

Spring Cloud Contract Reference Documentation

Table of Contents

Legal	5
1. Getting Started	5
1.1. Introducing Spring Cloud Contract	5
1.2. A Three-second Tour	10
1.3. Developing Your First Spring Cloud Contract-based Application	14
1.4. Step-by-step Guide to Consumer Driven Contracts (CDC) with Contracts on the Producer Side	29
1.5. Next Steps	50
2. Using Spring Cloud Contract	51
2.1. Provider Contract Testing with Stubs in Nexus or Artifactory	52
2.2. Provider Contract Testing with Stubs in Git	52
2.3. Consumer Driven Contracts with Contracts on the Producer Side	55
2.4. Consumer Driven Contracts with Contracts in an External Repository	55
2.5. Consumer-driven Contracts with Contracts on the Producer Side, Pushed to Git	62
2.6. Provider Contract Testing with Stubs in Artifactory for a non-Spring Application	63
2.7. Provider Contract Testing with Stubs in Artifactory in a Non-JVM World	66
2.8. Provider Contract Testing with REST Docs and Stubs in Nexus or Artifactory	69
2.9. What to Read Next	74
3. Spring Cloud Contract Features	74
3.1. Contract DSL	74
3.2. Contracts for HTTP	98
3.3. Integrations	183
3.4. Messaging	209
3.5. Spring Cloud Contract Stub Runner	227
3.6. Spring Cloud Contract WireMock	264
3.7. Build Tools Integration	269
3.8. What to Read Next	270
4. Maven Project	270
4.1. Adding the Maven Plugin	270
4.2. Maven and Rest Assured 2.0	271
4.3. Using Snapshot and Milestone Versions for Maven	273
4.4. Adding stubs	275
4.5. Run Plugin	275
4.6. Configure plugin	275

4.7. Configuration Options	276
4.8. Single Base Class for All Tests	278
4.9. Using Different Base Classes for Contracts	280
4.10. Invoking Generated Tests	281
4.11. Pushing Stubs to SCM	282
4.12. Maven Plugin and STS	284
4.13. Maven Plugin with Spock Tests	286
5. Gradle Project	288
5.1. Prerequisites	288
5.2. Add Gradle Plugin with Dependencies	289
5.3. Gradle and Rest Assured 2.0	292
5.4. Snapshot Versions for Gradle	293
5.5. Add stubs	293
5.6. Running the Plugin	294
5.7. Default Setup	294
5.8. Configuring the Plugin	295
5.9. Configuration Options	297
5.10. Single Base Class for All Tests	298
5.11. Different Base Classes for Contracts	299
5.12. Invoking Generated Tests	300
5.13. Publishing Stubs to Artifact Repository	300
5.14. Pushing Stubs to SCM	300
5.15. Spring Cloud Contract Verifier on the Consumer Side	301
6. Docker Project	302
6.1. A Short Introduction to Maven, JARs, and Binary Storage	302
6.2. Generating Tests on the Producer Side	303
6.3. Running Stubs on the Consumer Side	311
6.4. Running Contract Tests against Existing Middleware	313
7. Spring Cloud Contract customization	317
7.1. DSL Customization	317
7.2. WireMock Customization	325
7.3. Using the Pluggable Architecture	328
8. "How-to" Guides	340
8.1. Why use Spring Cloud Contract?	340
8.2. How Can I Write Contracts in a Language Other than Groovy?	340
8.3. How Can I Provide Dynamic Values to a Contract?	341
8.4. How to Do Stubs versioning?	343
8.5. How Can I use a Common Repository with Contracts Instead of Storing Them with the Producer?	344
8.6. How Can I Use Git as the Storage for Contracts and Stubs?	355
8.7. How Can I Debug the Request/Response Being Sent by the Generated Tests Client?	363

8.8. How Can I Debug the Mapping, Request, or Response Being Sent by WireMock?	364
8.9. How Can I See What Got Registered in the HTTP Server Stub?	364
8.10. How Can I Reference Text from File?	364
8.11. How Can I Generate Pact, YAML, or X files from Spring Cloud Contract Contracts?	364
8.12. How Can I Work with Transitive Dependencies?	367
8.13. How Can I Generate Spring REST Docs Snippets from the Contracts?	367
8.14. How Can I Use Stubs from a Location	370
8.15. How Can I Generate Stubs at Runtime	371
8.16. How Can I Make The Build Pass if There Are No Contracts or Stubs	371
8.17. How Can I Mark that a Contract Is in Progress	371
Appendix A: Common application properties	371
Spring Cloud Contract Reference Documentation	376
Legal	376
1. Getting Started	376
1.1. Introducing Spring Cloud Contract	376
1.2. A Three-second Tour	381
1.3. Developing Your First Spring Cloud Contract-based Application	385
1.4. Step-by-step Guide to Consumer Driven Contracts (CDC) with Contracts on the Producer Side	400
1.5. Next Steps	421
2. Using Spring Cloud Contract	422
2.1. Provider Contract Testing with Stubs in Nexus or Artifactory	423
2.2. Provider Contract Testing with Stubs in Git	423
2.3. Consumer Driven Contracts with Contracts on the Producer Side	426
2.4. Consumer Driven Contracts with Contracts in an External Repository	426
2.5. Consumer-driven Contracts with Contracts on the Producer Side, Pushed to Git	433
2.6. Provider Contract Testing with Stubs in Artifactory for a non-Spring Application	434
2.7. Provider Contract Testing with Stubs in Artifactory in a Non-JVM World	437
2.8. Provider Contract Testing with REST Docs and Stubs in Nexus or Artifactory	440
2.9. What to Read Next	445
3. Spring Cloud Contract Features	445
3.1. Contract DSL	445
3.2. Contracts for HTTP	469
3.3. Integrations	554
3.4. Messaging	580
3.5. Spring Cloud Contract Stub Runner	598
3.6. Spring Cloud Contract WireMock	635
3.7. Build Tools Integration	640
3.8. What to Read Next	641
4. Maven Project	641
4.1. Adding the Maven Plugin	641

4.2. Maven and Rest Assured 2.0	642
4.3. Using Snapshot and Milestone Versions for Maven	644
4.4. Adding stubs	646
4.5. Run Plugin	646
4.6. Configure plugin	646
4.7. Configuration Options	647
4.8. Single Base Class for All Tests	649
4.9. Using Different Base Classes for Contracts	651
4.10. Invoking Generated Tests	652
4.11. Pushing Stubs to SCM	653
4.12. Maven Plugin and STS	655
4.13. Maven Plugin with Spock Tests	657
5. Gradle Project	659
5.1. Prerequisites	659
5.2. Add Gradle Plugin with Dependencies	660
5.3. Gradle and Rest Assured 2.0	663
5.4. Snapshot Versions for Gradle	664
5.5. Add stubs	664
5.6. Running the Plugin	665
5.7. Default Setup	665
5.8. Configuring the Plugin	666
5.9. Configuration Options	668
5.10. Single Base Class for All Tests	669
5.11. Different Base Classes for Contracts	670
5.12. Invoking Generated Tests	671
5.13. Publishing Stubs to Artifact Repository	671
5.14. Pushing Stubs to SCM	671
5.15. Spring Cloud Contract Verifier on the Consumer Side	672
6. Docker Project	673
6.1. A Short Introduction to Maven, JARs, and Binary Storage	673
6.2. Generating Tests on the Producer Side	674
6.3. Running Stubs on the Consumer Side	682
6.4. Running Contract Tests against Existing Middleware	684
7. Spring Cloud Contract customization	688
7.1. DSL Customization	688
7.2. WireMock Customization	696
7.3. Using the Pluggable Architecture	699
8. “How-to” Guides	711
8.1. Why use Spring Cloud Contract?	711
8.2. How Can I Write Contracts in a Language Other than Groovy?	711
8.3. How Can I Provide Dynamic Values to a Contract?	712

8.4. How to Do Stubs versioning?	714
8.5. How Can I use a Common Repository with Contracts Instead of Storing Them with the Producer?	715
8.6. How Can I Use Git as the Storage for Contracts and Stubs?	726
8.7. How Can I Debug the Request/Response Being Sent by the Generated Tests Client?	734
8.8. How Can I Debug the Mapping, Request, or Response Being Sent by WireMock?	735
8.9. How Can I See What Got Registered in the HTTP Server Stub?	735
8.10. How Can I Reference Text from File?	735
8.11. How Can I Generate Pact, YAML, or X files from Spring Cloud Contract Contracts?	735
8.12. How Can I Work with Transitive Dependencies?	738
8.13. How Can I Generate Spring REST Docs Snippets from the Contracts?	738
8.14. How Can I Use Stubs from a Location.	741
8.15. How Can I Generate Stubs at Runtime	742
8.16. How Can I Make The Build Pass if There Are No Contracts or Stubs	742
8.17. How Can I Mark that a Contract Is in Progress.	742
Appendix B: Common application properties	742

Legal

4.1.0-M1

Copyright © 2012-2020

Copies of this document may be made for your own use and for distribution to others, provided that you do not charge any fee for such copies and further provided that each copy contains this Copyright Notice, whether distributed in print or electronically.

1. Getting Started

If you are getting started with Spring Cloud Contract or Spring in general, start by reading this section. It answers the basic “what?”, “how?” and “why?” questions. It includes an introduction to Spring Cloud Contract, along with installation instructions. We then walk you through building your first Spring Cloud Contract application, discussing some core principles as we go.

1.1. Introducing Spring Cloud Contract

Spring Cloud Contract moves TDD to the level of software architecture. It lets you perform consumer-driven and producer-driven contract testing.

1.1.1. History

Before becoming Spring Cloud Contract, this project was called [Accurest](#). It was created by [Marcin Grzeszczak](#) and [Jakub Kubrynski](#) from ([Codearte](#)).

The **0.1.0** release took place on 26 Jan 2015, and it became stable with **1.0.0** release on 29 Feb 2016.

Why Do You Need It?

Assume that we have a system that consists of multiple microservices, as the following image shows:

[Microservices Architecture] | *Deps.png*

Testing Issues

If we want to test the application in the top left corner of the image in the preceding section to determine whether it can communicate with other services, we could do one of two things:

- Deploy all microservices and perform end-to-end tests.
- Mock other microservices in unit and integration tests.

Both have their advantages but also a lot of disadvantages.

Deploy all microservices and perform end-to-end tests

Advantages:

- Simulates production.
- Tests real communication between services.

Disadvantages:

- To test one microservice, we have to deploy six microservices, a couple of databases, and other items.
- The environment where the tests run is locked for a single suite of tests (nobody else would be able to run the tests in the meantime).
- They take a long time to run.
- The feedback comes very late in the process.
- They are extremely hard to debug.

Mock other microservices in unit and integration tests

Advantages:

- They provide very fast feedback.
- They have no infrastructure requirements.

Disadvantages:

- The implementor of the service creates stubs that might have nothing to do with reality.
- You can go to production with passing tests and failing production.

To solve the aforementioned issues, Spring Cloud Contract was created. The main idea is to give you

very fast feedback, without the need to set up the whole world of microservices. If you work on stubs, then the only applications you need are those that your application directly uses. The following image shows the relationship of stubs to an application:

[Stubbed Services] | *Stubs2.png*

Spring Cloud Contract gives you the certainty that the stubs that you use were created by the service that you call. Also, if you can use them, it means that they were tested against the producer's side. In short, you can trust those stubs.

1.1.2. Purposes

The main purposes of Spring Cloud Contract are:

- To ensure that HTTP and messaging stubs (used when developing the client) do exactly what the actual server-side implementation does.
- To promote the ATDD (acceptance test-driven development) method, and the microservices architectural style.
- To provide a way to publish changes in contracts that are immediately visible on both sides.
- To generate boilerplate test code to be used on the server side.

By default, Spring Cloud Contract integrates with [Wiremock](#) as the HTTP server stub.



Spring Cloud Contract's purpose is NOT to start writing business features in the contracts. Assume that we have a business use case of fraud check. If a user can be a fraud for 100 different reasons, we would assume that you would create two contracts, one for the positive case and one for the negative case. Contract tests are used to test contracts between applications, not to simulate full behavior.

1.1.3. What Is a Contract?

As consumers of services, we need to define what exactly we want to achieve. We need to formulate our expectations. That is why we write contracts. In other words, a contract is an agreement on how the API or message communication should look. Consider the following example:

Assume that you want to send a request that contains the ID of a client company and the amount it wants to borrow from us. You also want to send it to the `/fraudcheck` URL by using the `PUT` method. The following listing shows a contract to check whether a client should be marked as a fraud in both Groovy and YAML:

groovy

```
/*
 * Copyright 2013-2020 the original author or authors.
 *
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 *     https://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */

package contracts

org.springframework.cloud.contract.spec.Contract.make {
    request { // (1)
        method 'PUT' // (2)
        url '/fraudcheck' // (3)
        body([ // (4)
            "client.id": $(regex('[0-9]{10}')),
            loanAmount : 99999
        ])
        headers { // (5)
            contentType('application/json')
        }
    }
    response { // (6)
        status OK() // (7)
        body([ // (8)
            fraudCheckStatus : "FRAUD",
            "rejection.reason": "Amount too high"
        ])
        headers { // (9)
            contentType('application/json')
        }
    }
}

/*
From the Consumer perspective, when shooting a request in the integration test:
```

- (1) - If the consumer sends a request
- (2) - With the "PUT" method
- (3) - to the URL "/fraudcheck"

- (4) - with the JSON body that
 - * has a field `client.id` that matches a regular expression `[0-9]{10}`
 - * has a field `loanAmount` that is equal to `99999`
- (5) - with header `Content-Type` equal to `application/json`
- (6) - then the response will be sent with
- (7) - status equal `200`
- (8) - and JSON body equal to

```
{ "fraudCheckStatus": "FRAUD", "rejectionReason": "Amount too high" }
```
- (9) - with header `Content-Type` equal to `application/json`

From the Producer perspective, in the autogenerated producer-side test:

- (1) - A request will be sent to the producer
 - (2) - With the "PUT" method
 - (3) - to the URL "/fraudcheck"
 - (4) - with the JSON body that
 - * has a field `client.id` that will have a generated value that matches a regular expression `[0-9]{10}`
 - * has a field `loanAmount` that is equal to `99999`
 - (5) - with header `Content-Type` equal to `application/json`
 - (6) - then the test will assert if the response has been sent with
 - (7) - status equal `200`
 - (8) - and JSON body equal to

```
{ "fraudCheckStatus": "FRAUD", "rejectionReason": "Amount too high" }
```
 - (9) - with header `Content-Type` matching `application/json.*`
- */

yaml

```
request: # (1)
  method: PUT # (2)
  url: /yamlfraudcheck # (3)
  body: # (4)
    "client.id": 1234567890
    loanAmount: 99999
  headers: # (5)
    Content-Type: application/json
  matchers:
    body:
      - path: $.['client.id'] # (6)
        type: by_regex
        value: "[0-9]{10}"
response: # (7)
  status: 200 # (8)
  body: # (9)
    fraudCheckStatus: "FRAUD"
    "rejection.reason": "Amount too high"
  headers: # (10)
    Content-Type: application/json
```

```

#From the Consumer perspective, when shooting a request in the integration test:
#
#(1) - If the consumer sends a request
#(2) - With the "PUT" method
#(3) - to the URL "/yamlfraudcheck"
#(4) - with the JSON body that
# * has a field `client.id`
# * has a field `loanAmount` that is equal to `99999`
#(5) - with header `Content-Type` equal to `application/json`
#(6) - and a `client.id` json entry matches the regular expression `[0-9]{10}`
#(7) - then the response will be sent with
#(8) - status equal `200`
#(9) - and JSON body equal to
# { "fraudCheckStatus": "FRAUD", "rejectionReason": "Amount too high" }
#(10) - with header `Content-Type` equal to `application/json`
#
#From the Producer perspective, in the autogenerated producer-side test:
#
#(1) - A request will be sent to the producer
#(2) - With the "PUT" method
#(3) - to the URL "/yamlfraudcheck"
#(4) - with the JSON body that
# * has a field `client.id` `1234567890`
# * has a field `loanAmount` that is equal to `99999`
#(5) - with header `Content-Type` equal to `application/json`
#(7) - then the test will assert if the response has been sent with
#(8) - status equal `200`
#(9) - and JSON body equal to
# { "fraudCheckStatus": "FRAUD", "rejectionReason": "Amount too high" }
#(10) - with header `Content-Type` equal to `application/json`

```



It is expected that contracts are coming from a **trusted source**. You should never download nor interact with contracts coming from untrusted locations.

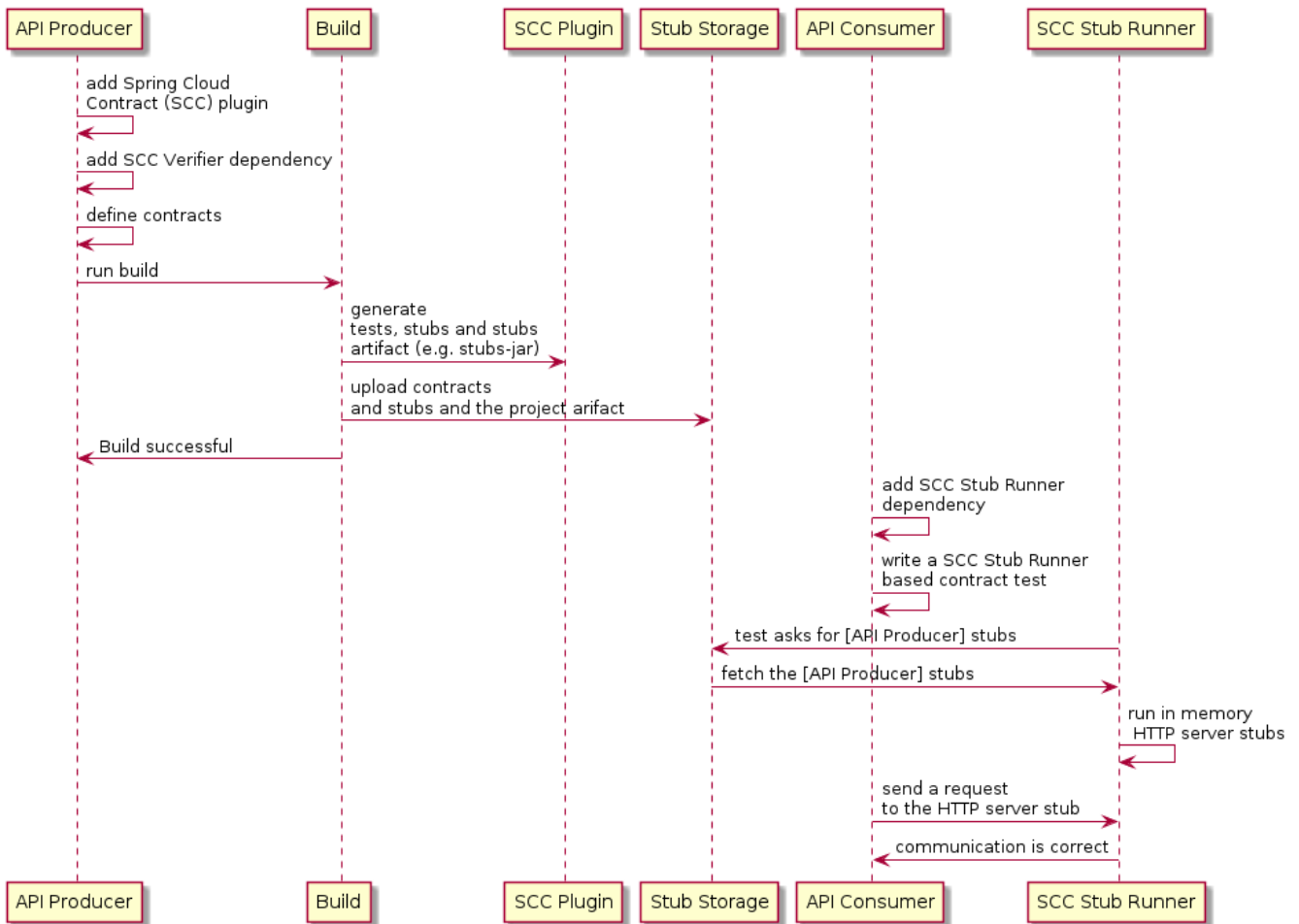
1.2. A Three-second Tour

This very brief tour walks through using Spring Cloud Contract. It consists of the following topics:

- [On the Producer Side](#)
- [On the Consumer Side](#)

You can find a somewhat longer tour [here](#).

