1,job2}@#{job3}</code>). By default, all Jobs found in the context are executed. @deprecated use <code>spring.batch.job.names</code> instead of <code>spring.cloud.task.batch.jobNames</code>.</td>
</tr>
<tr>
<td>spring.cloud.task.batch.listener.enabled</td>
<td><code>true</code>.</td>
<td>This property is used to determine if a task will be linked to the batch jobs that are run.</td>
</tr>
<tr>
<td>spring.cloud.task.closecontext-enabled</td>
<td><code>false</code>.</td>
<td>When set to true the context is closed at the end of the task. Else the context remains open.</td>
</tr>
</tbody>
</table>
| spring.cloud.task.events.enabled | `true`. | This property is used to determine if a
task app should emit task events.

- `spring.cloud.task.executionid` | '' | An id that will be used by the task when updating the task execution.
- `spring.cloud.task.external-execution-id` | '' | An id that can be associated with a task.
- `spring.cloud.task.initialize-enabled` | '' | If set to true then tables are initialized. If set to false tables are not initialized. Defaults to null. The requirement for it to be defaulted to null is so that we can support the `spring.cloud.task.initialize.enabled` until it is removed.
- `spring.cloud.task.parent-execution-id` | '' | The id of the parent task execution id that launched this task execution. Defaults to null if task execution had no parent.
- `spring.cloud.task.single-instance-enabled` | 'false' | This property is used to determine if a task will execute if another task with the same app name is running.
- `spring.cloud.task.single-instance-lock-check-interval` | '500' | Declares the time (in millis) that a task execution will wait between checks. Default time is: 500 millis.
- `spring.cloud.task.single-instance-lock-ttl` | '' | Declares the maximum amount of time (in millis) that a task execution can hold a lock to prevent another task from executing with a specific task name when the single-instance-enabled is set to true. Default time is: Integer.MAX_VALUE.
- `spring.cloud.task.table-prefix` | 'TASK_' | The prefix to append to the table names created by Spring Cloud Task.
- `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.
- `spring.cloud.vault.authentication` | '' | URL of the AWS-EC2 PKCS7 identity document.
- `spring.cloud.vault.aws-ec2.aws-ec2-path` | 'aws-ec2' | Mount path of the AWS-EC2 authentication backend.
- `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.credential-type | `` | aws credential type
| spring.cloud.vault.aws.enabled | `false` | Enable aws backend usage.
| spring.cloud.vault.aws.role | `` | Role name for credentials.
| spring.cloud.vault.aws.role-arn | `` | Role arn for assumed_role in case we have multiple roles associated with the vault role. @since 3.0.2
| spring.cloud.vault.aws.secret-key-property | `cloud.aws.credentials.secretKey` | Target property for the obtained secret key.
| spring.cloud.vault.aws.session-token-key-property | `cloud.aws.credentials.sessionToken` | Target property for the obtained secret key.
| spring.cloud.vault.aws.ttl | `0` | TTL for sts tokens. Defaults to whatever the vault Role may have for Max. Also limited to what AWS supports to be the max for STS. @since 3.0.2
| spring.cloud.vault.azure-msi.azure-path | `azure` | Mount path of the Azure MSI authentication backend.
| spring.cloud.vault.azure-msi.identity-token-service | `` | Identity token service URI. @since 3.0
| spring.cloud.vault.azure-msi.metadata-service | `` | Instance metadata service URI. @since 3.0
| 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.
| 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.
| spring.cloud.vault.consul.role | `` | Role name for credentials.