The following UML diagram shows the relationship of the parts within Spring Cloud Contract:



1.2.1. On the Producer Side

To start working with Spring Cloud Contract, you can add files with REST or messaging contracts expressed in either Groovy DSL or YAML to the contracts directory, which is set by the `contractsDslDir` property. By default, it is `$rootDir/src/test/resources/contracts`.

Then you can add the Spring Cloud Contract Verifier dependency and plugin to your build file, as the following example shows:

```

<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-contract-verifier</artifactId>
  <scope>test</scope>
</dependency>
  
```

The following listing shows how to add the plugin, which should go in the build/plugins portion of the file:

```
<plugin>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-contract-maven-plugin</artifactId>
  <version>${spring-cloud-contract.version}</version>
  <extensions>>true</extensions>
</plugin>
```

Running `./mvnw clean install` automatically generates tests that verify the application compliance with the added contracts. By default, the tests get generated under `org.springframework.cloud.contract.verifier.tests..`

As the implementation of the functionalities described by the contracts is not yet present, the tests fail.

To make them pass, you must add the correct implementation of either handling HTTP requests or messages. Also, you must add a base test class for auto-generated tests to the project. This class is extended by all the auto-generated tests, and it should contain all the setup information necessary to run them (for example `RestAssuredMockMvc` controller setup or messaging test setup).

The following example, from `pom.xml`, shows how to specify the base test class:

```
<build>
  <plugins>
    <plugin>
      <groupId>org.springframework.cloud</groupId>
      <artifactId>spring-cloud-contract-maven-plugin</artifactId>
      <version>2.1.2.RELEASE</version>
      <extensions>>true</extensions>
      <configuration>
        <baseClassForTests>com.example.contractTest.BaseTestClass</baseClassForTests> ①
      </configuration>
    </plugin>
    <plugin>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-maven-plugin</artifactId>
    </plugin>
  </plugins>
</build>
```

① The `baseClassForTests` element lets you specify your base test class. It must be a child of a `configuration` element within `spring-cloud-contract-maven-plugin`.

Once the implementation and the test base class are in place, the tests pass, and both the application and the stub artifacts are built and installed in the local Maven repository. You can now

merge the changes, and you can publish both the application and the stub artifacts in an online repository.

1.2.2. On the Consumer Side

You can use **Spring Cloud Contract Stub Runner** in the integration tests to get a running WireMock instance or messaging route that simulates the actual service.

To do so, add the dependency to **Spring Cloud Contract Stub Runner**, as the following example shows:

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-contract-stub-runner</artifactId>
  <scope>test</scope>
</dependency>
```

You can get the Producer-side stubs installed in your Maven repository in either of two ways:

- By checking out the Producer side repository and adding contracts and generating the stubs by running the following commands:

```
$ cd local-http-server-repo
$ ./mvnw clean install -DskipTests
```



The tests are being skipped because the producer-side contract implementation is not in place yet, so the automatically-generated contract tests fail.

- By getting already-existing producer service stubs from a remote repository. To do so, pass the stub artifact IDs and artifact repository URL as **Spring Cloud Contract Stub Runner** properties, as the following example shows:

```
stubrunner:
  ids: 'com.example:http-server-dsl+:stubs:8080'
  repositoryRoot: https://repo.spring.io/libs-snapshot
```

Now you can annotate your test class with **@AutoConfigureStubRunner**. In the annotation, provide the **group-id** and **artifact-id** values for **Spring Cloud Contract Stub Runner** to run the collaborators' stubs for you, as the following example shows:

```
@RunWith(SpringRunner.class)
@SpringBootTest(webEnvironment=WebEnvironment.NONE)
@AutoConfigureStubRunner(ids = {"com.example:http-server-dsl:+:stubs:6565"},
    stubsMode = StubRunnerProperties.StubsMode.LOCAL)
public class LoanApplicationServiceTests {
    . . .
}
```



Use the **REMOTE** `stubsMode` when downloading stubs from an online repository and **LOCAL** for offline work.

Now, in your integration test, you can receive stubbed versions of HTTP responses or messages that are expected to be emitted by the collaborator service.

1.3. Developing Your First Spring Cloud Contract-based Application

This brief tour walks through using Spring Cloud Contract. It consists of the following topics:

- [On the Producer Side](#)
- [On the Consumer Side](#)

You can find an even more brief tour [here](#).

For the sake of this example, the **Stub Storage** is Nexus/Artifactory.

The following UML diagram shows the relationship of the parts of Spring Cloud Contract:

[Getting started first application] | *getting-started-three-second.png*

1.3.1. On the Producer Side

To start working with **Spring Cloud Contract**, you can add the Spring Cloud Contract Verifier dependency and plugin to your build file, as the following example shows:

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-contract-verifier</artifactId>
  <scope>test</scope>
</dependency>
```

The following listing shows how to add the plugin, which should go in the `build/plugins` portion of the file:

```
<plugin>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-contract-maven-plugin</artifactId>
  <version>${spring-cloud-contract.version}</version>
  <extensions>>true</extensions>
</plugin>
```

The easiest way to get started is to go to [the Spring Initializr](#) and add “Web” and “Contract Verifier” as dependencies. Doing so pulls in the previously mentioned dependencies and everything else you need in the `pom.xml` file (except for setting the base test class, which we cover later in this section). The following image shows the settings to use in [the Spring Initializr](#):

[Spring Initializr with Web and Contract Verifier] |



start_spring_io_dependencies.png

Now you can add files with **REST/** messaging contracts expressed in either Groovy DSL or YAML to the `contracts` directory, which is set by the `contractsDsldir` property. By default, it is `$rootDir/src/test/resources/contracts`. Note that the file name does not matter. You can organize your contracts within this directory with whatever naming scheme you like.

For the HTTP stubs, a contract defines what kind of response should be returned for a given request (taking into account the HTTP methods, URLs, headers, status codes, and so on). The following example shows an HTTP stub contract in both Groovy and YAML:

groovy

```
package contracts

org.springframework.cloud.contract.spec.Contract.make {
    request {
        method 'PUT'
        url '/fraudcheck'
        body([
            "client.id": $(regex('[0-9]{10}')),
            loanAmount: 99999
        ])
        headers {
            contentType('application/json')
        }
    }
    response {
        status OK()
        body([
            fraudCheckStatus: "FRAUD",
            "rejection.reason": "Amount too high"
        ])
        headers {
            contentType('application/json')
        }
    }
}
```

yaml

```
request:
  method: PUT
  url: /fraudcheck
  body:
    "client.id": 1234567890
    loanAmount: 99999
  headers:
    Content-Type: application/json
  matchers:
    body:
      - path: $.['client.id']
        type: by_regex
        value: "[0-9]{10}"
response:
  status: 200
  body:
    fraudCheckStatus: "FRAUD"
    "rejection.reason": "Amount too high"
  headers:
    Content-Type: application/json;charset=UTF-8
```

If you need to use messaging, you can define:

- The input and output messages (taking into account from where it was sent, the message body, and the header).
- The methods that should be called after the message is received.
- The methods that, when called, should trigger a message.

The following example shows a Camel messaging contract:

groovy

```
def contractDsl = Contract.make {
    name "foo"
    label 'some_label'
    input {
        triggeredBy('bookReturnedTriggered()')
    }
    outputMessage {
        sentTo('activemq:output')
        body(''{ "bookName" : "foo" }''')
        headers {
            header('BOOK-NAME', 'foo')
            messagingContentType(applicationJson())
        }
    }
}
```

yaml

```
label: some_label
input:
  triggeredBy: bookReturnedTriggered
outputMessage:
  sentTo: activemq:output
  body:
    bookName: foo
  headers:
    BOOK-NAME: foo
  contentType: application/json
```

Running `./mvnw clean install` automatically generates tests that verify the application compliance with the added contracts. By default, the generated tests are under `org.springframework.cloud.contract.verifier.tests..`

The generated tests may differ, depending on which framework and test type you have set up in your plugin.

In the next listing, you can find:

- The default test mode for HTTP contracts in `MockMvc`
- A JAX-RS client with the `JAXRS` test mode
- A `WebTestClient`-based test (this is particularly recommended while working with Reactive, `Web-Flux`-based applications) set with the `WEBTESTCLIENT` test mode



You need only one of these test frameworks. `MockMvc` is the default. To use one of the other frameworks, add its library to your classpath.

The following listing shows samples for all frameworks:

mockmvc

```
@Test
public void validate_shouldMarkClientAsFraud() throws Exception {
    // given:
    MockMvcRequestSpecification request = given()
        .header("Content-Type", "application/vnd.fraud.v1+json")
        .body("{\"client.id\":\"1234567890\",\"loanAmount\":99999}");

    // when:
    ResponseOptions response = given().spec(request)
        .put("/fraudcheck");

    // then:
    assertThat(response.statusCode()).isEqualTo(200);
    assertThat(response.header("Content-
Type")).matches("application/vnd.fraud.v1.json.*");
    // and:
    DocumentContext parsedJson =
    JsonPath.parse(response.getBody().asString());
    assertThatJson(parsedJson).field("'fraudCheckStatus']").matches("[A-
Z]{5}");
    assertThatJson(parsedJson).field("'rejection.reason']").isEqualTo("Amount
too high");
}
```

```
@SuppressWarnings("rawtypes")
public class FooTest {
    WebTarget webTarget;

    @Test
    public void validate_() throws Exception {

        // when:
        Response response = webTarget
            .path("/users")
            .queryParams("limit", "10")
            .queryParams("offset", "20")
            .queryParams("filter", "email")
            .queryParams("sort", "name")
            .queryParams("search", "55")
            .queryParams("age", "99")
            .queryParams("name", "Denis.Stepanov")
            .queryParams("email", "bob@email.com")
            .request()
            .build("GET")
            .invoke();
        String responseAsString = response.readEntity(String.class);

        // then:
        assertThat(response.getStatus()).isEqualTo(200);

        // and:
        DocumentContext parsedJson = JsonPath.parse(responseAsString);
        assertThatJson(parsedJson).field("[ 'property1'']").isEqualTo("a");
    }
}
```

webtestclient

```
@Test
public void validate_shouldRejectABeerIfTooYoung() throws Exception {
    // given:
        WebClientRequestSpecification request = given()
            .header("Content-Type", "application/json")
            .body("{\"age\":10}");

    // when:
        WebClientResponse response = given().spec(request)
            .post("/check");

    // then:
        assertThat(response.statusCode()).isEqualTo(200);
        assertThat(response.header("Content-
Type")).matches("application/json.*");
    // and:
        DocumentContext parsedJson =
        JsonPath.parse(response.getBody().asString());
        assertThatJson(parsedJson).field("[ 'status'']").isEqualTo("NOT_OK");
    }
}
```

As the implementation of the functionalities described by the contracts is not yet present, the tests fail.

To make them pass, you must add the correct implementation of handling either HTTP requests or messages. Also, you must add a base test class for auto-generated tests to the project. This class is extended by all the auto-generated tests and should contain all the setup necessary information needed to run them (for example, [RestAssuredMockMvc](#) controller setup or messaging test setup).

The following example, from [pom.xml](#), shows how to specify the base test class:

```

<build>
  <plugins>
    <plugin>
      <groupId>org.springframework.cloud</groupId>
      <artifactId>spring-cloud-contract-maven-plugin</artifactId>
      <version>2.1.2.RELEASE</version>
      <extensions>>true</extensions>
      <configuration>

<baseClassForTests>com.example.contractTest.BaseTestClass</baseClassForTests> ①
      </configuration>
    </plugin>
    <plugin>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-maven-plugin</artifactId>
    </plugin>
  </plugins>
</build>

```

① The `baseClassForTests` element lets you specify your base test class. It must be a child of a `configuration` element within `spring-cloud-contract-maven-plugin`.

The following example shows a minimal (but functional) base test class:

```

package com.example.contractTest;

import org.junit.Before;

import io.restassured.module.mockmvc.RestAssuredMockMvc;

public class BaseTestClass {

    @Before
    public void setup() {
        RestAssuredMockMvc.standaloneSetup(new FraudController());
    }
}

```

This minimal class really is all you need to get your tests to work. It serves as a starting place to which the automatically generated tests attach.

Now we can move on to the implementation. For that, we first need a data class, which we then use in our controller. The following listing shows the data class:


```
package com.example.Test;

import com.fasterxml.jackson.annotation.JsonProperty;

public class LoanRequest {

    @JsonProperty("client.id")
    private String clientId;

    private Long loanAmount;

    public String getClientId() {
        return clientId;
    }

    public void setClientId(String clientId) {
        this.clientId = clientId;
    }

    public Long getLoanAmount() {
        return loanAmount;
    }

    public void setLoanRequestAmount(Long loanAmount) {
        this.loanAmount = loanAmount;
    }
}
```

The preceding class provides an object in which we can store the parameters. Because the client ID in the contract is called `client.id`, we need to use the `@JsonProperty("client.id")` parameter to map it to the `clientId` field.

Now we can move along to the controller, which the following listing shows:

```

package com.example.docTest;

import org.springframework.web.bind.annotation.PutMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class FraudController {

    @PutMapping(value = "/fraudcheck", consumes="application/json",
produces="application/json")
    public String check(@RequestBody LoanRequest loanRequest) { ①

        if (loanRequest.getLoanAmount() > 10000) { ②
            return "{fraudCheckStatus: FRAUD, rejection.reason: Amount too high}";
        } else {
            return "{fraudCheckStatus: OK, acceptance.reason: Amount OK}"; ④
        }
    }
}

```

- ① We map the incoming parameters to a `LoanRequest` object.
- ② We check the requested loan amount to see if it is too much.
- ③ If it is too much, we return the JSON (created with a simple string here) that the test expects.
- ④ If we had a test to catch when the amount is allowable, we could match it to this output.

The `FraudController` is about as simple as things get. You can do much more, including logging, validating the client ID, and so on.

Once the implementation and the test base class are in place, the tests pass, and both the application and the stub artifacts are built and installed in the local Maven repository. Information about installing the stubs jar to the local repository appears in the logs, as the following example shows:

```
[INFO] --- spring-cloud-contract-maven-plugin:1.0.0.BUILD-SNAPSHOT:generateStubs
(default-generateStubs) @ http-server ---
[INFO] Building jar: /some/path/http-server/target/http-server-0.0.1-SNAPSHOT-
stubs.jar
[INFO]
[INFO] --- maven-jar-plugin:2.6:jar (default-jar) @ http-server ---
[INFO] Building jar: /some/path/http-server/target/http-server-0.0.1-SNAPSHOT.jar
[INFO]
[INFO] --- spring-boot-maven-plugin:1.5.5.BUILD-SNAPSHOT:repackage (default) @
http-server ---
[INFO]
[INFO] --- maven-install-plugin:2.5.2:install (default-install) @ http-server ---
[INFO] Installing /some/path/http-server/target/http-server-0.0.1-SNAPSHOT.jar to
/path/to/your/.m2/repository/com/example/http-server/0.0.1-SNAPSHOT/http-server-
0.0.1-SNAPSHOT.jar
[INFO] Installing /some/path/http-server/pom.xml to
/path/to/your/.m2/repository/com/example/http-server/0.0.1-SNAPSHOT/http-server-
0.0.1-SNAPSHOT.pom
[INFO] Installing /some/path/http-server/target/http-server-0.0.1-SNAPSHOT-
stubs.jar to /path/to/your/.m2/repository/com/example/http-server/0.0.1-
SNAPSHOT/http-server-0.0.1-SNAPSHOT-stubs.jar
```

You can now merge the changes and publish both the application and the stub artifacts in an online repository.

1.3.2. On the Consumer Side

You can use Spring Cloud Contract Stub Runner in the integration tests to get a running WireMock instance or messaging route that simulates the actual service.

To get started, add the dependency to [Spring Cloud Contract Stub Runner](#), as follows:

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-contract-stub-runner</artifactId>
  <scope>test</scope>
</dependency>
```

You can get the Producer-side stubs installed in your Maven repository in either of two ways:

- By checking out the Producer side repository and adding contracts and generating the stubs by running the following commands:

```
$ cd local-http-server-repo
$ ./mvnw clean install -DskipTests
```



The tests are skipped because the Producer-side contract implementation is not yet in place, so the automatically-generated contract tests fail.

- By getting existing producer service stubs from a remote repository. To do so, pass the stub artifact IDs and artifact repository URL as **Spring Cloud Contract Stub Runner** properties, as the following example shows:

```
stubrunner:
  ids: 'com.example:http-server-dsl:+:stubs:8080'
  repositoryRoot: https://repo.spring.io/libs-snapshot
```

Now you can annotate your test class with **@AutoConfigureStubRunner**. In the annotation, provide the **group-id** and **artifact-id** for **Spring Cloud Contract Stub Runner** to run the collaborators' stubs for you, as the following example shows:

```
@RunWith(SpringRunner.class)
@SpringBootTest(webEnvironment=WebEnvironment.NONE)
@AutoConfigureStubRunner(ids = {"com.example:http-server-dsl:+:stubs:6565"},
    stubsMode = StubRunnerProperties.StubsMode.LOCAL)
public class LoanApplicationServiceTests {
    . . .
}
```



Use the **REMOTE stubsMode** when downloading stubs from an online repository and **LOCAL** for offline work.

In your integration test, you can receive stubbed versions of HTTP responses or messages that are expected to be emitted by the collaborator service. You can see entries similar to the following in the build logs:

```

2016-07-19 14:22:25.403 INFO 41050 --- [           main]
o.s.c.c.stubrunner.AetherStubDownloader : Desired version is + - will try to
resolve the latest version
2016-07-19 14:22:25.438 INFO 41050 --- [           main]
o.s.c.c.stubrunner.AetherStubDownloader : Resolved version is 0.0.1-SNAPSHOT
2016-07-19 14:22:25.439 INFO 41050 --- [           main]
o.s.c.c.stubrunner.AetherStubDownloader : Resolving artifact com.example:http-
server:jar:stubs:0.0.1-SNAPSHOT using remote repositories []
2016-07-19 14:22:25.451 INFO 41050 --- [           main]
o.s.c.c.stubrunner.AetherStubDownloader : Resolved artifact com.example:http-
server:jar:stubs:0.0.1-SNAPSHOT to /path/to/your/.m2/repository/com/example/http-
server/0.0.1-SNAPSHOT/http-server-0.0.1-SNAPSHOT-stubs.jar
2016-07-19 14:22:25.465 INFO 41050 --- [           main]
o.s.c.c.stubrunner.AetherStubDownloader : Unpacking stub from JAR [URI:
file:/path/to/your/.m2/repository/com/example/http-server/0.0.1-SNAPSHOT/http-
server-0.0.1-SNAPSHOT-stubs.jar]
2016-07-19 14:22:25.475 INFO 41050 --- [           main]
o.s.c.c.stubrunner.AetherStubDownloader : Unpacked file to
[/var/folders/0p/xwq47sq106x1_g3dtv6qfm940000gq/T/contracts100276532569594265]
2016-07-19 14:22:27.737 INFO 41050 --- [           main]
o.s.c.c.stubrunner.StubRunnerExecutor   : All stubs are now running RunningStubs
[NamesAndPorts={com.example:http-server:0.0.1-SNAPSHOT:stubs=8080}]

```

1.4. Step-by-step Guide to Consumer Driven Contracts (CDC) with Contracts on the Producer Side

Consider an example of fraud detection and the loan issuance process. The business scenario is such that we want to issue loans to people but do not want them to steal from us. The current implementation of our system grants loans to everybody.

Assume that **Loan Issuance** is a client to the **Fraud Detection** server. In the current sprint, we must develop a new feature: if a client wants to borrow too much money, we mark the client as a fraud.

Technical remarks

- Fraud Detection has an **artifact-id** of **http-server**.
- Loan Issuance has an **artifact-id** of **http-client**.
- Both have a **group-id** of **com.example**.
- For the sake of this example, the **Stub Storage** is Nexus/Artifactory.

Social remarks

- Both the client and the server development teams need to communicate directly and discuss changes while going through the process.
- CDC is all about communication.

The [server-side code is available here](#), and [the client code is available here](#).



In this case, the producer owns the contracts. Physically, all the contracts are in the producer's repository.

1.4.1. Technical Note

If you use the SNAPSHOT, Milestone, or Release Candidate versions, you need to add the following section to your build:

Maven

```
<repositories>
  <repository>
    <id>spring-snapshots</id>
    <name>Spring Snapshots</name>
    <url>https://repo.spring.io/snapshot</url>
    <snapshots>
      <enabled>true</enabled>
    </snapshots>
  </repository>
  <repository>
    <id>spring-milestones</id>
    <name>Spring Milestones</name>
    <url>https://repo.spring.io/milestone</url>
    <snapshots>
      <enabled>>false</enabled>
    </snapshots>
  </repository>
  <repository>
    <id>spring-releases</id>
    <name>Spring Releases</name>
    <url>https://repo.spring.io/release</url>
    <snapshots>
      <enabled>>false</enabled>
    </snapshots>
  </repository>
</repositories>
<pluginRepositories>
  <pluginRepository>
    <id>spring-snapshots</id>
    <name>Spring Snapshots</name>
    <url>https://repo.spring.io/snapshot</url>
    <snapshots>
      <enabled>true</enabled>
    </snapshots>
  </pluginRepository>
  <pluginRepository>
    <id>spring-milestones</id>
    <name>Spring Milestones</name>
    <url>https://repo.spring.io/milestone</url>
    <snapshots>
      <enabled>>false</enabled>
    </snapshots>
  </pluginRepository>
  <pluginRepository>
    <id>spring-releases</id>
    <name>Spring Releases</name>
    <url>https://repo.spring.io/release</url>
    <snapshots>
```

```
        <enabled>false</enabled>
    </snapshots>
</pluginRepository>
</pluginRepositories>
```

Gradle

```
repositories {
    mavenCentral()
    mavenLocal()
    maven { url "https://repo.spring.io/snapshot" }
    maven { url "https://repo.spring.io/milestone" }
    maven { url "https://repo.spring.io/release" }
}
```

For simplicity, we use the following acronyms:

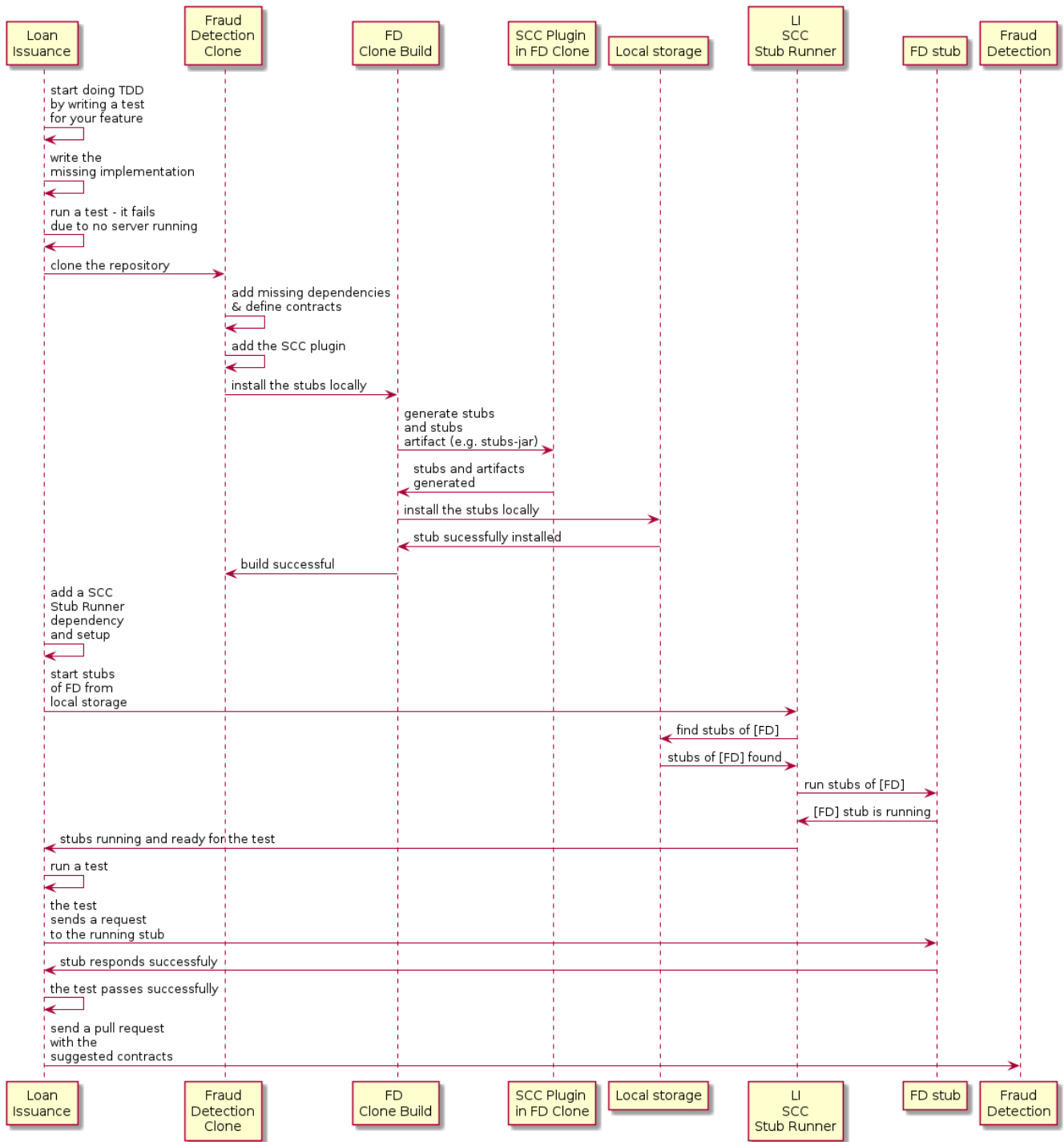
- Loan Issuance (LI): The HTTP client
- Fraud Detection (FD): The HTTP server
- SCC: Spring Cloud Contract

1.4.2. The Consumer Side (Loan Issuance)

As a developer of the Loan Issuance service (a consumer of the Fraud Detection server), you might do the following steps:

1. Start doing TDD by writing a test for your feature.
2. Write the missing implementation.
3. Clone the Fraud Detection service repository locally.
4. Define the contract locally in the repository of the fraud detection service.
5. Add the Spring Cloud Contract (SCC) plugin.
6. Run the integration tests.
7. File a pull request.
8. Create an initial implementation.
9. Take over the pull request.
10. Write the missing implementation.
11. Deploy your application.
12. Work online.

We start with the loan issuance flow, which the following UML diagram shows:



Start Doing TDD by Writing a Test for Your Feature

The following listing shows a test that we might use to check whether a loan amount is too large:

```

@Test
public void shouldBeRejectedDueToAbnormalLoanAmount() {
    // given:
    LoanApplication application = new LoanApplication(new Client("1234567890"),
        99999);
    // when:
    LoanApplicationResult loanApplication = service.loanApplication(application);
    // then:
    assertThat(loanApplication.getLoanApplicationStatus())
        .isEqualTo(LoanApplicationStatus.LOAN_APPLICATION_REJECTED);
    assertThat(loanApplication.getRejectionReason()).isEqualTo("Amount too high");
}

```

Assume that you have written a test of your new feature. If a loan application for a big amount is received, the system should reject that loan application with some description.

Write the Missing Implementation

At some point in time, you need to send a request to the Fraud Detection service. Assume that you need to send the request containing the ID of the client and the amount the client wants to borrow. You want to send it to the `/fraudcheck` URL by using the `PUT` method. To do so, you might use code similar to the following:

```

ResponseEntity<FraudServiceResponse> response = restTemplate.exchange(
    "http://localhost:" + port + fraudCheck(), HttpMethod.PUT,
    new HttpEntity<>(request, httpHeaders), FraudServiceResponse.class);

```

For simplicity, the port of the Fraud Detection service is set to `8080`, and the application runs on `8090`.



If you start the test at this point, it breaks, because no service currently runs on port `8080`.

Clone the Fraud Detection service repository locally

You can start by playing around with the server side contract. To do so, you must first clone it, by running the following command:

```

$ git clone https://your-git-server.com/server-side.git local-http-server-repo

```

Define the Contract Locally in the Repository of the Fraud Detection Service

As a consumer, you need to define what exactly you want to achieve. You need to formulate your expectations. To do so, write the following contract:



Place the contract in the `src/test/resources/contracts/fraud` folder. The `fraud` folder is important because the producer's test base class name references that folder.

The following example shows our contract, in both Groovy and YAML:

groovy

```
/*
 * Copyright 2013-2020 the original author or authors.
 *
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 *     https://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */
```

```
package contracts
```

```
org.springframework.cloud.contract.spec.Contract.make {
    request { // (1)
        method 'PUT' // (2)
        url '/fraudcheck' // (3)
        body([ // (4)
            "client.id": $(regex('[0-9]{10}')),
            loanAmount : 99999
        ])
        headers { // (5)
            contentType('application/json')
        }
    }
    response { // (6)
        status OK() // (7)
        body([ // (8)
            fraudCheckStatus : "FRAUD",
            "rejection.reason": "Amount too high"
        ])
        headers { // (9)
            contentType('application/json')
        }
    }
}
```

```
/*
```

From the Consumer perspective, when shooting a request in the integration test:

- (1) - If the consumer sends a request
- (2) - With the "PUT" method
- (3) - to the URL "/fraudcheck"

- (4) - with the JSON body that
 - * has a field `client.id` that matches a regular expression `[0-9]{10}`
 - * has a field `loanAmount` that is equal to `99999`
- (5) - with header `Content-Type` equal to `application/json`
- (6) - then the response will be sent with
- (7) - status equal `200`
- (8) - and JSON body equal to

```
{ "fraudCheckStatus": "FRAUD", "rejectionReason": "Amount too high" }
```
- (9) - with header `Content-Type` equal to `application/json`

From the Producer perspective, in the autogenerated producer-side test:

- (1) - A request will be sent to the producer
- (2) - With the "PUT" method
- (3) - to the URL "/fraudcheck"
- (4) - with the JSON body that
 - * has a field `client.id` that will have a generated value that matches a regular expression `[0-9]{10}`
 - * has a field `loanAmount` that is equal to `99999`
- (5) - with header `Content-Type` equal to `application/json`
- (6) - then the test will assert if the response has been sent with
- (7) - status equal `200`
- (8) - and JSON body equal to

```
{ "fraudCheckStatus": "FRAUD", "rejectionReason": "Amount too high" }
```
- (9) - with header `Content-Type` matching `application/json.*`
*/

yaml

```
request: # (1)
  method: PUT # (2)
  url: /yamlfraudcheck # (3)
  body: # (4)
    "client.id": 1234567890
    loanAmount: 99999
  headers: # (5)
    Content-Type: application/json
  matchers:
    body:
      - path: $.['client.id'] # (6)
        type: by_regex
        value: "[0-9]{10}"
response: # (7)
  status: 200 # (8)
  body: # (9)
    fraudCheckStatus: "FRAUD"
    "rejection.reason": "Amount too high"
  headers: # (10)
    Content-Type: application/json
```

```

#From the Consumer perspective, when shooting a request in the integration test:
#
#(1) - If the consumer sends a request
#(2) - With the "PUT" method
#(3) - to the URL "/yamlfraudcheck"
#(4) - with the JSON body that
# * has a field `client.id`
# * has a field `loanAmount` that is equal to `99999`
#(5) - with header `Content-Type` equal to `application/json`
#(6) - and a `client.id` json entry matches the regular expression `[0-9]{10}`
#(7) - then the response will be sent with
#(8) - status equal `200`
#(9) - and JSON body equal to
# { "fraudCheckStatus": "FRAUD", "rejectionReason": "Amount too high" }
#(10) - with header `Content-Type` equal to `application/json`
#
#From the Producer perspective, in the autogenerated producer-side test:
#
#(1) - A request will be sent to the producer
#(2) - With the "PUT" method
#(3) - to the URL "/yamlfraudcheck"
#(4) - with the JSON body that
# * has a field `client.id` `1234567890`
# * has a field `loanAmount` that is equal to `99999`
#(5) - with header `Content-Type` equal to `application/json`
#(7) - then the test will assert if the response has been sent with
#(8) - status equal `200`
#(9) - and JSON body equal to
# { "fraudCheckStatus": "FRAUD", "rejectionReason": "Amount too high" }
#(10) - with header `Content-Type` equal to `application/json`

```

The YAML contract is quite straightforward. However, when you take a look at the contract written with a statically typed Groovy DSL, you might wonder what the `value(client(...), server(...))` parts are. By using this notation, Spring Cloud Contract lets you define parts of a JSON block, a URL, or other structure that is dynamic. In the case of an identifier or a timestamp, you need not hardcode a value. You want to allow some different ranges of values. To enable ranges of values, you can set regular expressions that match those values for the consumer side. You can provide the body by means of either a map notation or a String with interpolations. We highly recommend using the map notation.



To set up contracts, you must understand the map notation. See the [Groovy docs regarding JSON](#).

The previously shown contract is an agreement between two sides that:

- If an HTTP request is sent with all of:
 - A `PUT` method on the `/fraudcheck` endpoint
 - A JSON body with a `client.id` that matches the regular expression `[0-9]{10}` and `loanAmount`

equal to 99999

- A **Content-Type** header with a value of `application/vnd.fraud.v1+json`
- Then an HTTP response is sent to the consumer that
 - Has status `200`
 - Contains a JSON body with the `fraudCheckStatus` field containing a value of `FRAUD` and the `rejectionReason` field having a value of `Amount too high`
 - Has a **Content-Type** header with a value of `application/vnd.fraud.v1+json`

Once you are ready to check the API in practice in the integration tests, you need to install the stubs locally.

Add the Spring Cloud Contract Verifier Plugin

We can add either a Maven or a Gradle plugin. In this example, we show how to add Maven. First, we add the **Spring Cloud Contract** BOM, as the following example shows:

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

Next, add the **Spring Cloud Contract Verifier** Maven plugin, as the following example shows:

```
  <plugin>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-contract-maven-plugin</artifactId>
    <version>${spring-cloud-contract.version}</version>
    <extensions>>true</extensions>
    <configuration>

<packageWithBaseClasses>com.example.fraud</packageWithBaseClasses>
<!--           <convertToYaml>true</convertToYaml>-->
    </configuration>
  </plugin>
```

Since the plugin was added, you get the **Spring Cloud Contract Verifier** features, which, from the provided contracts:

- Generate and run tests

- Produce and install stubs

You do not want to generate tests, since you, as the consumer, want only to play with the stubs. You need to skip the test generation and invocation. To do so, run the following commands:

```
$ cd local-http-server-repo
$ ./mvnw clean install -DskipTests
```

Once you run those commands, you should see something like the following content in the logs:

```
[INFO] --- spring-cloud-contract-maven-plugin:1.0.0.BUILD-SNAPSHOT:generateStubs
(default-generateStubs) @ http-server ---
[INFO] Building jar: /some/path/http-server/target/http-server-0.0.1-SNAPSHOT-
stubs.jar
[INFO]
[INFO] --- maven-jar-plugin:2.6:jar (default-jar) @ http-server ---
[INFO] Building jar: /some/path/http-server/target/http-server-0.0.1-SNAPSHOT.jar
[INFO]
[INFO] --- spring-boot-maven-plugin:1.5.5.BUILD-SNAPSHOT:repackage (default) @
http-server ---
[INFO]
[INFO] --- maven-install-plugin:2.5.2:install (default-install) @ http-server ---
[INFO] Installing /some/path/http-server/target/http-server-0.0.1-SNAPSHOT.jar to
/path/to/your/.m2/repository/com/example/http-server/0.0.1-SNAPSHOT/http-server-
0.0.1-SNAPSHOT.jar
[INFO] Installing /some/path/http-server/pom.xml to
/path/to/your/.m2/repository/com/example/http-server/0.0.1-SNAPSHOT/http-server-
0.0.1-SNAPSHOT.pom
[INFO] Installing /some/path/http-server/target/http-server-0.0.1-SNAPSHOT-
stubs.jar to /path/to/your/.m2/repository/com/example/http-server/0.0.1-
SNAPSHOT/http-server-0.0.1-SNAPSHOT-stubs.jar
```

The following line is extremely important:

```
[INFO] Installing /some/path/http-server/target/http-server-0.0.1-SNAPSHOT-
stubs.jar to /path/to/your/.m2/repository/com/example/http-server/0.0.1-
SNAPSHOT/http-server-0.0.1-SNAPSHOT-stubs.jar
```

It confirms that the stubs of the `http-server` have been installed in the local repository.

Running the Integration Tests

In order to profit from the Spring Cloud Contract Stub Runner functionality of automatic stub downloading, you must do the following in your consumer side project (**Loan Application service**):

1. Add the **Spring Cloud Contract** BOM, as follows:

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

2. Add the dependency to **Spring Cloud Contract Stub Runner**, as follows:

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-contract-stub-runner</artifactId>
  <scope>test</scope>
</dependency>
```

3. Annotate your test class with **@AutoConfigureStubRunner**. In the annotation, provide the **group-id** and **artifact-id** for the Stub Runner to download the stubs of your collaborators.

```
@SpringBootTest(webEnvironment = WebEnvironment.NONE)
@AutoConfigureStubRunner(ids = {
    "com.example:http-server-dsl:0.0.1:stubs"}, stubsMode =
StubRunnerProperties.StubsMode.LOCAL)
public class LoanApplicationServiceTests {
```

4. (Optional) Because you are playing with the collaborators offline, you can also provide the offline work switch (**StubRunnerProperties.StubsMode.LOCAL**).

Now, when you run your tests, you see something like the following output in the logs:

```

2016-07-19 14:22:25.403 INFO 41050 --- [           main]
o.s.c.c.stubrunner.AetherStubDownloader : Desired version is + - will try to
resolve the latest version
2016-07-19 14:22:25.438 INFO 41050 --- [           main]
o.s.c.c.stubrunner.AetherStubDownloader : Resolved version is 0.0.1-SNAPSHOT
2016-07-19 14:22:25.439 INFO 41050 --- [           main]
o.s.c.c.stubrunner.AetherStubDownloader : Resolving artifact com.example:http-
server:jar:stubs:0.0.1-SNAPSHOT using remote repositories []
2016-07-19 14:22:25.451 INFO 41050 --- [           main]
o.s.c.c.stubrunner.AetherStubDownloader : Resolved artifact com.example:http-
server:jar:stubs:0.0.1-SNAPSHOT to /path/to/your/.m2/repository/com/example/http-
server/0.0.1-SNAPSHOT/http-server-0.0.1-SNAPSHOT-stubs.jar
2016-07-19 14:22:25.465 INFO 41050 --- [           main]
o.s.c.c.stubrunner.AetherStubDownloader : Unpacking stub from JAR [URI:
file:/path/to/your/.m2/repository/com/example/http-server/0.0.1-SNAPSHOT/http-
server-0.0.1-SNAPSHOT-stubs.jar]
2016-07-19 14:22:25.475 INFO 41050 --- [           main]
o.s.c.c.stubrunner.AetherStubDownloader : Unpacked file to
[/var/folders/0p/xwq47sq106x1_g3dtv6qfm940000gq/T/contracts100276532569594265]
2016-07-19 14:22:27.737 INFO 41050 --- [           main]
o.s.c.c.stubrunner.StubRunnerExecutor   : All stubs are now running RunningStubs
[NamesAndPorts={com.example:http-server:0.0.1-SNAPSHOT:stubs=8080}]

```

This output means that Stub Runner has found your stubs and started a server for your application with a group ID of `com.example` and an artifact ID of `http-server` with version `0.0.1-SNAPSHOT` of the stubs and with the `stubs` classifier on port `8080`.