<p>|spring.cloud.vault.consul.token-property | <code>spring.cloud.consul.token</code> | Target property for the obtained token. |
|spring.cloud.vault.couchbase.backend | <code>database</code> | Couchbase backend path. |
|spring.cloud.vault.couchbase.enabled | <code>false</code> | Enable couchbase backend usage. |
|spring.cloud.vault.couchbase.password-property | <code>spring.couchbase.password</code> | Target property for the obtained password. |
|spring.cloud.vault.couchbase.role | <code>''</code> | Role name for credentials. |
|spring.cloud.vault.couchbase.static-role | <code>false</code> | Enable static role usage. |
|spring.cloud.vault.couchbase.username-property | <code>spring.couchbase.username</code> | Target property for the obtained username. |
|spring.cloud.vault.database.backend | <code>database</code> | Database backend path. |
|spring.cloud.vault.database.enabled | <code>false</code> | Enable database backend usage. |
|spring.cloud.vault.database.password-property | <code>spring.datasource.password</code> | Target property for the obtained password. |
|spring.cloud.vault.database.role | <code>''</code> | Role name for credentials. |
|spring.cloud.vault.database.static-role | <code>false</code> | Enable static role usage. |
|spring.cloud.vault.database.username-property | <code>spring.datasource.username</code> | Target property for the obtained username. |
|spring.cloud.vault.discovery.enabled | <code>false</code> | Flag to indicate that Vault server discovery is enabled (vault server URL will be looked up via discovery). |
|spring.cloud.vault.discovery.service-id | <code>vault</code> | Service id to locate Vault. |
|spring.cloud.vault.elasticsearch.backend | <code>database</code> | Database backend path. |
|spring.cloud.vault.elasticsearch.enabled | <code>false</code> | Enable elasticsearch backend usage. |
|spring.cloud.vault.elasticsearch.password-property | <code>spring.elasticsearch.rest.password</code> | Target property for the obtained password. |
|spring.cloud.vault.elasticsearch.role | <code>''</code> | Role name for credentials. |
|spring.cloud.vault.elasticsearch.static-role | <code>false</code> | Enable static role usage. |
|spring.cloud.vault.elasticsearch.username-property | <code>spring.elasticsearch.rest.username</code> | Target property for the obtained username. |
|spring.cloud.vault.enabled | <code>true</code> | Enable Vault config server. |
|spring.cloud.vault.fail-fast | <code>false</code> | Fail fast if data cannot be obtained from Vault. |
|spring.cloud.vault.gcp-gce.gcp-path | <code>gcp</code> | Mount path of the Kubernetes authentication backend. |
|spring.cloud.vault.gcp-gce.role | <code>''</code> | Name of the role against which the login is being attempted. |
|spring.cloud.vault.gcp-gce.service-account | <code>''</code> | Optional service account id. Using the default id if left unconfigured. |
|spring.cloud.vault.gcp-iam.credentials.encoded-key | <code>''</code> | The base64 encoded contents of an OAuth2 account private key in JSON format. |
|spring.cloud.vault.gcp-iam.credentials.location | <code>''</code> | Location of the OAuth2 credentials private key. &lt;p&gt;Since this is a Resource, the private key can be in a multitude of locations, such as a local file system, classpath, URL, etc. &lt;/p&gt; |
|spring.cloud.vault.gcp-iam.gcp-path | <code>gcp</code> | Mount path of the Kubernetes authentication backend. |
|spring.cloud.vault.gcp-iam.jwt-validity | <code>15m</code> | Validity of the JWT token. |
|spring.cloud.vault.gcp-iam.project-id | <code>''</code> | Overrides the GCP project Id. |
|spring.cloud.vault.gcp-iam.role | <code>''</code> | Name of the role against which the login is being attempted. |
|spring.cloud.vault.gcp-iam.service-account-id | <code>''</code> | Overrides the GCP service account id. |</p>
<table>
<thead>
<tr>
<th>spring.cloud.vault.host</th>
<th><code>localhost</code></th>
<th>Vault server host.</th>
</tr>
</thead>
<tbody>
<tr>
<td>spring.cloud.vault.kubernetes.kubernetes-path</td>
<td><code>kubernetes</code></td>
<td>Mount path of the Kubernetes authentication backend.</td>
</tr>
<tr>
<td>spring.cloud.vault.kubernetes.role</td>
<td><code>''</code></td>