Filing a Pull Request

What you have done until now is an iterative process. You can play around with the contract, install it locally, and work on the consumer side until the contract works as you wish.

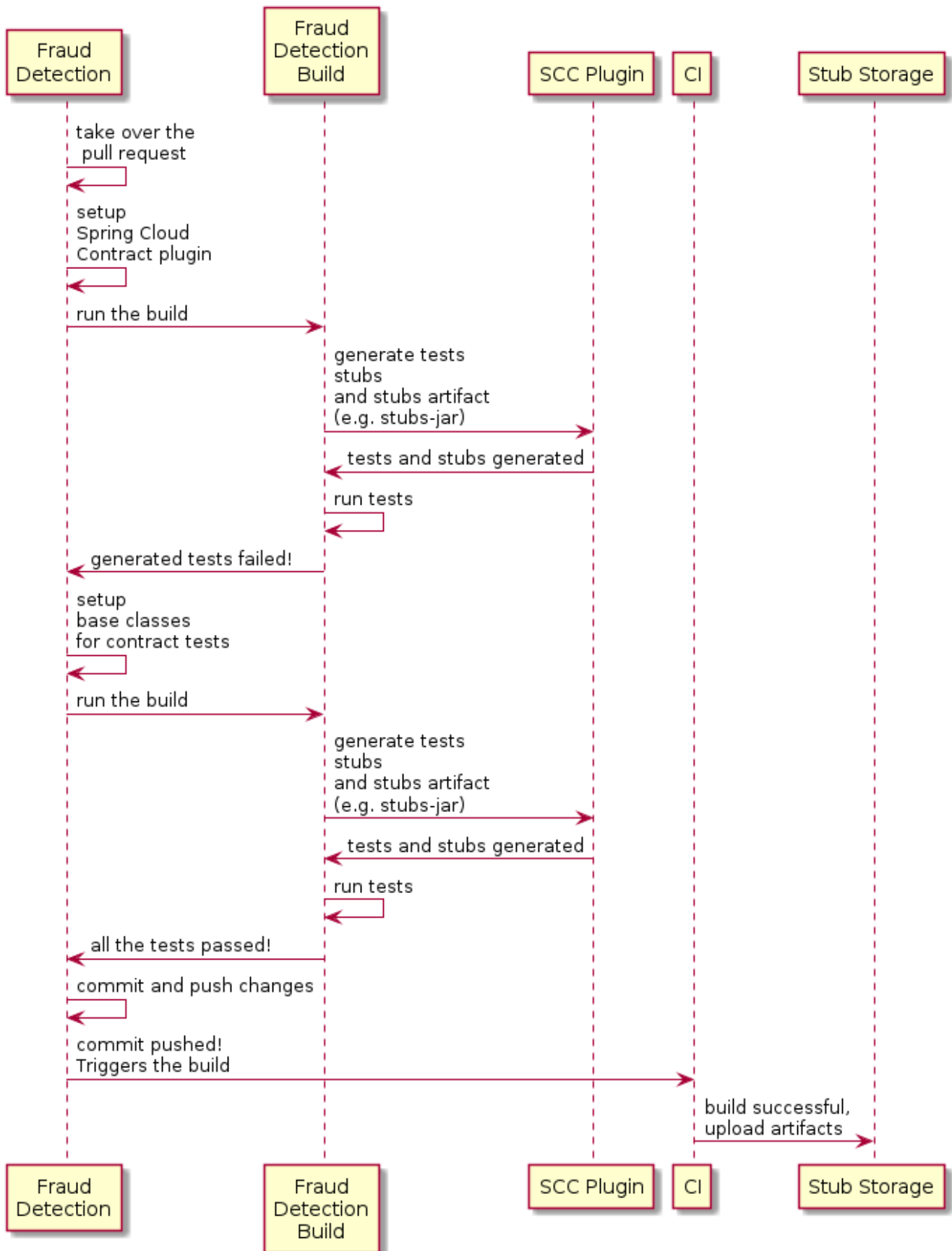
Once you are satisfied with the results and the test passes, you can publish a pull request to the server side. Currently, the consumer side work is done.

1.4.3. The Producer Side (Fraud Detection server)

As a developer of the Fraud Detection server (a server to the Loan Issuance service), you might want to:

- Take over the pull request
- Write the missing implementation
- Deploy the application

The following UML diagram shows the fraud detection flow:



Taking over the Pull Request

As a reminder, the following listing shows the initial implementation:

```
@RequestMapping(value = "/fraudcheck", method = PUT)
public FraudCheckResult fraudCheck(@RequestBody FraudCheck fraudCheck) {
    return new FraudCheckResult(FraudCheckStatus.OK, NO_REASON);
}
```

Then you can run the following commands:

```
$ git checkout -b contract-change-pr master
$ git pull https://your-git-server.com/server-side-fork.git contract-change-pr
```

You must add the dependencies needed by the autogenerated tests, as follows:

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-contract-verifier</artifactId>
  <scope>test</scope>
</dependency>
```

In the configuration of the Maven plugin, you must pass the `packageWithBaseClasses` property, as follows:

```
    <plugin>
      <groupId>org.springframework.cloud</groupId>
      <artifactId>spring-cloud-contract-maven-plugin</artifactId>
      <version>${spring-cloud-contract.version}</version>
      <extensions>true</extensions>
      <configuration>

<packageWithBaseClasses>com.example.fraud</packageWithBaseClasses>
<!--           <convertToYaml>true</convertToYaml>-->
      </configuration>
    </plugin>
```



This example uses “convention-based” naming by setting the `packageWithBaseClasses` property. Doing so means that the two last packages combine to make the name of the base test class. In our case, the contracts were placed under `src/test/resources/contracts/fraud`. Since you do not have two packages starting from the `contracts` folder, pick only one, which should be `fraud`. Add the `Base` suffix and capitalize `fraud`. That gives you the `FraudBase` test class name.

All the generated tests extend that class. Over there, you can set up your Spring Context or whatever is necessary. In this case, you should use [Rest Assured MVC](#) to start the server side `FraudDetectionController`. The following listing shows the `FraudBase` class:

```

/*
 * Copyright 2013-2020 the original author or authors.
 *
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 *     https://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */

package com.example.fraud;

import io.restassured.module.mockmvc.RestAssuredMockMvc;
import org.junit.jupiter.api.BeforeEach;

public class FraudBase {

    @BeforeEach
    public void setup() {
        RestAssuredMockMvc.standaloneSetup(new FraudDetectionController(),
            new FraudStatsController(stubbedStatsProvider()));
    }

    private StatsProvider stubbedStatsProvider() {
        return fraudType -> {
            switch (fraudType) {
                case DRUNKS:
                    return 100;
                case ALL:
                    return 200;
            }
            return 0;
        };
    }

    public void assertThatRejectionReasonIsNull(Object rejectionReason) {
        assert rejectionReason == null;
    }

}

```

Now, if you run the `./mvnw clean install`, you get something like the following output:

Results :

Tests in error:

ContractVerifierTest.validate_shouldMarkClientAsFraud:32 » IllegalState
Parsed...

This error occurs because you have a new contract from which a test was generated, and it failed since you have not implemented the feature. The auto-generated test would look like the following test method:

```
@Test
public void validate_shouldMarkClientAsFraud() throws Exception {
    // given:
    MockMvcRequestSpecification request = given()
        .header("Content-Type", "application/vnd.fraud.v1+json")
        .body("{\"client.id\":\"1234567890\",\"loanAmount\":99999}");

    // when:
    ResponseOptions response = given().spec(request)
        .put("/fraudcheck");

    // then:
    assertThat(response.statusCode()).isEqualTo(200);
    assertThat(response.header("Content-
Type")).matches("application/vnd.fraud.v1.json.*");
    // and:
    DocumentContext parsedJson =
    JsonPath.parse(response.getBody().asString());
    assertThatJson(parsedJson).field("[ 'fraudCheckStatus' ]").matches("[A-
Z]{5}");
    assertThatJson(parsedJson).field("[ 'rejection.reason' ]").isEqualTo("Amount
too high");
}
```

If you used the Groovy DSL, you can see that all the `producer()` parts of the Contract that were present in the `value(consumer(...), producer(...))` blocks got injected into the test. If you use YAML, the same applies for the `matchers` sections of the `response`.

Note that, on the producer side, you are also doing TDD. The expectations are expressed in the form of a test. This test sends a request to our own application with the URL, headers, and body defined in the contract. It also expects precisely defined values in the response. In other words, you have the `red` part of `red, green, and refactor`. It is time to convert the `red` into the `green`.

Write the Missing Implementation

Because you know the expected input and expected output, you can write the missing implementation as follows:

```
@RequestMapping(value = "/fraudcheck", method = PUT)
public FraudCheckResult fraudCheck(@RequestBody FraudCheck fraudCheck) {
    if (amountGreaterThanThreshold(fraudCheck)) {
        return new FraudCheckResult(FraudCheckStatus.FRAUD, AMOUNT_TOO_HIGH);
    }
    return new FraudCheckResult(FraudCheckStatus.OK, NO_REASON);
}
```

When you run `./mvnw clean install` again, the tests pass. Since the Spring Cloud Contract Verifier plugin adds the tests to the `generated-test-sources`, you can actually run those tests from your IDE.

Deploying Your Application

Once you finish your work, you can deploy your changes. To do so, you must first merge the branch by running the following commands:

```
$ git checkout master
$ git merge --no-ff contract-change-pr
$ git push origin master
```

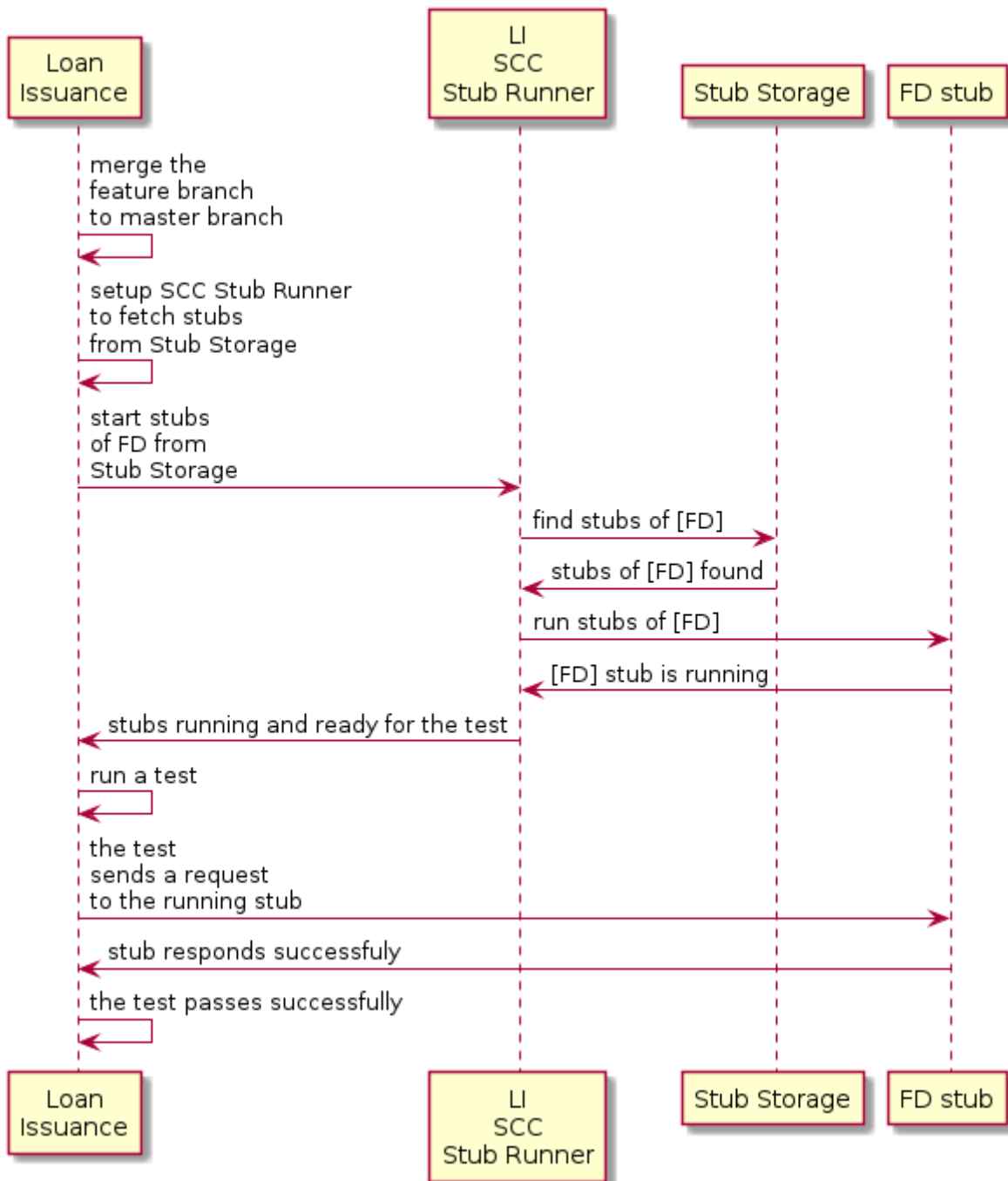
Your CI might run a command such as `./mvnw clean deploy`, which would publish both the application and the stub artifacts.

1.4.4. Consumer Side (Loan Issuance), Final Step

As a developer of the loan issuance service (a consumer of the Fraud Detection server), you need to:

- Merge our feature branch to `master`
- Switch to online mode of working

The following UML diagram shows the final state of the process:



Merging a Branch to Master

The following commands show one way to merge a branch into master with Git:

```
$ git checkout master
$ git merge --no-ff contract-change-pr
```

Working Online

Now you can disable the offline work for Spring Cloud Contract Stub Runner and indicate where the repository with your stubs is located. At this moment, the stubs of the server side are automatically downloaded from Nexus/Artifactory. You can set the value of `stubsMode` to `REMOTE`. The

following code shows an example of achieving the same thing by changing the properties:

```
stubrunner:  
  ids: 'com.example:http-server-dsl+:stubs:8080'  
  repositoryRoot: https://repo.spring.io/libs-snapshot
```

That's it. You have finished the tutorial.

1.5. Next Steps

Hopefully, this section provided some of the Spring Cloud Contract basics and got you on your way to writing your own applications. If you are a task-oriented type of developer, you might want to jump over to spring.io and check out some [getting started](#) guides that solve specific “How do I do that with Spring?” problems. We also have Spring Cloud Contract-specific “[how-to](#)” reference documentation.

Otherwise, the next logical step is to read [Using Spring Cloud Contract](#). If you are really impatient, you could also jump ahead and read about [Spring Cloud Contract features](#).

In addition, you can check out the following videos:

- "Consumer Driven Contracts and Your Microservice Architecture" by Olga Maciaszek-Sharma and Marcin Grzejszczak

The image shows a presentation slide titled "Who is who?" under a "Demo" header. It features two server icons. The left server is labeled "CONSUMER" and has "BLACK TERMINAL" and "BLACK IDE" listed below it. The right server is labeled "PRODUCER" and has "WHITE TERMINAL" and "WHITE IDE" listed below it. The slide is part of a presentation at BITCONF (Bydgoszcz IT Conference) on 14.09.2018, presented by Marcin Grzejszczak and Olga Maciaszek-Sharma. The slide number is 28 and the Spring logo is in the bottom right corner. At the bottom of the slide, it says "CONSUMER DRIVEN CONTRACTS LIKE TDD TO THE API".

- "Contract Tests in the Enterprise" by Marcin Grzejszczak

Generating Stubs From Proxy

```
graph LR; A[Test that calls Customer Rental History Service] --> B[PROXY]; B --> C[Customer Rental History Service]; C --> D[Payment processor]; C --> E[Mainframe]; B --> F[Record traffic and dump stubs (e.g. once per day)]; F --> G[Upload stubs for other teams to use];
```

SpringOne Platform by Pivotal

DEVOXX™
POLAND

- "Why Contract Tests Matter?" by Marcin Grzejszczak

IT talk ^{LUB} + LJUG

You can find the default project samples at [samples](#).

2. Using Spring Cloud Contract

This section goes into more detail about how you should use Spring Cloud Contract. It covers topics such as flows of how to work with Spring Cloud Contract. We also cover some Spring Cloud Contract best practices.

If you are starting out with Spring Cloud Contract, you should probably read the [Getting Started](#) guide before diving into this section.

2.1. Provider Contract Testing with Stubs in Nexus or Artifactory

You can check the [Developing Your First Spring Cloud Contract based application](#) link to see the provider contract testing with stubs in the Nexus or Artifactory flow.

2.2. Provider Contract Testing with Stubs in Git

In this flow, we perform the provider contract testing (the producer has no knowledge of how consumers use their API). The stubs are uploaded to a separate repository (they are not uploaded to Artifactory or Nexus).

2.2.1. Prerequisites

Before testing provider contracts with stubs in git, you must provide a git repository that contains all the stubs for each producer. For an example of such a project, see [this samples](#) or [this sample](#). As a result of pushing stubs there, the repository has the following structure:

```
$ tree .
├── META-INF
│   └── folder.with.group.id.as.its.name
│       ├── folder-with-artifact-id
│       │   └── folder-with-version
│       │       ├── contractA.groovy
│       │       ├── contractB.yml
│       │       └── contractC.groovy
```

You must also provide consumer code that has Spring Cloud Contract Stub Runner set up. For an example of such a project, see [this sample](#) and search for a `BeerControllerGitTest` test. You must also provide producer code that has Spring Cloud Contract set up, together with a plugin. For an example of such a project, see [this sample](#).

2.2.2. The Flow

The flow looks exactly as the one presented in [Developing Your First Spring Cloud Contract based application](#), but the `Stub Storage` implementation is a git repository.

You can read more about setting up a git repository and setting consumer and producer side in the [How To page](#) of the documentation.

2.2.3. Consumer setup

In order to fetch the stubs from a git repository instead of Nexus or Artifactory, you need to use the `git` protocol in the URL of the `repositoryRoot` property in Stub Runner. The following example shows how to set it up:

Annotation

```
@AutoConfigureStubRunner(  
    stubsMode = StubRunnerProperties.StubsMode.REMOTE,  
    repositoryRoot = "git://git@github.com:spring-cloud-samples/spring-cloud-  
contract-nodejs-contracts-git.git",  
    ids = "com.example:artifact-id:0.0.1")
```

JUnit 4 Rule

```
@Rule  
    public StubRunnerRule rule = new StubRunnerRule()  
        .downloadStub("com.example","artifact-id", "0.0.1")  
        .repoRoot("git://git@github.com:spring-cloud-samples/spring-cloud-  
contract-nodejs-contracts-git.git")  
        .stubsMode(StubRunnerProperties.StubsMode.REMOTE);
```

JUnit 5 Extension

```
@RegisterExtension  
    public StubRunnerExtension stubRunnerExtension = new StubRunnerExtension()  
        .downloadStub("com.example","artifact-id", "0.0.1")  
        .repoRoot("git://git@github.com:spring-cloud-samples/spring-cloud-  
contract-nodejs-contracts-git.git")  
        .stubsMode(StubRunnerProperties.StubsMode.REMOTE);
```

2.2.4. Setting up the Producer

To push the stubs to a git repository instead of Nexus or Artifactory, you need to use the **git** protocol in the URL of the plugin setup. Also you need to explicitly tell the plugin to push the stubs at the end of the build process. The following examples show how to do so in both Maven and Gradle:

Maven