<td>Name of the role against which the login is being attempted.</td>
</tr>
<tr>
<td>spring.cloud.vault.kubernetes.service-account-token-file</td>
<td><code>/var/run/secrets/kubernetes.io/serviceaccount/token</code></td>
<td>Path to the service account token file.</td>
</tr>
<tr>
<td>spring.cloud.vault.kv.application-name</td>
<td><code>application</code></td>
<td>Application name to be used for the context.</td>
</tr>
<tr>
<td>spring.cloud.vault.kv.backend</td>
<td><code>secret</code></td>
<td>Name of the default backend.</td>
</tr>
<tr>
<td>spring.cloud.vault.kvbackend-version</td>
<td><code>'2'</code></td>
<td>Key-Value backend version. Currently supported versions are: &lt;ul&gt; &lt;li&gt; Version 1 (unversioned key-value backend). &lt;/li&gt; &lt;li&gt; Version 2 (versioned key-value backend). &lt;/li&gt; &lt;/ul&gt;</td>
</tr>
<tr>
<td>spring.cloud.vault.kv.default-context</td>
<td><code>application</code></td>
<td>Name of the default context.</td>
</tr>
<tr>
<td>spring.cloud.vault.kv.enabled</td>
<td><code>true</code></td>
<td>Enable the key-value backend.</td>
</tr>
<tr>
<td>spring.cloud.vault.kv.profile-separator</td>
<td><code>/</code></td>
<td>Profile-separator to combine application name and profile.</td>
</tr>
<tr>
<td>spring.cloud.vault.kv.profiles</td>
<td><code>''</code></td>
<td>List of active profiles. @since 3.0</td>
</tr>
<tr>
<td>spring.cloud.vault.mongodb.backend</td>
<td><code>mongodb</code></td>
<td>MongoDB backend path.</td>
</tr>
<tr>
<td>spring.cloud.vault.mongodb.enabled</td>
<td><code>false</code></td>
<td>Enable mongodb backend usage.</td>
</tr>
<tr>
<td>spring.cloud.vault.mongodb.password-property</td>
<td><code>spring.data.mongodb.password</code></td>
<td>Target property for the obtained password.</td>
</tr>
<tr>
<td>spring.cloud.vault.mongodb.role</td>
<td><code>''</code></td>
<td>Role name for credentials.</td>
</tr>
<tr>
<td>spring.cloud.vault.mongodb.static-role</td>
<td><code>false</code></td>
<td>Enable static role usage. @since 2.2</td>
</tr>
<tr>
<td>spring.cloud.vault.mongodb.username-property</td>
<td><code>spring.data.mongodb.username</code></td>
<td>Target property for the obtained username.</td>
</tr>
<tr>
<td>spring.cloud.vault.mysql.backend</td>
<td><code>mysql</code></td>
<td>mysql backend path.</td>
</tr>
<tr>
<td>spring.cloud.vault.mysql.enabled</td>
<td><code>false</code></td>
<td>Enable mysql backend usage.</td>
</tr>
<tr>
<td>spring.cloud.vault.mysql.password-property</td>
<td><code>spring.datasource.password</code></td>
<td>Target property for the obtained username.</td>
</tr>
<tr>
<td>spring.cloud.vault.mysql.role</td>
<td><code>''</code></td>
<td>Role name for credentials.</td>
</tr>
<tr>
<td>spring.cloud.vault.mysql.username-property</td>
<td><code>spring.datasource.username</code></td>
<td>Target property for the obtained username.</td>
</tr>
<tr>
<td>spring.cloud.vault.namespace</td>
<td><code>''</code></td>
<td>Vault namespace (requires Vault Enterprise).</td>
</tr>
<tr>
<td>spring.cloud.vault.pcf.instance-certificate</td>
<td><code>''</code></td>
<td>Path to the instance certificate (PEM). Defaults to <code>{code CF_INSTANCE_CERT}</code> env variable.</td>
</tr>
<tr>
<td>spring.cloud.vault.pcf.instance-key</td>
<td><code>''</code></td>
<td>Path to the instance key (PEM). Defaults to <code>{code CF_INSTANCE_KEY}</code> env variable.</td>
</tr>
<tr>
<td>spring.cloud.vault.pcfpcf-path</td>
<td><code>pcf</code></td>
<td>Mount path of the Kubernetes authentication backend.</td>
</tr>
<tr>
<td>spring.cloud.vault.pcf.role</td>
<td><code>''</code></td>
<td>Name of the role against which the login is being attempted.</td>
</tr>
<tr>
<td>spring.cloud.vault.port</td>
<td><code>8200</code></td>
<td>Vault server port.</td>
</tr>
<tr>
<td>spring.cloud.vault.postgresql.backend</td>
<td><code>postgresql</code></td>
<td>postgresql backend path.</td>
</tr>
<tr>
<td>spring.cloud.vault.postgresql.enabled</td>
<td><code>false</code></td>
<td>Enable postgresql backend usage.</td>
</tr>
<tr>
<td>spring.cloud.vault.postgresql.password-property</td>
<td><code>spring.datasource.password</code></td>
<td>Target property for the obtained username.</td>
</tr>
<tr>
<td>spring.cloud.vault.postgresql.role</td>
<td>``</td>
<td>Role name for credentials.</td>
</tr>
<tr>
<td>spring.cloud.vault.postgresql.username-property</td>
<td><code>spring.datasource.username</code></td>
<td>Target property for the obtained username.</td>
</tr>
<tr>
<td>spring.cloud.vault.rabbitmq.backend</td>
<td><code>rabbitmq</code></td>
<td>rabbitmq backend path.</td>
</tr>
<tr>
<td>spring.cloud.vault.rabbitmq.enabled</td>
<td><code>false</code></td>
<td>Enable rabbitmq backend usage.</td>
</tr>
<tr>
<td>spring.cloud.vault.rabbitmq.password-property</td>
<td><code>spring.rabbitmq.password</code></td>
<td>Target property for the obtained password.</td>
</tr>
<tr>
<td>spring.cloud.vault.rabbitmq.role</td>
<td>``</td>
<td>Role name for credentials.</td>
</tr>
<tr>
<td>spring.cloud.vault.rabbitmq.username-property</td>
<td><code>spring.rabbitmq.username</code></td>
<td>Target property for the obtained username.</td>
</tr>
<tr>
<td>spring.cloud.vault.read-timeout</td>
<td><code>15000</code></td>
<td>Read timeout.</td>
</tr>
<tr>
<td>spring.cloud.vault.scheme</td>
<td><code>https</code></td>
<td>Protocol scheme. Can be either <code>http</code> or <code>https</code>.</td>
</tr>
<tr>
<td>spring.cloud.vault.session.lifecycle.enabled</td>
<td><code>true</code></td>
<td>Enable session lifecycle management.</td>
</tr>
<tr>
<td>spring.cloud.vault.session.lifecycle.expiry-threshold</td>
<td><code>7s</code></td>
<td>The expiry threshold for a @link LoginToken. The threshold represents a minimum TTL duration to consider a login token as valid. Tokens with a shorter TTL are considered expired and are not used anymore. Should be greater than @code refreshBeforeExpiry to prevent token expiry.</td>
</tr>
<tr>
<td>spring.cloud.vault.session.lifecycle.refresh-before-expiry</td>
<td><code>5s</code></td>
<td>The time period that is at least required before renewing the @link LoginToken.</td>
</tr>
<tr>
<td>spring.cloud.vault.ssl.cert-auth-path</td>
<td><code>cert</code></td>
<td>Mount path of the TLS cert authentication backend.</td>
</tr>
<tr>
<td>spring.cloud.vault.ssl.enabled-cipher-suites</td>
<td>``</td>
<td>List of enabled SSL/TLS cipher suites. @since 3.0.2</td>
</tr>
<tr>
<td>spring.cloud.vault.ssl.enabled-protocols</td>
<td>``</td>
<td>List of enabled SSL/TLS protocol. @since 3.0.2</td>
</tr>
<tr>
<td>spring.cloud.vault.ssl.key-store</td>
<td>``</td>
<td>Trust store that holds certificates and private keys.</td>
</tr>
<tr>
<td>spring.cloud.vault.ssl.key-store-password</td>
<td>``</td>
<td>Password used to access the key store.</td>
</tr>
<tr>
<td>spring.cloud.vault.ssl.key-store-type</td>
<td>``</td>
<td>Type of the key store. @since 3.0</td>
</tr>
<tr>
<td>spring.cloud.vault.ssl.trust-store</td>
<td>``</td>
<td>Trust store that holds SSL certificates.</td>
</tr>
<tr>
<td>spring.cloud.vault.ssl.trust-store-password</td>
<td>``</td>
<td>Password used to access the trust store.</td>
</tr>
<tr>
<td>spring.cloud.vault.ssl.trust-store-type</td>
<td>``</td>
<td>Type of the trust store. @since 3.0</td>
</tr>
<tr>
<td>spring.cloud.vault.token</td>
<td>``</td>
<td>Static vault token. Required if @link #authentication is @code TOKEN.</td>
</tr>
<tr>
<td>spring.cloud.vault.uri</td>
<td>``</td>
<td>Vault URI. Can be set with scheme, host and port.</td>
</tr>
<tr>
<td>spring.cloud.zookeeper.base-sleep-time-ms</td>
<td><code>50</code></td>
<td>Initial amount of time to wait between retries.</td>
</tr>
<tr>
<td>spring.cloud.zookeeper.block-until-connected-unit</td>
<td>``</td>
<td>The unit of time related to blocking on connection to Zookeeper.</td>
</tr>
<tr>
<td>spring.cloud.zookeeper.block-until-connected-wait</td>
<td><code>10</code></td>
<td>Wait time to block on connection to Zookeeper.</td>
</tr>
<tr>
<td>spring.cloud.zookeeper.config.default-context</td>
<td><code>application</code></td>
<td>The name of the default context.</td>
</tr>
<tr>
<td>spring.cloud.zookeeper.config.enabled</td>
<td><code>true</code></td>
<td></td>
</tr>
<tr>
<td>spring.cloud.zookeeper.config.fail-fast</td>
<td><code>true</code></td>
<td>Throw exceptions during config lookup if true, otherwise, log warnings.</td>
</tr>
</tbody>
</table>
| spring.cloud.zookeeper.config.name | `""` | Alternative to spring.application.name to use in looking up values in zookeeper.