```
<plugin>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-contract-maven-plugin</artifactId>
  <version>${spring-cloud-contract.version}</version>
  <extensions>>true</extensions>
  <configuration>
    <!-- Base class mappings etc. -->

    <!-- We want to pick contracts from a Git repository -->
    <contractsRepositoryUrl>git://git://git@github.com:spring-cloud-
samples/spring-cloud-contract-nodejs-contracts-git.git</contractsRepositoryUrl>

    <!-- We reuse the contract dependency section to set up the path
to the folder that contains the contract definitions. In our case the
path will be /groupId/artifactId/version/contracts -->
    <contractDependency>
      <groupId>${project.groupId}</groupId>
      <artifactId>${project.artifactId}</artifactId>
      <version>${project.version}</version>
    </contractDependency>

    <!-- The contracts mode can't be classpath -->
    <contractsMode>REMOTE</contractsMode>
  </configuration>
  <executions>
    <execution>
      <phase>package</phase>
      <goals>
        <!-- By default we will not push the stubs back to SCM,
you have to explicitly add it as a goal -->
        <goal>pushStubsToScm</goal>
      </goals>
    </execution>
  </executions>
</plugin>
```

Gradle

```
contracts {
    // We want to pick contracts from a Git repository
    contractDependency {
        stringNotation = "${project.group}:${project.name}:${project.version}"
    }
    /*
    We reuse the contract dependency section to set up the path
    to the folder that contains the contract definitions. In our case the
    path will be /groupId/artifactId/version/contracts
    */
    contractRepository {
        repositoryUrl = "git://git://git@github.com:spring-cloud-samples/spring-
cloud-contract-nodejs-contracts-git.git"
    }
    // The mode can't be classpath
    contractsMode = "REMOTE"
    // Base class mappings etc.
}

/*
In this scenario we want to publish stubs to SCM whenever
the `publish` task is run
*/
publish.dependsOn("publishStubsToScm")
```

You can read more about setting up a git repository in the [How To section](#) of the documentation.

2.3. Consumer Driven Contracts with Contracts on the Producer Side

See [Step-by-step Guide to Consumer Driven Contracts \(CDC\) with Contracts on the Producer Side](#) to see the Consumer Driven Contracts with contracts on the producer side flow.

2.4. Consumer Driven Contracts with Contracts in an External Repository

In this flow, we perform Consumer Driven Contract testing. The contract definitions are stored in a separate repository.

2.4.1. Prerequisites

To use consumer-driven contracts with the contracts held in an external repository, you need to set up a git repository that:

- Contains all the contract definitions for each producer.

- Can package the contract definitions in a JAR.
- For each contract producer, contains a way (for example, `pom.xml`) to install stubs locally through the Spring Cloud Contract Plugin (SCC Plugin).

For more information, see the [How To section](#), where we describe how to set up such a repository. For an example of such a project, see [this sample](#).

You also need consumer code that has Spring Cloud Contract Stub Runner set up. For an example of such a project, see [this sample](#). You also need producer code that has Spring Cloud Contract set up, together with a plugin. For an example of such a project, see [this sample](#). The stub storage is Nexus or Artifactory.

At a high level, the flow is as follows:

1. The consumer works with the contract definitions from the separate repository.
2. Once the consumer's work is done, a branch with working code is created on the consumer side, and a pull request is made to the separate repository that holds the contract definitions.
3. The producer takes over the pull request to the separate repository with contract definitions and installs the JAR with all contracts locally.
4. The producer generates tests from the locally stored JAR and writes the missing implementation to make the tests pass.
5. Once the producer's work is done, the pull request to the repository that holds the contract definitions is merged.
6. After the CI tool builds the repository with the contract definitions and the JAR with contract definitions gets uploaded to Nexus or Artifactory, the producer can merge its branch.
7. Finally, the consumer can switch to working online to fetch stubs of the producer from a remote location, and the branch can be merged to master.

2.4.2. Consumer Flow

The consumer:

1. Writes a test that would send a request to the producer.

The test fails due to no server being present.

2. Clones the repository that holds the contract definitions.
3. Sets up the requirements as contracts under the folder, with the consumer name as a subfolder of the producer.

For example, for a producer named `producer` and a consumer named `consumer`, the contracts would be stored under `src/main/resources/contracts/producer/consumer/`

4. Once the contracts are defined, installs the producer stubs to local storage, as the following example shows:


```

$ cd src/main/resource/contracts/producer
$ ./mvnw clean install

```

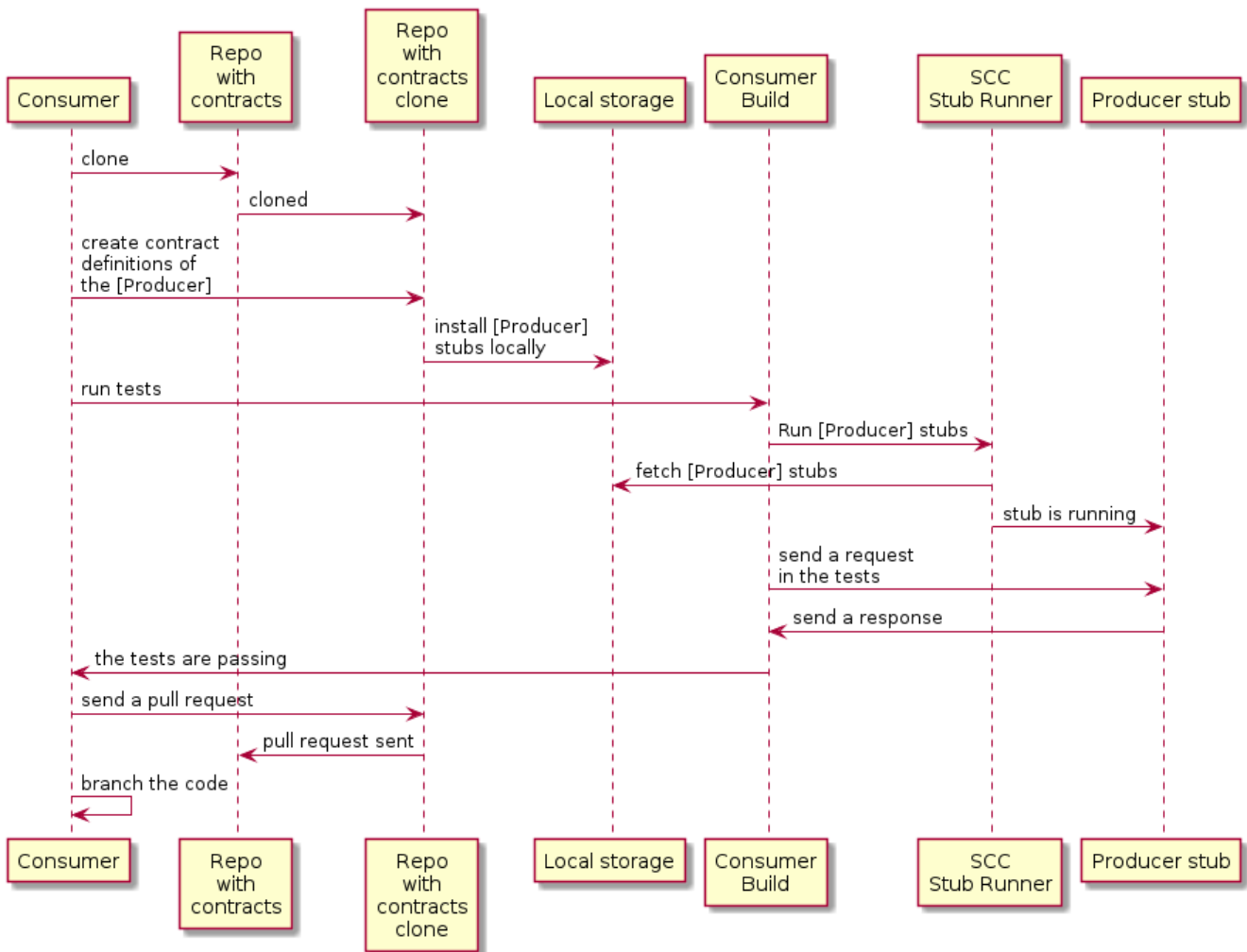
5. Sets up Spring Cloud Contract (SCC) Stub Runner in the consumer tests, to:

- Fetch the producer stubs from local storage.
- Work in the stubs-per-consumer mode (this enables consumer driven contracts mode).

The SCC Stub Runner:

- Fetches the producer stubs.
- Runs an in-memory HTTP server stub with the producer stubs. Now your test communicates with the HTTP server stub, and your tests pass.
- Creates a pull request to the repository with contract definitions, with the new contracts for the producer.
- Branches your consumer code, until the producer team has merged their code.

The following UML diagram shows the consumer flow:



2.4.3. Producer Flow

The producer:

1. Takes over the pull request to the repository with contract definitions. You can do it from the command line, as follows

```
$ git checkout -b the_branch_with_pull_request master
git pull https://github.com/user_id/project_name.git
the_branch_with_pull_request
```

2. Installs the contract definitions, as follows

```
$ ./mvnw clean install
```

3. Sets up the plugin to fetch the contract definitions from a JAR instead of from `src/test/resources/contracts`, as follows:

Maven

```
<plugin>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-contract-maven-plugin</artifactId>
  <version>${spring-cloud-contract.version}</version>
  <extensions>true</extensions>
  <configuration>
    <!-- We want to use the JAR with contracts with the following
coordinates -->
    <contractDependency>
      <groupId>com.example</groupId>
      <artifactId>beer-contracts</artifactId>
    </contractDependency>
    <!-- The JAR with contracts should be taken from Maven local -->
    <contractsMode>LOCAL</contractsMode>
    <!-- ... additional configuration -->
  </configuration>
</plugin>
```

Gradle

```
contracts {
  // We want to use the JAR with contracts with the following coordinates
  // group id `com.example`, artifact id `beer-contracts`, LATEST version and
  NO classifier
  contractDependency {
    stringNotation = 'com.example:beer-contracts:::'
  }
  // The JAR with contracts should be taken from Maven local
  contractsMode = "LOCAL"
  // Additional configuration
}
```

4. Runs the build to generate tests and stubs, as follows:

Maven

```
./mvnw clean install
```

Gradle

```
./gradlew clean build
```

5. Writes the missing implementation, to make the tests pass.

6. Merges the pull request to the repository with contract definitions, as follows:

```
$ git commit -am "Finished the implementation to make the contract tests pass"
$ git checkout master
$ git merge --no-ff the_branch_with_pull_request
$ git push origin master
```

The CI system builds the project with the contract definitions and uploads the JAR with the contract definitions to Nexus or Artifactory.

7. Switches to working remotely.

8. Sets up the plugin so that the contract definitions are no longer taken from the local storage but from a remote location, as follows:

Maven

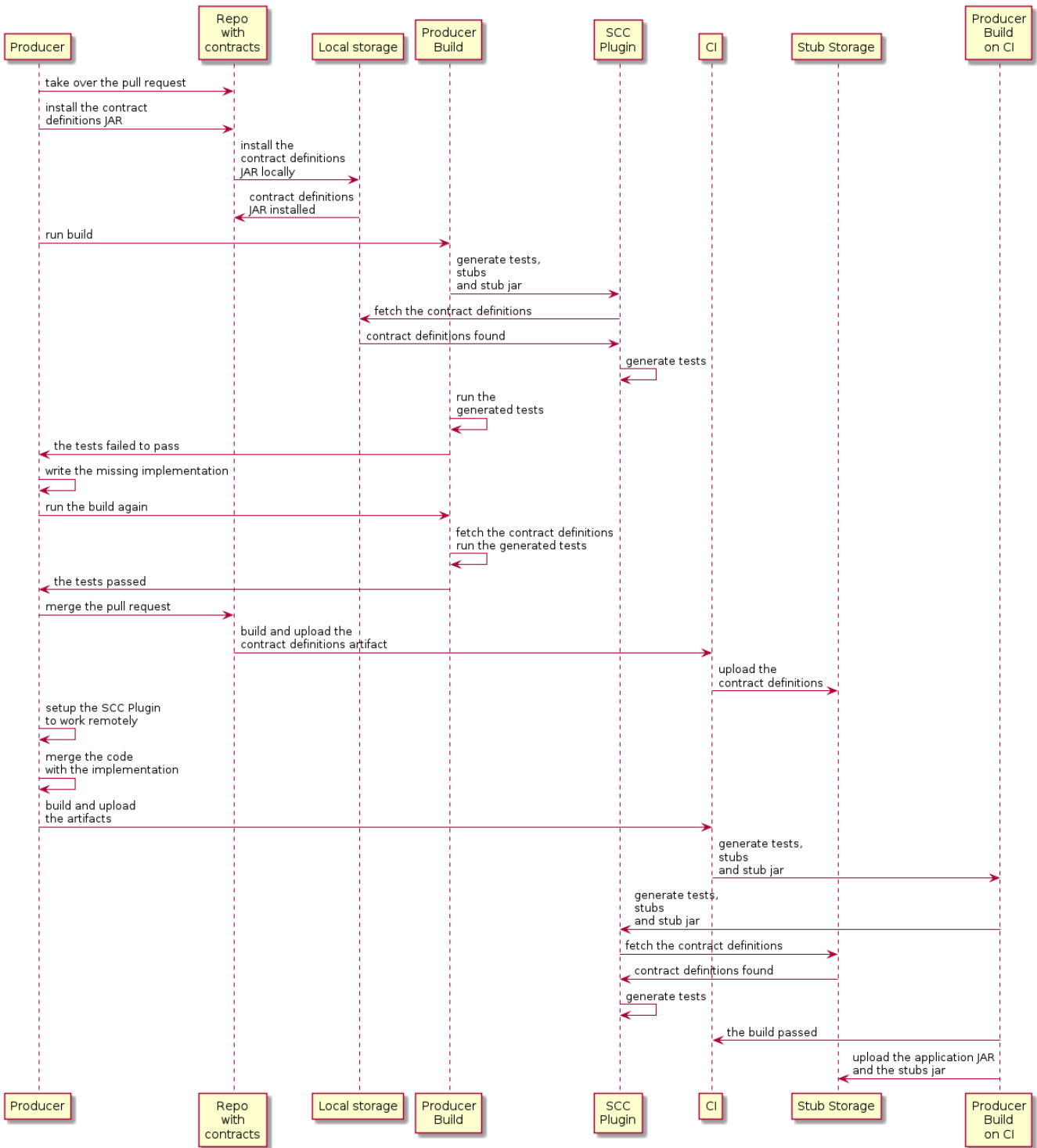
```
<plugin>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-contract-maven-plugin</artifactId>
  <version>${spring-cloud-contract.version}</version>
  <extensions>true</extensions>
  <configuration>
    <!-- We want to use the JAR with contracts with the following
coordinates -->
    <contractDependency>
      <groupId>com.example</groupId>
      <artifactId>beer-contracts</artifactId>
    </contractDependency>
    <!-- The JAR with contracts should be taken from a remote location -->
    <contractsMode>REMOTE</contractsMode>
    <!-- ... additional configuration -->
  </configuration>
</plugin>
```

Gradle

```
contracts {
  // We want to use the JAR with contracts with the following coordinates
  // group id `com.example`, artifact id `beer-contracts`, LATEST version and
  NO classifier
  contractDependency {
    stringNotation = 'com.example:beer-contracts:::'
  }
  // The JAR with contracts should be taken from a remote location
  contractsMode = "REMOTE"
  // Additional configuration
}
```

9. Merges the producer code with the new implementation.
10. The CI system:
 - Builds the project.
 - Generates tests, stubs, and the stub JAR.
 - Uploads the artifact with the application and the stubs to Nexus or Artifactory.

The following UML diagram shows the producer process:



2.5. Consumer-driven Contracts with Contracts on the Producer Side, Pushed to Git

You can read the [Step-by-step Guide to Consumer Driven Contracts \(CDC\) with contracts laying on the producer side](#) to see the consumer driven contracts with contracts on the producer side flow.

The stub storage implementation is a git repository. We describe its setup in the [Provider Contract Testing with Stubs in Git](#) section.

You can read more about setting up a git repository for the consumer and producer sides in the [How To](#) section of the documentation.

2.6. Provider Contract Testing with Stubs in Artifactory for a non-Spring Application

2.6.1. The Flow

You can read [Developing Your First Spring Cloud Contract-based Application](#) to see the flow for provider contract testing with stubs in Nexus or Artifactory.

2.6.2. Setting up the Consumer

For the consumer side, you can use a JUnit rule. That way, you need not start a Spring context. The following listing shows such a rule (in JUnit4 and JUnit 5);

JUnit 4 Rule

```
@Rule
public StubRunnerRule rule = new StubRunnerRule()
    .downloadStub("com.example","artifact-id", "0.0.1")
    .repoRoot("git://git@github.com:spring-cloud-samples/spring-cloud-
contract-nodejs-contracts-git.git")
    .stubsMode(StubRunnerProperties.StubsMode.REMOTE);
```

JUnit 5 Extension

```
@RegisterExtension
public StubRunnerExtension stubRunnerExtension = new StubRunnerExtension()
    .downloadStub("com.example","artifact-id", "0.0.1")
    .repoRoot("git://git@github.com:spring-cloud-samples/spring-cloud-
contract-nodejs-contracts-git.git")
    .stubsMode(StubRunnerProperties.StubsMode.REMOTE);
```

2.6.3. Setting up the Producer

By default, the Spring Cloud Contract Plugin uses Rest Assured's `MockMvc` setup for the generated tests. Since non-Spring applications do not use `MockMvc`, you can change the `testMode` to `EXPLICIT` to send a real request to an application bound at a specific port.

In this example, we use a framework called [Javalin](#) to start a non-Spring HTTP server.

Assume that we have the following application:

```
package com.example.demo;

import io.javalin.Javalin;

public class DemoApplication {

    public static void main(String[] args) {
        new DemoApplication().run(7000);
    }

    public Javalin start(int port) {
        return Javalin.create().start(port);
    }

    public Javalin registerGet(Javalin app) {
        return app.get("/", ctx -> ctx.result("Hello World"));
    }

    public Javalin run(int port) {
        return registerGet(start(port));
    }
}
```

Given that application, we can set up the plugin to use the **EXPLICIT** mode (that is, to send out requests to a real port), as follows:

Maven

```
<plugin>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-contract-maven-plugin</artifactId>
  <version>${spring-cloud-contract.version}</version>
  <extensions>>true</extensions>
  <configuration>
    <baseClassForTests>com.example.demo.BaseClass</baseClassForTests>
    <!-- This will setup the EXPLICIT mode for the tests -->
    <testMode>EXPLICIT</testMode>
  </configuration>
</plugin>
```

Gradle

```
contracts {
  // This will setup the EXPLICIT mode for the tests
  testMode = "EXPLICIT"
  baseClassForTests = "com.example.demo.BaseClass"
}
```

The base class might resemble the following:

```

import io.javalin.Javalin;
import io.restassured.RestAssured;
import org.junit.After;
import org.junit.Before;
import org.springframework.cloud.test.TestSocketUtils;

public class BaseClass {

    Javalin app;

    @Before
    public void setup() {
        // pick a random port
        int port = TestSocketUtils.findAvailableTcpPort();
        // start the application at a random port
        this.app = start(port);
        // tell Rest Assured where the started application is
        RestAssured.baseURI = "http://localhost:" + port;
    }

    @After
    public void close() {
        // stop the server after each test
        this.app.stop();
    }

    private Javalin start(int port) {
        // reuse the production logic to start a server
        return new DemoApplication().run(port);
    }
}

```

With such a setup:

- We have set up the Spring Cloud Contract plugin to use the **EXPLICIT** mode to send real requests instead of mocked ones.
- We have defined a base class that:
 - Starts the HTTP server on a random port for each test.
 - Sets Rest Assured to send requests to that port.
 - Closes the HTTP server after each test.

2.7. Provider Contract Testing with Stubs in Artifactory in a Non-JVM World

In this flow, we assume that:

- The API Producer and API Consumer are non-JVM applications.
- The contract definitions are written in YAML.
- The Stub Storage is Artifactory or Nexus.
- Spring Cloud Contract Docker (SCC Docker) and Spring Cloud Contract Stub Runner Docker (SCC Stub Runner Docker) images are used.

You can read more about how to use Spring Cloud Contract with Docker [here](#).

[Here](#), you can read a blog post about how to use Spring Cloud Contract in a polyglot world.

[Here](#), you can find a sample of a NodeJS application that uses Spring Cloud Contract both as a producer and a consumer.

2.7.1. Producer Flow

At a high level, the producer:

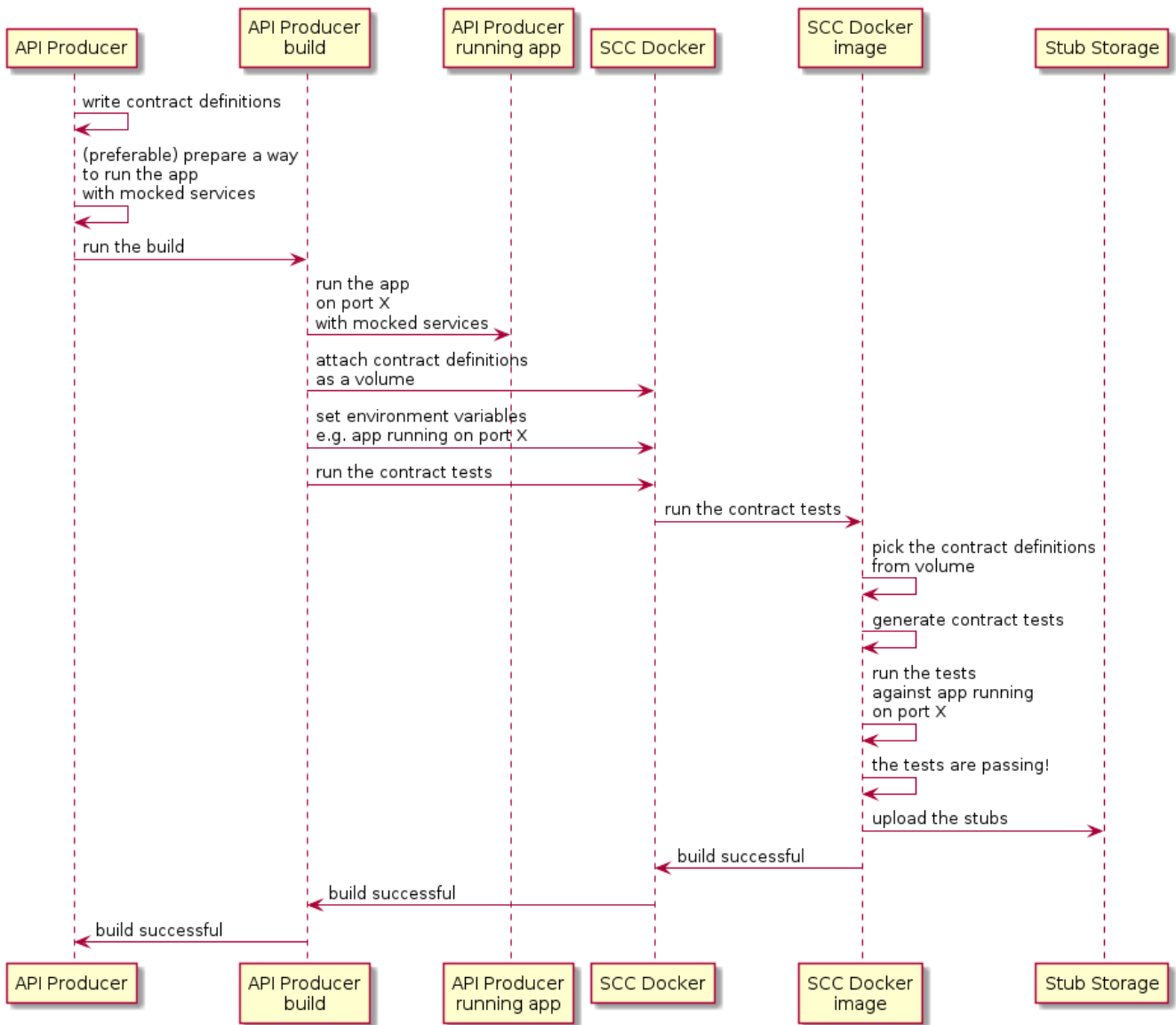
1. Writes contract definitions (for example, in YAML).
2. Sets up the build tool to:
 - a. Start the application with mocked services on a given port.

If mocking is not possible, you can set up the infrastructure and define tests in a stateful way.
 - b. Run the Spring Cloud Contract Docker image and pass the port of a running application as an environment variable.

The SCC Docker image: * Generates the tests from the attached volume. * Runs the tests against the running application.

Upon test completion, stubs get uploaded to a stub storage site (such as Artifactory or Git).

The following UML diagram shows the producer flow:



2.7.2. Consumer Flow

At a high level, the consumer:

1. Sets up the build tool to:

- Start the Spring Cloud Contract Stub Runner Docker image and start the stubs.

The environment variables configure:

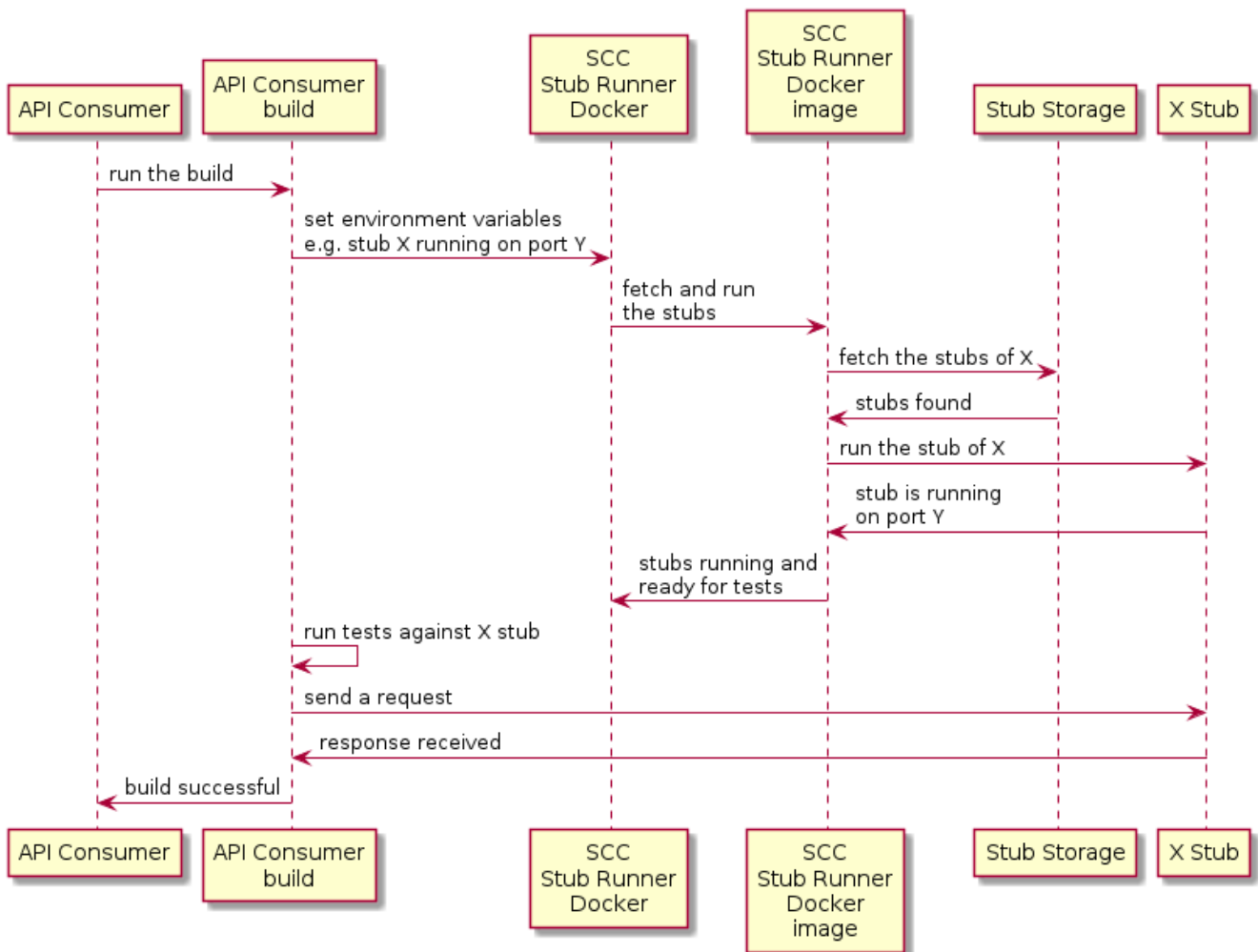
- The stubs to fetch.
- The location of the repositories.

Note that:

- To use the local storage, you can also attach it as a volume.
- The ports at which the stubs are running need to be exposed.

2. Run the application tests against the running stubs.

The following UML diagram shows the consumer flow:



2.8. Provider Contract Testing with REST Docs and Stubs in Nexus or Artifactory

In this flow, we do not use a Spring Cloud Contract plugin to generate tests and stubs. We write [Spring RESTDocs](#), and, from them, we automatically generate stubs. Finally, we set up our builds to package the stubs and upload them to the stub storage site — in our case, Nexus or Artifactory.

2.8.1. Producer Flow

As a producer, we:

1. Write RESTDocs tests of our API.
2. Add Spring Cloud Contract Stub Runner starter to our build (`spring-cloud-starter-contract-stub-runner`), as follows:

Maven

```
<dependencies>
  <dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-contract-stub-runner</artifactId>
    <scope>test</scope>
  </dependency>
</dependencies>

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

Gradle

```
dependencies {
    testImplementation 'org.springframework.cloud:spring-cloud-starter-
contract-stub-runner'
}

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

3. We set up the build tool to package our stubs, as follows:

Maven

```
<!-- pom.xml -->
<plugins>
  <plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-assembly-plugin</artifactId>
    <executions>
      <execution>
        <id>stub</id>
        <phase>prepare-package</phase>
        <goals>
          <goal>single</goal>
        </goals>
        <inherited>>false</inherited>
        <configuration>
          <attach>>true</attach>
          <descriptors>
            ${basedir}/src/assembly/stub.xml
          </descriptors>
        </configuration>
      </execution>
    </executions>
  </plugin>
</plugins>

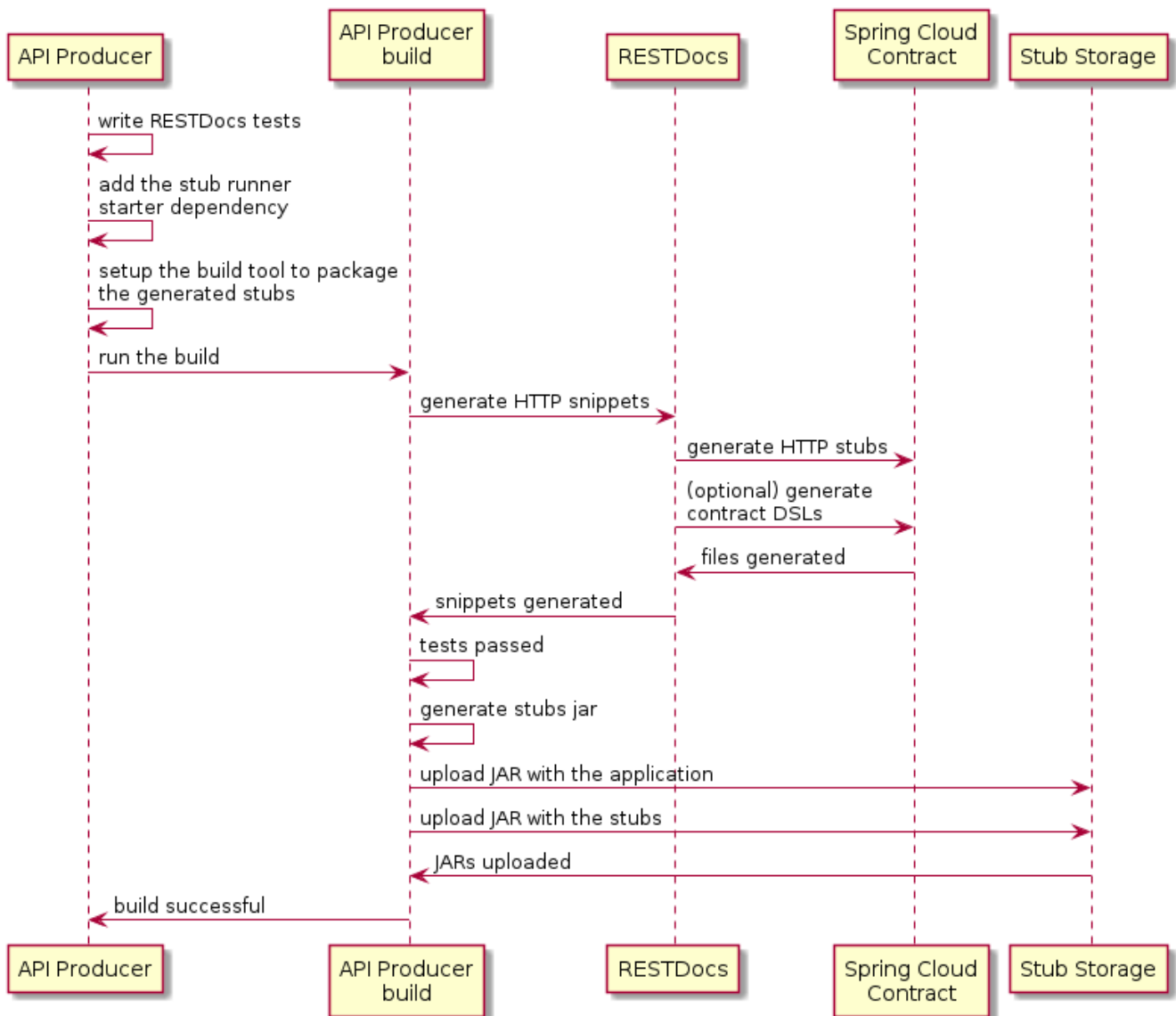
<!-- src/assembly/stub.xml -->
<assembly
  xmlns="http://maven.apache.org/plugins/maven-assembly-
plugin/assembly/1.1.3"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/plugins/maven-assembly-
plugin/assembly/1.1.3 http://maven.apache.org/xsd/assembly-1.1.3.xsd">
  <id>stubs</id>
  <formats>
    <format>jar</format>
  </formats>
  <includeBaseDirectory>>false</includeBaseDirectory>
  <fileSets>
    <fileSet>
      <directory>${project.build.directory}/generated-
snippets/stubs</directory>
      <outputDirectory>META-
INF/${project.groupId}/${project.artifactId}/${project.version}/mappings</output
Directory>
      <includes>
        <include>**/*</include>
      </includes>
    </fileSet>
  </fileSets>
</assembly>
```


Gradle

```
task stubsJar(type: Jar) {
    classifier = "stubs"
    into("META-INF/${project.group}/${project.name}/${project.version}/mappings") {
        include('**/*.*)
        from("${project.buildDir}/generated-snippets/stubs")
    }
}
// we need the tests to pass to build the stub jar
stubsJar.dependsOn(test)
bootJar.dependsOn(stubsJar)
```

Now, when we run the tests, stubs are automatically published and packaged.

The following UML diagram shows the producer flow:



2.8.2. Consumer Flow

Since the consumer flow is not affected by the tool used to generate the stubs, you can read [Developing Your First Spring Cloud Contract-based Application](#) to see the flow for consumer side of the provider contract testing with stubs in Nexus or Artifactory.

2.9. What to Read Next

You should now understand how you can use Spring Cloud Contract and some best practices that you should follow. You can now go on to learn about specific [Spring Cloud Contract features](#), or you could skip ahead and read about the [advanced features of Spring Cloud Contract](#).

3. Spring Cloud Contract Features

This section dives into the details of Spring Cloud Contract. Here you can learn about the key features that you may want to use and customize. If you have not already done so, you might want to read the "[Getting Started](#)" and "[Using Spring Cloud Contract](#)" sections, so that you have a good grounding in the basics.

3.1. Contract DSL

Spring Cloud Contract supports DSLs written in the following languages:

- Groovy
- YAML
- Java
- Kotlin



Spring Cloud Contract supports defining multiple contracts in a single file (In Groovy return a list instead of a single contract).

The following example shows a contract definition:

Groovy

```
org.springframework.cloud.contract.spec.Contract.make {
    request {
        method 'PUT'
        url '/api/12'
        headers {
            header 'Content-Type':
'application/vnd.org.springframework.cloud.contract.verifier.twitter-places-
analyzer.v1+json'
        }
        body '''\
    [{
        "created_at": "Sat Jul 26 09:38:57 +0000 2014",
        "id": 492967299297845248,
        "id_str": "492967299297845248",
        "text": "Gonna see you at Warsaw",
        "place":
        {
            "attributes": {},
            "bounding_box":
            {
                "coordinates":
                [[
                    [-77.119759,38.791645],
                    [-76.909393,38.791645],
                    [-76.909393,38.995548],
                    [-77.119759,38.995548]
                ]],
                "type": "Polygon"
            },
            "country": "United States",
            "country_code": "US",
            "full_name": "Washington, DC",
            "id": "01fbe706f872cb32",
            "name": "Washington",
            "place_type": "city",
            "url": "https://api.twitter.com/1/geo/id/01fbe706f872cb32.json"
        }
    }
    ...
}
response {
    status OK()
}
}
```

YAML

```
description: Some description
name: some name
priority: 8
ignored: true
request:
  url: /foo
  queryParameters:
    a: b
    b: c
  method: PUT
  headers:
    foo: bar
    fooReq: baz
  body:
    foo: bar
  matchers:
    body:
      - path: $.foo
        type: by_regex
        value: bar
    headers:
      - key: foo
        regex: bar
response:
  status: 200
  headers:
    foo2: bar
    foo3: foo33
    fooRes: baz
  body:
    foo2: bar
    foo3: baz
    nullValue: null
  matchers:
    body:
      - path: $.foo2
        type: by_regex
        value: bar
      - path: $.foo3
        type: by_command
        value: executeMe($it)
      - path: $.nullValue
        type: by_null
        value: null
    headers:
      - key: foo2
        regex: bar
      - key: foo3
        command: andMeToo($it)
```

Java

```
import java.util.Collection;
import java.util.Collections;
import java.util.function.Supplier;

import org.springframework.cloud.contract.spec.Contract;
import org.springframework.cloud.contract.verifier.util.ContractVerifierUtil;

class contract_rest implements Supplier<Collection<Contract>> {

    @Override
    public Collection<Contract> get() {
        return Collections.singletonList(Contract.make(c -> {
            c.description("Some description");
            c.name("some name");
            c.priority(8);
            c.ignored();
            c.request(r -> {
                r.url("/foo", u -> {
                    u.queryParameters(q -> {
                        q.parameter("a", "b");
                        q.parameter("b", "c");
                    });
                });
                r.method(r.PUT());
                r.headers(h -> {
                    h.header("foo", r.value(r.client(r.regex("bar")),
r.server("bar")));
                    h.header("fooReq", "baz");
                });
                r.body(ContractVerifierUtil.map().entry("foo", "bar"));
                r.bodyMatchers(m -> {
                    m.jsonPath("$.foo", m.byRegex("bar"));
                });
            });
            c.response(r -> {
                r.fixedDelayMilliseconds(1000);
                r.status(r.OK());
                r.headers(h -> {
                    h.header("foo2", r.value(r.server(r.regex("bar")),
r.client("bar")));
                    h.header("foo3", r.value(r.server(r.execute("andMeToo($it)")),
r.client("foo33")));
                    h.header("fooRes", "baz");
                });
                r.body(ContractVerifierUtil.map().entry("foo2",
"bar").entry("foo3", "baz").entry("nullValue", null));
                r.bodyMatchers(m -> {
                    m.jsonPath("$.foo2", m.byRegex("bar"));
                    m.jsonPath("$.foo3", m.byCommand("executeMe($it)"));
                });
            });
        });
    }
}
```