| spring.cloud.zookeeper.config.profile-separator | `"\`,"` | Separator for profile appended to the application name.
| spring.cloud.zookeeper.config.root | `"config"` | Root folder where the configuration for Zookeeper is kept.
| 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 | `""` |
| spring.cloud.zookeeper.dependency-names | `""` |
| spring.cloud.zookeeper.discovery.enabled | `true` | 
| spring.cloud.zookeeper.discovery.initial-status | `""` | The initial status of this instance (defaults to @link StatusConstants#STATUS_UP{}).
| 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 CompositeDiscoveryClient; 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.
| spring.cloud.zookeeper.session-timeout | `""` | The configured/negotiated session timeout in milliseconds. Please refer to &lt;a href="https://cwiki.apache.org/confluence/display/CURATOR/TN148"&gt;Curator's Tech Note 14&lt;/a&gt; to understand how Curator implements connection sessions. @see &lt;a href="https://cwiki.apache.org/confluence/display/CURATOR/TN148"&gt;Curator's Tech Note 14&lt;/a&gt;
| 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 `{@link java.util.concurrent.Executor} bean names that should be ignored and not wrapped in a trace representation. |
| spring.sleuth.baggage.correlation-enabled | `true` | Enables correlating the baggage context with logging contexts. |
| spring.sleuth.baggage.correlation-fields | `` | |
| spring.sleuth.baggage.local-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. |
| spring.sleuth.baggage.tag-fields | `` | |
| spring.sleuth.circuitbreaker.enabled | `true` | Enable Spring Cloud CircuitBreaker instrumentation. |
| spring.sleuth.enabled | `true` | Enable span information propagation when using Feign. |
| 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.function.enabled | `true` | Enable instrumenting of Spring Cloud Function and Spring Cloud Function based projects (e.g. Spring Cloud Stream). |
| spring.sleuth.grpc.enabled | `true` | Enable span information propagation when using GRPC. |
| spring.sleuth.http.enabled | `true` | Enables HTTP support. |
| 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. |
| 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` | JMS remote service name. |
| 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` | Kafka remote service name. |
| spring.sleuth.messaging.kafka.streams.enabled | `false` | Should Kafka Streams be turned on. |
| spring.sleuth.messaging.rabbit.enabled | `true` | Enable tracing of RabbitMQ. |
| spring.sleuth.messaging.rabbit.remote-service-name | `rabbitmq` | Rabbit remote service name. |
| spring.sleuth.mongodb.enabled | `true` | Enable tracing for MongoDB. |
| spring.sleuth.opentracing.enabled | `true` | Enables OpenTracing support. |
| spring.sleuth.propagation.type | `` | Tracing context propagation types. |
| spring.sleuth.quartz.enabled | `true` | Enable tracing for Quartz. |
| 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. @deprecated use explicit value via `{link SleuthReactorProperties#instrumentationType}

| spring.sleuth.reactor.enabled | ‘true’ | When true enables instrumentation for reactor. |
| spring.sleuth.reactor.instrumentation-type | ‘‘’ | |
| 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.*$]’ | Thread names for which spans will not be sampled. |
| spring.sleuth.sampler.probability | ‘‘’ | Probability of requests that should be sampled. E.g. 1.0 - 100% requests should be sampled. The precision is whole-numbers only (i.e. there is no support for 0.1% of the traces). |