```
        m.jsonPath("$.nullValue", m.byNull());
    });
});
}
}
```

```
import org.springframework.cloud.contract.spec.ContractDsl.Companion.contract
import org.springframework.cloud.contract.spec.withQueryParameters

contract {
    name = "some name"
    description = "Some description"
    priority = 8
    ignored = true
    request {
        url = url("/foo") withQueryParameters {
            parameter("a", "b")
            parameter("b", "c")
        }
        method = PUT
        headers {
            header("foo", value(client(regex("bar")), server("bar")))
            header("fooReq", "baz")
        }
        body = body(mapOf("foo" to "bar"))
        bodyMatchers {
            jsonPath("$.foo", byRegex("bar"))
        }
    }
    response {
        delay = fixedMilliseconds(1000)
        status = OK
        headers {
            header("foo2", value(server(regex("bar")), client("bar")))
            header("foo3", value(server(execute("andMeToo(\${it}")),
client("foo33"))))
            header("fooRes", "baz")
        }
        body = body(mapOf(
            "foo" to "bar",
            "foo3" to "baz",
            "nullValue" to null
        ))
        bodyMatchers {
            jsonPath("$.foo2", byRegex("bar"))
            jsonPath("$.foo3", byCommand("executeMe(\${it}"))
            jsonPath("$.nullValue", byNull)
        }
    }
}
```



You can compile contracts to stubs mapping by using the following standalone Maven command:

```
mvn org.springframework.cloud:spring-cloud-contract-maven-  
plugin:convert
```

3.1.1. Contract DSL in Groovy

If you are not familiar with Groovy, do not worry. You can use Java syntax in the Groovy DSL files as well.

If you decide to write the contract in Groovy, do not be alarmed if you have not used Groovy before. Knowledge of the language is not really needed, as the Contract DSL uses only a tiny subset of it (only literals, method calls, and closures). Also, the DSL is statically typed, to make it programmer-readable without any knowledge of the DSL itself.



Remember that, inside the Groovy contract file, you have to provide the fully qualified name to the `Contract` class and `make` static imports, such as `org.springframework.cloud.spec.Contract.make { ... }`. You can also provide an import to the `Contract` class (`import org.springframework.cloud.spec.Contract`) and then call `Contract.make { ... }`.

3.1.2. Contract DSL in Java

To write a contract definition in Java, you need to create a class that implements either the `Supplier<Contract>` interface (for a single contract) or `Supplier<Collection<Contract>>` (for multiple contracts).

You can also write the contract definitions under `src/test/java` (for example, `src/test/java/contracts`) so that you do not have to modify the classpath of your project. In this case, you have to provide a new location of contract definitions to your Spring Cloud Contract plugin.

The following example (in both Maven and Gradle) has the contract definitions under `src/test/java`:

Maven

```
<plugin>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-contract-maven-plugin</artifactId>
  <version>${spring-cloud-contract.version}</version>
  <extensions>true</extensions>
  <configuration>
    <contractsDirectory>src/test/java/contracts</contractsDirectory>
  </configuration>
</plugin>
```

Gradle

```
contracts {
    contractsDslDir = new File(project.rootDir, "src/test/java/contracts")
}
```

3.1.3. Contract DSL in Kotlin

To get started with writing contracts in Kotlin, you need to start with a (newly created) Kotlin Script file (*.kts*). As with the Java DSL, you can put your contracts in any directory of your choice. By default, the Maven plugin will look at the `src/test/resources/contracts` directory and Gradle plugin will look at the `src/contractTest/resources/contracts` directory.



Since 3.0.0, the Gradle plugin will also look at the legacy directory `src/test/resources/contracts` for migration purposes. When contracts are found in this directory, a warning will be logged during your build.

You need to explicitly pass the `spring-cloud-contract-spec-kotlin` dependency to your project plugin setup. The following example (in both Maven and Gradle) shows how to do so:

Maven

```
<plugin>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-contract-maven-plugin</artifactId>
  <version>${spring-cloud-contract.version}</version>
  <extensions>>true</extensions>
  <configuration>
    <!-- some config -->
  </configuration>
  <dependencies>
    <dependency>
      <groupId>org.springframework.cloud</groupId>
      <artifactId>spring-cloud-contract-spec-kotlin</artifactId>
      <version>${spring-cloud-contract.version}</version>
    </dependency>
  </dependencies>
</plugin>

<dependencies>
  <!-- Remember to add this for the DSL support in the IDE and on the
consumer side -->
  <dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-contract-spec-kotlin</artifactId>
    <scope>test</scope>
  </dependency>
</dependencies>
```

Gradle

```
buildscript {
    repositories {
        // ...
    }
    dependencies {
        classpath "org.springframework.cloud:spring-cloud-contract-gradle-
plugin:${scContractVersion}"
    }
}

dependencies {
    // ...

    // Remember to add this for the DSL support in the IDE and on the consumer
side
    testImplementation "org.springframework.cloud:spring-cloud-contract-spec-
kotlin"
    // Kotlin versions are very particular down to the patch version. The
<kotlin_version> needs to be the same as you have imported for your project.
    testImplementation "org.jetbrains.kotlin:kotlin-scripting-compiler-
embeddable:<kotlin_version>"
}
```



Remember that, inside the Kotlin Script file, you have to provide the fully qualified name to the `ContractDSL` class. Generally you would use its contract function as follows: `org.springframework.cloud.contract.spec.ContractDsl.contract { ... }`. You can also provide an import to the `contract` function (`import org.springframework.cloud.contract.spec.ContractDsl.Companion.contract`) and then call `contract { ... }`.

3.1.4. Contract DSL in YAML

To see a schema of a YAML contract, visit the [YML Schema](#) page.

3.1.5. Limitations



The support for verifying the size of JSON arrays is experimental. If you want to turn it on, set the value of the following system property to `true`: `spring.cloud.contract.verifier.assert.size`. By default, this feature is set to `false`. You can also set the `assertJsonSize` property in the plugin configuration.



Because JSON structure can have any form, it can be impossible to parse it properly when using the Groovy DSL and the `value(consumer(...), producer(...))` notation in `GString`. That is why you should use the Groovy Map notation.

3.1.6. Common Top-Level Elements

The following sections describe the most common top-level elements:

- [Description](#)
- [Name](#)
- [Ignoring Contracts](#)
- [Contracts in Progress](#)
- [Passing Values from Files](#)
- [Metadata](#)

Description

You can add a `description` to your contract. The description is arbitrary text. The following code shows an example:

Groovy

```
org.springframework.cloud.contract.spec.Contract.make {  
    description(''  
given:  
    An input  
when:  
    Sth happens  
then:  
    Output  
''')  
}
```

YAML

```
description: Some description
name: some name
priority: 8
ignored: true
request:
  url: /foo
  queryParameters:
    a: b
    b: c
  method: PUT
  headers:
    foo: bar
    fooReq: baz
  body:
    foo: bar
  matchers:
    body:
      - path: $.foo
        type: by_regex
        value: bar
    headers:
      - key: foo
        regex: bar
response:
  status: 200
  headers:
    foo2: bar
    foo3: foo33
    fooRes: baz
  body:
    foo2: bar
    foo3: baz
    nullValue: null
  matchers:
    body:
      - path: $.foo2
        type: by_regex
        value: bar
      - path: $.foo3
        type: by_command
        value: executeMe($it)
      - path: $.nullValue
        type: by_null
        value: null
    headers:
      - key: foo2
        regex: bar
      - key: foo3
        command: andMeToo($it)
```

Java

```
Contract.make(c -> {
    c.description("Some description");
}));
```

Kotlin

```
contract {
    description = ""
given:
    An input
when:
    Sth happens
then:
    Output
""
}
```

Name

You can provide a name for your contract. Assume that you provide the following name: **should register a user**. If you do so, the name of the autogenerated test is **validate_should_register_a_user**. Also, the name of the stub in a WireMock stub is **should_register_a_user.json**.



You must ensure that the name does not contain any characters that make the generated test not compile. Also, remember that, if you provide the same name for multiple contracts, your autogenerated tests fail to compile and your generated stubs override each other.

The following example shows how to add a name to a contract:

Groovy

```
org.springframework.cloud.contract.spec.Contract.make {  
    name("some_special_name")  
}
```

YAML

```
name: some name
```

Java

```
Contract.make(c -> {  
    c.name("some name");  
}));
```

Kotlin

```
contract {  
    name = "some_special_name"  
}
```

Ignoring Contracts

If you want to ignore a contract, you can either set a value for ignored contracts in the plugin configuration or set the **ignored** property on the contract itself. The following example shows how to do so:

Groovy

```
org.springframework.cloud.contract.spec.Contract.make {  
    ignored()  
}
```

YAML

```
ignored: true
```

Java

```
Contract.make(c -> {  
    c.ignored();  
}));
```

Kotlin

```
contract {  
    ignored = true  
}
```

Contracts in Progress

A contract in progress does not generate tests on the producer side but does allow generation of stubs.



Use this feature with caution as it may lead to false positives, because you generate stubs for your consumers to use without actually having the implementation in place.

If you want to set a contract in progress, the following example shows how to do so:

Groovy

```
org.springframework.cloud.contract.spec.Contract.make {  
    inProgress()  
}
```

YAML

```
inProgress: true
```

Java

```
Contract.make(c -> {  
    c.inProgress();  
}));
```

Kotlin

```
contract {  
    inProgress = true  
}
```

You can set the value of the `failOnInProgress` Spring Cloud Contract plugin property to ensure that your build breaks when at least one contract in progress remains in your sources.

Passing Values from Files

Starting with version `1.2.0`, you can pass values from files. Assume that you have the following resources in your project:

```
├── src  
│   ├── test  
│   │   ├── resources  
│   │   │   ├── contracts  
│   │   │   │   ├── readFromFile.groovy  
│   │   │   │   ├── request.json  
│   │   │   │   └── response.json
```

Further assume that your contract is as follows:

Groovy

```
/*
 * Copyright 2013-2020 the original author or authors.
 *
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 *     https://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */

import org.springframework.cloud.contract.spec.Contract

Contract.make {
    request {
        method('PUT')
        headers {
            contentType(applicationJson())
        }
        body(file("request.json"))
        url("/1")
    }
    response {
        status OK()
        body(file("response.json"))
        headers {
            contentType(applicationJson())
        }
    }
}
```

YAML

```
request:
  method: GET
  url: /foo
  bodyFromFile: request.json
response:
  status: 200
  bodyFromFile: response.json
```

Java

```
import java.util.Collection;
import java.util.Collections;
import java.util.function.Supplier;

import org.springframework.cloud.contract.spec.Contract;

class contract_rest_from_file implements Supplier<Collection<Contract>> {

    @Override
    public Collection<Contract> get() {
        return Collections.singletonList(Contract.make(c -> {
            c.request(r -> {
                r.url("/foo");
                r.method(r.GET());
                r.body(r.file("request.json"));
            });
            c.response(r -> {
                r.status(r.OK());
                r.body(r.file("response.json"));
            });
        }));
    }
}
```

Kotlin

```
import org.springframework.cloud.contract.spec.ContractDsl.Companion.contract

contract {
    request {
        url = url("/1")
        method = PUT
        headers {
            contentType = APPLICATION_JSON
        }
        body = bodyFromFile("request.json")
    }
    response {
        status = OK
        body = bodyFromFile("response.json")
        headers {
            contentType = APPLICATION_JSON
        }
    }
}
```

Further assume that the JSON files are as follows:

request.json

```
{
  "status": "REQUEST"
}
```

response.json

```
{
  "status": "RESPONSE"
}
```

When test or stub generation takes place, the contents of the `request.json` and `response.json` files are passed to the body of a request or a response. The name of the file needs to be a file in a location relative to the folder in which the contract resides.

If you need to pass the contents of a file in binary form, you can use the `fileAsBytes` method in the coded DSL or a `bodyFromFileAsBytes` field in YAML.

The following example shows how to pass the contents of binary files:

Groovy

```
import org.springframework.cloud.contract.spec.Contract

Contract.make {
    request {
        url("/1")
        method(PUT())
        headers {
            contentType(applicationOctetStream())
        }
        body(fileAsBytes("request.pdf"))
    }
    response {
        status 200
        body(fileAsBytes("response.pdf"))
        headers {
            contentType(applicationOctetStream())
        }
    }
}
```

YAML

```
request:
  url: /1
  method: PUT
  headers:
    Content-Type: application/octet-stream
  bodyFromFileAsBytes: request.pdf
response:
  status: 200
  bodyFromFileAsBytes: response.pdf
  headers:
    Content-Type: application/octet-stream
```

Java

```
import java.util.Collection;
import java.util.Collections;
import java.util.function.Supplier;

import org.springframework.cloud.contract.spec.Contract;

class contract_rest_from_pdf implements Supplier<Collection<Contract>> {

    @Override
    public Collection<Contract> get() {
        return Collections.singletonList(Contract.make(c -> {
            c.request(r -> {
                r.url("/1");
                r.method(r.PUT());
                r.body(r.fileAsBytes("request.pdf"));
                r.headers(h -> {
                    h.contentType(h.applicationOctetStream());
                });
            });
            c.response(r -> {
                r.status(r.OK());
                r.body(r.fileAsBytes("response.pdf"));
                r.headers(h -> {
                    h.contentType(h.applicationOctetStream());
                });
            });
        }));
    }
}
```

Kotlin

```
import org.springframework.cloud.contract.spec.ContractDsl.Companion.contract

contract {
    request {
        url = url("/1")
        method = PUT
        headers {
            contentType = APPLICATION_OCTET_STREAM
        }
        body = bodyFromFileAsBytes("contracts/request.pdf")
    }
    response {
        status = OK
        body = bodyFromFileAsBytes("contracts/response.pdf")
        headers {
            contentType = APPLICATION_OCTET_STREAM
        }
    }
}
```



You should use this approach whenever you want to work with binary payloads, both for HTTP and messaging.

Metadata

You can add `metadata` to your contract. Via the metadata you can pass in configuration to extensions. Below you can find an example of using the `wiremock` key. Its value is a map whose key is `stubMapping` and value being WireMock's `StubMapping` object. Spring Cloud Contract is able to patch parts of your generated stub mapping with your custom code. You may want to do that in order to add webhooks, custom delays or integrate with third party WireMock extensions.

groovy

```
Contract.make {
  request {
    method GET()
    url '/drunks'
  }
  response {
    status OK()
    body([
      count: 100
    ])
    headers {
      contentType("application/json")
    }
  }
  metadata([
    wiremock: [
      stubMapping: '''\
        {
          "response" : {
            "fixedDelayMilliseconds": 2000
          }
        }
      ...
    ]
  ])
}
```

yaml

```
name: "should count all frauds"
request:
  method: GET
  url: /yamlfrauds
response:
  status: 200
  body:
    count: 200
  headers:
    Content-Type: application/json
metadata:
  wiremock:
    stubMapping: >
      {
        "response" : {
          "fixedDelayMilliseconds": 2000
        }
      }
}
```

java

```
Contract.make(c -> {
    c.metadata(MetadataUtil.map().entry("wiremock",
    ContractVerifierUtil.map().entry("stubMapping",
        "{ \"response\" : { \"fixedDelayMilliseconds\" : 2000 } }"));
}));
```

kotlin

```
contract {
    metadata("wiremock" to ("stubmapping" to ""
{
    "response" : {
        "fixedDelayMilliseconds": 2000
    }
}""))
}
```

In the following sections you can find examples of the supported metadata entries.

3.2. Contracts for HTTP

Spring Cloud Contract lets you verify applications that use REST or HTTP as a means of communication. Spring Cloud Contract verifies that, for a request that matches the criteria from the **request** part of the contract, the server provides a response that is in keeping with the **response** part of the contract. Subsequently, the contracts are used to generate WireMock stubs that, for any request matching the provided criteria, provide a suitable response.

3.2.1. HTTP Top-Level Elements

You can call the following methods in the top-level closure of a contract definition:

- **request**: Mandatory
- **response** : Mandatory
- **priority**: Optional

The following example shows how to define an HTTP request contract:

Groovy

```
org.springframework.cloud.contract.spec.Contract.make {
    // Definition of HTTP request part of the contract
    // (this can be a valid request or invalid depending
    // on type of contract being specified).
    request {
        method GET()
        url "/foo"
        //...
    }

    // Definition of HTTP response part of the contract
    // (a service implementing this contract should respond
    // with following response after receiving request
    // specified in "request" part above).
    response {
        status 200
        //...
    }

    // Contract priority, which can be used for overriding
    // contracts (1 is highest). Priority is optional.
    priority 1
}
```

YAML

```
priority: 8
request:
...
response:
...
```

Java

```
org.springframework.cloud.contract.spec.Contract.make(c -> {
    // Definition of HTTP request part of the contract
    // (this can be a valid request or invalid depending
    // on type of contract being specified).
    c.request(r -> {
        r.method(r.GET());
        r.url("/foo");
        // ...
    });

    // Definition of HTTP response part of the contract
    // (a service implementing this contract should respond
    // with following response after receiving request
    // specified in "request" part above).
    c.response(r -> {
        r.status(200);
        // ...
    });

    // Contract priority, which can be used for overriding
    // contracts (1 is highest). Priority is optional.
    c.priority(1);
});
```

Kotlin

```
contract {
    // Definition of HTTP request part of the contract
    // (this can be a valid request or invalid depending
    // on type of contract being specified).
    request {
        method = GET
        url = url("/foo")
        // ...
    }

    // Definition of HTTP response part of the contract
    // (a service implementing this contract should respond
    // with following response after receiving request
    // specified in "request" part above).
    response {
        status = OK
        // ...
    }

    // Contract priority, which can be used for overriding
    // contracts (1 is highest). Priority is optional.
    priority = 1
}
```



If you want to make your contract have a higher priority, you need to pass a lower number to the `priority` tag or method. For example, a `priority` with a value of `5` has higher priority than a `priority` with a value of `10`.

3.2.2. HTTP Request

The HTTP protocol requires only the method and the URL to be specified in a request. The same information is mandatory in request definition of the contract.

The following example shows a contract for a request:

Groovy

```
org.springframework.cloud.contract.spec.Contract.make {
    request {
        // HTTP request method (GET/POST/PUT/DELETE).
        method 'GET'

        // Path component of request URL is specified as follows.
        urlPath('/users')
    }

    response {
        //...
        status 200
    }
}
```

YAML

```
method: PUT
url: /foo
```

Java

```
org.springframework.cloud.contract.spec.Contract.make(c -> {
    c.request(r -> {
        // HTTP request method (GET/POST/PUT/DELETE).
        r.method("GET");

        // Path component of request URL is specified as follows.
        r.urlPath("/users");
    });

    c.response(r -> {
        // ...
        r.status(200);
    });
});
```

Kotlin

```
contract {
    request {
        // HTTP request method (GET/POST/PUT/DELETE).
        method = method("GET")

        // Path component of request URL is specified as follows.
        urlPath = path("/users")
    }
    response {
        // ...
        status = code(200)
    }
}
```

You can specify an absolute rather than a relative `url`, but using `urlPath` is the recommended way, as doing so makes the tests be host-independent.

The following example uses `url`:

Groovy

```
org.springframework.cloud.contract.spec.Contract.make {
    request {
        method 'GET'

        // Specifying `url` and `urlPath` in one contract is illegal.
        url('http://localhost:8888/users')
    }

    response {
        //...
        status 200
    }
}
```

YAML

```
request:
  method: PUT
  urlPath: /foo
```

Java

```
org.springframework.cloud.contract.spec.Contract.make(c -> {
    c.request(r -> {
        r.method("GET");