| spring.sleuth.sampler.rate | ‘10’ | A rate per second can be a nice choice for low-traffic endpoints as it allows you surge protection. For example, you may never expect the endpoint to get more than 50 requests per second. If there was a sudden surge of traffic, to 5000 requests per second, you would still end up with 50 traces per second. Conversely, if you had a percentage, like 10%, the same surge would end up with 500 traces per second, possibly overloading your storage. Amazon X-Ray includes a rate-limited sampler (named Reservoir) for this purpose. Brave has taken the same approach via the `{link brave.sampler.RateLimitingSampler}`. |
| spring.sleuth.sampler.refresh.enabled | ‘true’ | Enable refresh scope for sampler. |
| spring.sleuth.scheduled.enabled | ‘true’ | Enable tracing for `{link org.springframework.scheduling.annotation.Scheduled}`. |
| spring.sleuth.scheduled.skip-pattern | ‘‘’ | Pattern for the fully qualified name of a class that should be skipped. |
| spring.sleuth.span-filter.additional-span-name-patterns-to-ignore | ‘‘’ | Additional list of span names to ignore. Will be appended to `{link #spanNamePatternsToSkip}`. |
| spring.sleuth.span-filter.enabled | ‘false’ | Will turn on the default Sleuth handler mechanism. Might ignore exporting of certain spans; |
| spring.sleuth.span-filter.span-name-patterns-to-skip | ‘^catalogWatchTaskScheduler$’ | List of span names to ignore. They will not be sent to external systems. |
| 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. |
| spring.sleuth.tracer.mode | ‘‘’ | Set which tracer implementation should be picked. |
| 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
<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>spring.sleuth.web.filter-order</td>
<td>0</td>
<td>Order in which the tracing filters should be registered.</td>
</tr>
<tr>
<td>spring.sleuth.web.ignore-auto-configured-skip-patterns</td>
<td>false</td>
<td>If set to true, auto-configured skip patterns will be ignored.</td>
</tr>
<tr>
<td>spring.sleuth.web.servlet.enabled</td>
<td>true</td>
<td>Enable servlet instrumentation.</td>
</tr>
<tr>
<td>spring.sleuth.web.skip-pattern</td>
<td></td>
<td></td>
</tr>
<tr>
<td>spring.sleuth.web.webclient.enabled</td>
<td>true</td>
<td>Enable tracing instrumentation for WebClient.</td>
</tr>
<tr>
<td>spring.zipkin.activemq.message-max-bytes</td>
<td>100000</td>
<td>Maximum number of bytes for a given message with spans sent to Zipkin over ActiveMQ.</td>
</tr>
<tr>
<td>spring.zipkin.activemq.queue</td>
<td>zipkin</td>
<td>Name of the ActiveMQ queue where spans should be sent to Zipkin.</td>
</tr>
<tr>
<td>spring.zipkin.api-path</td>
<td></td>
<td>The API path to append to baseUrl (above) as suffix. This applies if you use other monitoring tools, such as New Relic. The trace API doesn’t need the API path, so you can set it to blank (“”) in the configuration.</td>
</tr>
<tr>
<td>spring.zipkin.base-url</td>
<td><a href="http://localhost:9411/">http://localhost:9411/</a></td>
<td>URL of the zipkin query server instance. You can also provide the service id of the Zipkin server if Zipkin server is registered in service discovery (e.g. <a href="https://zipkinserver/">https://zipkinserver/</a>).</td>
</tr>
<tr>
<td>spring.zipkin.compression.enabled</td>
<td>false</td>
<td></td>
</tr>
<tr>
<td>spring.zipkin.discovery-client-enabled</td>
<td></td>
<td>If set to @code false}, will treat the {@code ZipkinProperties#baseUrl} as a URL always.</td>
</tr>
<tr>
<td>spring.zipkin.enabled</td>
<td>true</td>
<td>Enables sending spans to Zipkin.</td>
</tr>
<tr>
<td>spring.zipkin.encoder</td>
<td></td>
<td>Encoding type of spans sent to Zipkin. Set to @code SpanBytesEncoder#JSON_V1} if your server is not recent.</td>
</tr>
<tr>
<td>spring.zipkin.kafka.topic</td>
<td>zipkin</td>
<td>Name of the Kafka topic where spans should be sent to Zipkin.</td>
</tr>
<tr>
<td>spring.zipkin.locator.discovery.enabled</td>
<td>false</td>
<td>Enabling of locating the host name via service discovery.</td>
</tr>
<tr>
<td>spring.zipkin.message-timeout</td>
<td>1</td>
<td>Timeout in seconds before pending spans will be sent in batches to Zipkin.</td>
</tr>
<tr>
<td>spring.zipkin.rabbitmq.addresses</td>
<td></td>
<td>Addresses of the RabbitMQ brokers used to send spans to Zipkin</td>
</tr>
<tr>
<td>spring.zipkin.rabbitmq.queue</td>
<td>zipkin</td>
<td>Name of the RabbitMQ queue where spans should be sent to Zipkin.</td>
</tr>
<tr>
<td>spring.zipkin.sender.type</td>
<td></td>
<td>Means of sending spans to Zipkin.</td>
</tr>
<tr>
<td>spring.zipkin.service.name</td>
<td></td>
<td>The name of the service, from which the Span was sent via HTTP, that should appear in Zipkin.</td>
</tr>
<tr>
<td>stubrunner.amqp.enabled</td>
<td>false</td>
<td>Whether to enable support for Stub Runner and AMQP.</td>
</tr>
<tr>
<td>stubrunner.amqp.mockConnection</td>
<td>true</td>
<td>Whether to enable support for Stub Runner and AMQP mocked connection factory.</td>
</tr>
<tr>
<td>stubrunner.classifier</td>
<td>stubs</td>
<td>The classifier to use by default in ivy coordinates for a stub.</td>
</tr>
<tr>
<td>stubrunner.cloud.consul.enabled</td>
<td>true</td>
<td>Whether to enable stubs registration in Consul.</td>
</tr>
<tr>
<td>stubrunner.cloud.delegate.enabled</td>
<td>true</td>
<td>Whether to enable DiscoveryClientStub Runner implementation.</td>
</tr>
<tr>
<td>stubrunner.cloud.enabled</td>
<td>true</td>
<td>Whether to enable Spring Cloud support for Stub Runner.</td>
</tr>
<tr>
<td>Property</td>
<td>Default</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------------------------------------</td>
<td>---------</td>
<td>------------------------------------------------</td>
</tr>
<tr>
<td>stubrunner.cloud.eureka.enabled</td>
<td><code>true</code></td>
<td>Whether to enable stubs registration in Eureka.</td>
</tr>
<tr>
<td>stubrunner.cloud.loadbalancer.enabled</td>
<td><code>true</code></td>
<td>Whether to enable Stub Runner's Spring Cloud Load Balancer integration.</td>
</tr>
<tr>
<td>stubrunner.cloud.stubbed.discovery.enabled</td>
<td><code>true</code></td>
<td>Whether Service Discovery should be stubbed for Stub Runner. If set to false, stubs will get registered in real service discovery.</td>
</tr>
<tr>
<td>stubrunner.cloud.zookeeper.enabled</td>
<td><code>true</code></td>
<td>Whether to enable stubs registration in Zookeeper.</td>
</tr>
<tr>
<td>stubrunner.consumer-name</td>
<td><code>''</code></td>
<td>You can override the default {code spring.application.name} of this field by setting a value to this parameter.</td>
</tr>
<tr>
<td>stubrunner.delete-stubs-after-test</td>
<td><code>true</code></td>
<td>If set to {code false} will NOT delete stubs from a temporary folder after running tests.</td>
</tr>
<tr>
<td>stubrunner.fail-on-no-stubs</td>
<td><code>true</code></td>
<td>When enabled, this flag will tell stub runner to throw an exception when no stubs / contracts were found.</td>
</tr>
<tr>
<td>stubrunner.generate-stubs</td>
<td><code>false</code></td>
<td>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.</td>
</tr>
<tr>
<td>stubrunner.http-server-stub-configurer</td>
<td><code>''</code></td>
<td>Configuration for an HTTP server stub.</td>
</tr>
<tr>
<td>stubrunner.ids</td>
<td><code>[]</code></td>
<td>The ids of the stubs to run in {quote;ivy{quote; notation {groupId}:artifactId:[version]:[classifier][:port]). {code groupId}, {code classifier}, {code version} and {code port} can be optional.</td>
</tr>
<tr>
<td>stubrunner.ids-to-service-ids</td>
<td><code>''</code></td>
<td>Mapping of Ivy notation based ids to serviceIds inside your application. Example {quote;a:b&amp;quote;: -gt; &amp;quote;mService{quote; &amp;quote;artifactId{quote; &amp;quote;myOtherService{quote;</td>
</tr>
<tr>
<td>stubrunner.integration.enabled</td>
<td><code>true</code></td>
<td>Whether to enable Stub Runner integration with Spring Integration.</td>
</tr>
<tr>
<td>stubrunner.jms.enabled</td>
<td><code>true</code></td>
<td>Whether to enable Stub Runner integration with Spring JMS.</td>
</tr>
<tr>
<td>stubrunner.kafka.enabled</td>
<td><code>true</code></td>
<td>Whether to enable Stub Runner integration with Spring Kafka.</td>
</tr>
<tr>
<td>stubrunner.kafka.initializer.enabled</td>
<td><code>true</code></td>
<td>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.</td>
</tr>
<tr>
<td>stubrunner.mappings-output-folder</td>
<td><code>''</code></td>
<td>Dumps the mappings of each HTTP server to the selected folder.</td>
</tr>
<tr>
<td>stubrunner.max-port</td>
<td><code>15000</code></td>
<td>Max value of a port for the automatically started WireMock server.</td>
</tr>
<tr>
<td>stubrunner.min-port</td>
<td><code>10000</code></td>
<td>Min value of a port for the automatically started WireMock server.</td>
</tr>
<tr>
<td>stubrunner.password</td>
<td><code>''</code></td>
<td>Repository password.</td>
</tr>
<tr>
<td>stubrunner.properties</td>
<td><code>''</code></td>
<td>Map of properties that can be passed to custom {link org.springframework.cloud.contract.stubrunner.StubDownloaderBuilder}.</td>
</tr>
<tr>
<td>stubrunner.proxy-host</td>
<td><code>''</code></td>
<td>Repository proxy host.</td>
</tr>
<tr>
<td>stubrunner.proxy-port</td>
<td><code>''</code></td>
<td>Repository proxy port.</td>
</tr>
<tr>
<td>stubrunner.server-id</td>
<td><code>''</code></td>
<td></td>
</tr>
<tr>
<td>stubrunner.stream.enabled</td>
<td><code>true</code></td>
<td>Whether to enable Stub Runner integration with Spring Cloud Stream.</td>
</tr>
<tr>
<td>stubrunner.stubs-mode</td>
<td><code>''</code></td>
<td>Pick where the stubs should come from.</td>
</tr>
<tr>
<td>stubrunner.stubs-per-consumer</td>
<td><code>false</code></td>
<td>Should only stubs for this particular consumer get registered in HTTP server stub.</td>
</tr>
<tr>
<td>stubrunner.username</td>
<td>``</td>
<td>Repository username.</td>
</tr>
<tr>
<td>---------------------</td>
<td>-----</td>
<td>----------------------</td>
</tr>
<tr>
<td>wiremock.placeholders.enabled</td>
<td><code>true</code></td>
<td>Flag to indicate that http URLs in generated wiremock stubs should be filtered to add or resolve a placeholder for a dynamic port.</td>
</tr>
<tr>
<td>wiremock.reset-mappings-after-each-test</td>
<td><code>false</code></td>
<td></td>
</tr>
<tr>
<td>wiremock.rest-template-ssl-enabled</td>
<td><code>false</code></td>
<td></td>
</tr>
<tr>
<td>wiremock.server.files</td>
<td><code>[]</code></td>
<td></td>
</tr>
<tr>
<td>wiremock.server.https-port</td>
<td><code>-1</code></td>
<td></td>
</tr>
<tr>
<td>wiremock.server.https-port-dynamic</td>
<td><code>false</code></td>
<td></td>
</tr>
<tr>
<td>wiremock.server.port</td>
<td><code>8080</code></td>
<td></td>
</tr>
<tr>
<td>wiremock.server.port-dynamic</td>
<td><code>false</code></td>
<td></td>
</tr>
<tr>
<td>wiremock.server.stubs</td>
<td><code>[]</code></td>
<td></td>
</tr>
</tbody>
</table>

|===

:leveloffset!: 