        // Specifying `url` and `urlPath` in one contract is illegal.
        r.url("http://localhost:8888/users");
    });

    c.response(r -> {
        // ...
        r.status(200);
    });
});
```


Kotlin

```
contract {
    request {
        method = GET

        // Specifying `url` and `urlPath` in one contract is illegal.
        url("http://localhost:8888/users")
    }
    response {
        // ...
        status = OK
    }
}
```

`request` may contain query parameters, as the following example (which uses `urlPath`) shows:

Groovy

```
org.springframework.cloud.contract.spec.Contract.make {
    request {
        //...
        method GET()

        urlPath('/users') {

            // Each parameter is specified in form
            // `paramName` : paramValue` where parameter value
            // may be a simple literal or one of matcher functions,
            // all of which are used in this example.
            queryParameters {

                // If a simple literal is used as value
                // default matcher function is used (equalTo)
                parameter 'limit': 100

                // `equalTo` function simply compares passed value
                // using identity operator (==).
                parameter 'filter': equalTo("email")

                // `containing` function matches strings
                // that contains passed substring.
                parameter 'gender': value(consumer(containing("[mf]")),
producer('mf'))

                // `matching` function tests parameter
                // against passed regular expression.
                parameter 'offset': value(consumer(matching("[0-9]+")),
producer(123))

                // `notMatching` functions tests if parameter
                // does not match passed regular expression.
                parameter 'loginStartsWith':
value(consumer(notMatching(".{0,2}")), producer(3))
            }
        }

        //...
    }

    response {
        //...
        status 200
    }
}
```

YAML

```
request:  
...  
queryParameters:  
  a: b  
  b: c
```

```
org.springframework.cloud.contract.spec.Contract.make(c -> {
    c.request(r -> {
        // ...
        r.method(r.GET());

        r.urlPath("/users", u -> {

            // Each parameter is specified in form
            // `paramName` : paramValue` where parameter value
            // may be a simple literal or one of matcher functions,
            // all of which are used in this example.
            u.queryParameters(q -> {

                // If a simple literal is used as value
                // default matcher function is used (equalTo)
                q.parameter("limit", 100);

                // `equalTo` function simply compares passed value
                // using identity operator (==).
                q.parameter("filter", r.equalTo("email"));

                // `containing` function matches strings
                // that contains passed substring.
                q.parameter("gender", r.value(r.consumer(r.containing("[mf]")),
r.producer("mf"))));

                // `matching` function tests parameter
                // against passed regular expression.
                q.parameter("offset", r.value(r.consumer(r.matching("[0-9]+")),
r.producer(123)));

                // `notMatching` functions tests if parameter
                // does not match passed regular expression.
                q.parameter("loginStartsWith",
r.value(r.consumer(r.notMatching(".{0,2}")), r.producer(3)));
            });
        });

        // ...
    });

    c.response(r -> {
        // ...
        r.status(200);
    });
});
```

Kotlin

```
contract {
    request {
        // ...
        method = GET

        // Each parameter is specified in form
        // `paramName` : paramValue` where parameter value
        // may be a simple literal or one of matcher functions,
        // all of which are used in this example.
        urlPath = path("/users") withQueryParameters {
            // If a simple literal is used as value
            // default matcher function is used (equalTo)
            parameter("limit", 100)

            // `equalTo` function simply compares passed value
            // using identity operator (==).
            parameter("filter", equalTo("email"))

            // `containing` function matches strings
            // that contains passed substring.
            parameter("gender", value(consumer(containing("[mf]"))),
producer("mf")))

            // `matching` function tests parameter
            // against passed regular expression.
            parameter("offset", value(consumer(matching("[0-9]+")),
producer(123)))

            // `notMatching` functions tests if parameter
            // does not match passed regular expression.
            parameter("loginStartsWith", value(consumer(notMatching(".{0,2}")),
producer(3)))
        }
        // ...
    }
    response {
        // ...
        status = code(200)
    }
}
```

request can contain additional request headers, as the following example shows:

Groovy

```
org.springframework.cloud.contract.spec.Contract.make {
    request {
        //...
        method GET()
        url "/foo"

        // Each header is added in form `Header-Name' : 'Header-Value'`.
        // there are also some helper methods
        headers {
            header 'key': 'value'
            contentType(applicationJson())
        }

        //...
    }

    response {
        //...
        status 200
    }
}
```

YAML

```
request:
  ...
headers:
  foo: bar
  fooReq: baz
```

Java

```
org.springframework.cloud.contract.spec.Contract.make(c -> {
    c.request(r -> {
        // ...
        r.method(r.GET());
        r.url("/foo");

        // Each header is added in form `Header-Name` : `Header-Value`.
        // there are also some helper methods
        r.headers(h -> {
            h.header("key", "value");
            h.contentType(h.applicationJson());
        });

        // ...
    });

    c.response(r -> {
        // ...
        r.status(200);
    });
});
```

Kotlin

```
contract {
    request {
        // ...
        method = GET
        url = url("/foo")

        // Each header is added in form `Header-Name` : `Header-Value`.
        // there are also some helper variables
        headers {
            header("key", "value")
            contentType = APPLICATION_JSON
        }

        // ...
    }
    response {
        // ...
        status = OK
    }
}
```

request may contain additional request cookies, as the following example shows:

Groovy

```
org.springframework.cloud.contract.spec.Contract.make {
    request {
        //...
        method GET()
        url "/foo"

        // Each Cookies is added in form `Cookie-Key' : 'Cookie-Value'`.
        // there are also some helper methods
        cookies {
            cookie 'key': 'value'
            cookie('another_key', 'another_value')
        }

        //...
    }

    response {
        //...
        status 200
    }
}
```

YAML

```
request:
  ...
cookies:
  foo: bar
  fooReq: baz
```


Java

```
org.springframework.cloud.contract.spec.Contract.make(c -> {
    c.request(r -> {
        // ...
        r.method(r.GET());
        r.url("/foo");

        // Each Cookies is added in form ``Cookie-Key' : 'Cookie-Value'`.
        // there are also some helper methods
        r.cookies(ck -> {
            ck.cookie("key", "value");
            ck.cookie("another_key", "another_value");
        });

        // ...
    });

    c.response(r -> {
        // ...
        r.status(200);
    });
});
```

Kotlin

```
contract {
    request {
        // ...
        method = GET
        url = url("/foo")

        // Each Cookies is added in form ``Cookie-Key' : 'Cookie-Value'`.
        // there are also some helper methods
        cookies {
            cookie("key", "value")
            cookie("another_key", "another_value")
        }

        // ...
    }

    response {
        // ...
        status = code(200)
    }
}
```

request may contain a request body, as the following example shows:

Groovy

```
org.springframework.cloud.contract.spec.Contract.make {
    request {
        //...
        method GET()
        url "/foo"

        // Currently only JSON format of request body is supported.
        // Format will be determined from a header or body's content.
        body '''{ "login" : "john", "name": "John The Contract" }'''
    }

    response {
        //...
        status 200
    }
}
```

YAML

```
request:
  ...
body:
  foo: bar
```

Java

```
org.springframework.cloud.contract.spec.Contract.make(c -> {
    c.request(r -> {
        // ...
        r.method(r.GET());
        r.url("/foo");

        // Currently only JSON format of request body is supported.
        // Format will be determined from a header or body's content.
        r.body("{\"login\" : \"john\", \"name\": \"John The Contract\" }");
    });

    c.response(r -> {
        // ...
        r.status(200);
    });
});
```

Kotlin

```
contract {
    request {
        // ...
        method = GET
        url = url("/foo")

        // Currently only JSON format of request body is supported.
        // Format will be determined from a header or body's content.
        body = body("{ \"login\" : \"john\", \"name\": \"John The Contract\" }")
    }
    response {
        // ...
        status = OK
    }
}
```

`request` can contain multipart elements. To include multipart elements, use the `multipart` method/section, as the following examples show:

Groovy

```
org.springframework.cloud.contract.spec.Contract contractDsl =
org.springframework.cloud.contract.spec.Contract.make {
    request {
        method 'PUT'
        url '/multipart'
        headers {
            contentType('multipart/form-data;boundary=AaB03x')
        }
        multipart(
            // key (parameter name), value (parameter value) pair
            formParameter: $(c(regex('.+')), p("formParameterValue")),
            someBooleanParameter: $(c(regex(anyBoolean())), p('true')),
            // a named parameter (e.g. with `file` name) that represents file
with
            // `name` and `content`. You can also call `named("fileName",
"fileContent")`
            file: named(
                // name of the file
                name: $(c(regex(nonEmpty())), p('filename.csv')),
                // content of the file
                content: $(c(regex(nonEmpty())), p('file content')),
                // content type for the part
                contentType: $(c(regex(nonEmpty()))),
p('application/json'))
        )
    }
    response {
        status OK()
    }
}

org.springframework.cloud.contract.spec.Contract contractDsl =
org.springframework.cloud.contract.spec.Contract.make {
    request {
        method "PUT"
        url "/multipart"
        headers {
            contentType('multipart/form-data;boundary=AaB03x')
        }
        multipart(
            file: named(
                name: value(stub(regex('.+')), test('file')),
                content: value(stub(regex('.+')), test([100, 117, 100, 97]
as byte[]))
            )
        )
    }
    response {
        status 200
    }
}
```

```
}  
}
```

YAML

```
request:  
  method: PUT  
  url: /multipart  
  headers:  
    Content-Type: multipart/form-data;boundary=AaB03x  
  multipart:  
    params:  
      # key (parameter name), value (parameter value) pair  
      formParameter: "formParameterValue"  
      someBooleanParameter: true  
    named:  
      - paramName: file  
        fileName: filename.csv  
        fileContent: file content  
  matchers:  
    multipart:  
      params:  
        - key: formParameter  
          regex: ".+"  
        - key: someBooleanParameter  
          predefined: any_boolean  
      named:  
        - paramName: file  
          fileName:  
            predefined: non_empty  
          fileContent:  
            predefined: non_empty  
  response:  
    status: 200
```

Java

```
import java.util.Collection;  
import java.util.Collections;  
import java.util.HashMap;  
import java.util.Map;  
import java.util.function.Supplier;  
  
import org.springframework.cloud.contract.spec.Contract;  
import org.springframework.cloud.contract.spec.internal.DslProperty;  
import org.springframework.cloud.contract.spec.internal.Request;  
import org.springframework.cloud.contract.verifier.util.ContractVerifierUtil;  
  
class contract_multipart implements Supplier<Collection<Contract>> {
```

```

private static Map<String, DslProperty> namedProps(Request r) {
    Map<String, DslProperty> map = new HashMap<>();
    // name of the file
    map.put("name", r.$(r.c(r.regex(r.nonEmpty())), r.p("filename.csv")));
    // content of the file
    map.put("content", r.$(r.c(r.regex(r.nonEmpty())), r.p("file content")));
    // content type for the part
    map.put("contentType", r.$(r.c(r.regex(r.nonEmpty())),
r.p("application/json")));
    return map;
}

@Override
public Collection<Contract> get() {
    return Collections.singletonList(Contract.make(c -> {
        c.request(r -> {
            r.method("PUT");
            r.url("/multipart");
            r.headers(h -> {
                h.contentType("multipart/form-data;boundary=AaB03x");
            });
            r.multipart(ContractVerifierUtil.map()
                // key (parameter name), value (parameter value) pair
                .entry("formParameter", r.$(r.c(r.regex("\\.+\\")),
r.p("\\formParameterValue\\")))
                .entry("someBooleanParameter",
r.$(r.c(r.regex(r.anyBoolean())), r.p("true")))
                // a named parameter (e.g. with `file` name) that
                // represents file
                // with
                // `name` and `content`. You can also call
                `named("fileName",
                // "fileContent")`
                .entry("file", r.named(namedProps(r))));
            });
        c.response(r -> {
            r.status(r.OK());
        });
    }));
}
}

```

Kotlin

```
import org.springframework.cloud.contract.spec.ContractDsl.Companion.contract

contract {
    request {
        method = PUT
        url = url("/multipart")
        multipart {
            field("formParameter", value(consumer(regex("\\.+\\")),
producer("\\formParameterValue\\")))
            field("someBooleanParameter", value(consumer(anyBoolean),
producer("true")))
            field("file",
                named(
                    // name of the file
                    value(consumer(regex(nonEmpty)), producer("filename.csv")),
                    // content of the file
                    value(consumer(regex(nonEmpty)), producer("file content")),
                    // content type for the part
                    value(consumer(regex(nonEmpty)), producer("application/json"))
                )
            )
        }
        headers {
            contentType = "multipart/form-data;boundary=AaB03x"
        }
    }
    response {
        status = OK
    }
}
```

In the preceding example, we defined parameters in either of two ways:

Coded DSL

- Directly, by using the map notation, where the value can be a dynamic property (such as `formParameter: $(consumer(...), producer(...))`).
- By using the `named(...)` method that lets you set a named parameter. A named parameter can set a `name` and `content`. You can call it either by using a method with two arguments, such as `named("fileName", "fileContent")`, or by using a map notation, such as `named(name: "fileName", content: "fileContent")`.

YAML

- The multipart parameters are set in the `multipart.params` section.
- The named parameters (the `fileName` and `fileContent` for a given parameter name) can be set in the `multipart.named` section. That section contains the `paramName` (the name of the parameter), `fileName` (the name of the file), `fileContent` (the content of the file) fields.

- The dynamic bits can be set in the `matchers.multipart` section.
 - For parameters, use the `params` section, which can accept `regex` or a `predefined` regular expression.
 - For named parameters, use the `named` section where you first define the parameter name with `paramName`. Then you can pass the parametrization of either `fileName` or `fileContent` in a `regex` or in a `predefined` regular expression.



For the `named(...)` section you always have to add a pair of `value(producer(...), consumer(...))` calls. Just setting DSL properties such as just `value(producer(...))` or just `file(...)` will not work. Check this [issue](#) for more information.

From the contract in the preceding example, the generated test and stub look as follows:

Test

```
// given:
MockMvcRequestSpecification request = given()
    .header("Content-Type", "multipart/form-data;boundary=AaB03x")
    .param("formParameter", "\"formParameterValue\"")
    .param("someBooleanParameter", "true")
    .multiPart("file", "filename.csv", "file content".getBytes());

// when:
ResponseOptions response = given().spec(request)
    .put("/multipart");

// then:
assertThat(response.statusCode()).isEqualTo(200);
```

Stub

```
    ...
  {
    "request" : {
      "url" : "/multipart",
      "method" : "PUT",
      "headers" : {
        "Content-Type" : {
          "matches" : "multipart/form-data;boundary=AaB03x.*"
        }
      },
      "bodyPatterns" : [ {
        "matches" : ".*--(.*)\\r?\\nContent-Disposition: form-data;
name=\\\"formParameter\\\"\\r?\\n(Content-Type: .*\\r?\\n)?(Content-Transfer-
Encoding: .*\\r?\\n)?(Content-Length: \\d+\\r?\\n)?\\r?\\n\\\".+\\\"\\r?\\n--.*"
      }, {
        "matches" : ".*--(.*)\\r?\\nContent-Disposition: form-data;
name=\\\"someBooleanParameter\\\"\\r?\\n(Content-Type: .*\\r?\\n)?(Content-Transfer-
Encoding: .*\\r?\\n)?(Content-Length: \\d+\\r?\\n)?\\r?\\n(true|false)\\r?\\n--
.*"
      }, {
        "matches" : ".*--(.*)\\r?\\nContent-Disposition: form-data; name=\\\"file\\\";
filename=\\\"[\\\\S\\\\s]+\\\"\\r?\\n(Content-Type: .*\\r?\\n)?(Content-Transfer-
Encoding: .*\\r?\\n)?(Content-Length: \\d+\\r?\\n)?\\r?\\n[\\\\S\\\\s]+\\r?\\n--
.*"
      } ]
    },
    "response" : {
      "status" : 200,
      "transformers" : [ "response-template", "foo-transformer" ]
    }
  }
  ...
}
```

3.2.3. HTTP Response

The response must contain an HTTP status code and may contain other information. The following code shows an example:

Groovy

```
org.springframework.cloud.contract.spec.Contract.make {
    request {
        //...
        method GET()
        url "/foo"
    }
    response {
        // Status code sent by the server
        // in response to request specified above.
        status OK()
    }
}
```

YAML

```
response:
  ...
  status: 200
```

Java

```
org.springframework.cloud.contract.spec.Contract.make(c -> {
    c.request(r -> {
        // ...
        r.method(r.GET());
        r.url("/foo");
    });
    c.response(r -> {
        // Status code sent by the server
        // in response to request specified above.
        r.status(r.OK());
    });
});
```

Kotlin

```
contract {
  request {
    // ...
    method = GET
    url =url("/foo")
  }
  response {
    // Status code sent by the server
    // in response to request specified above.
    status = OK
  }
}
```

Besides status, the response may contain headers, cookies, and a body, which are specified the same way as in the request (see [HTTP Request](#)).



In the Groovy DSL, you can reference the `org.springframework.cloud.contract.spec.internal.HttpStatus` methods to provide a meaningful status instead of a digit. For example, you can call `OK()` for a status `200` or `BAD_REQUEST()` for `400`.

3.2.4. Dynamic properties

The contract can contain some dynamic properties: timestamps, IDs, and so on. You do not want to force the consumers to stub their clocks to always return the same value of time so that it gets matched by the stub.

For the Groovy DSL, you can provide the dynamic parts in your contracts in two ways: pass them directly in the body or set them in a separate section called `bodyMatchers`.



Before 2.0.0, these were set by using `testMatchers` and `stubMatchers`. See the [migration guide](#) for more information.

For YAML, you can use only the `matchers` section.



Entries inside the `matchers` must reference existing elements of the payload. For more information, see [this issue](#).

Dynamic Properties inside the Body



This section is valid only for the Coded DSL (Groovy, Java, and so on). See the [Dynamic Properties in the Matchers Sections](#) section for YAML examples of a similar feature.

You can set the properties inside the body either with the `value` method or, if you use the Groovy

map notation, with `$()`. The following example shows how to set dynamic properties with the `value` method:

value

```
value(consumer(...), producer(...))
value(c(...), p(...))
value(stub(...), test(...))
value(client(...), server(...))
```

\$

```
$(consumer(...), producer(...))
$(c(...), p(...))
$(stub(...), test(...))
$(client(...), server(...))
```

Both approaches work equally well. The `stub` and `client` methods are aliases over the `consumer` method. Subsequent sections take a closer look at what you can do with those values.

Regular Expressions



This section is valid only for the Groovy DSL. See the [Dynamic Properties in the Matchers Sections](#) section for YAML examples of a similar feature.

You can use regular expressions to write your requests in the contract DSL. Doing so is particularly useful when you want to indicate that a given response should be provided for requests that follow a given pattern. Also, you can use regular expressions when you need to use patterns and not exact values both for your tests and your server-side tests.

Make sure that regex matches a whole region of a sequence, as, internally, `Pattern.matches()` is called. For instance, `abc` does not match `abc`, but `.abc` does. There are several additional [known limitations](#) as well.

The following example shows how to use regular expressions to write a request:

Groovy

```
org.springframework.cloud.contract.spec.Contract.make {
    request {
        method('GET')
        url $(consumer(~/\[/[0-9]{2}/), producer('/12'))
    }
    response {
        status OK()
        body(
            id: $(anyNumber()),
            surname: $(
                consumer('Kowalsky'),
                producer(regex('[a-zA-Z]+'))
            ),
            name: 'Jan',
            created: $(consumer('2014-02-02 12:23:43')),
            producer(execute('currentDate(it)')),
            correlationId: value(consumer('5d1f9fef-e0dc-4f3d-a7e4-
72d2220dd827'),
                producer(regex('[a-fA-F0-9]{8}-[a-fA-F0-9]{4}-[a-fA-F0-
9]{4}-[a-fA-F0-9]{4}-[a-fA-F0-9]{12}'))
            )
        )
        headers {
            header 'Content-Type': 'text/plain'
        }
    }
}
```

Java

```
org.springframework.cloud.contract.spec.Contract.make(c -> {
    c.request(r -> {
        r.method("GET");
        r.url(r.$(r.consumer(r.regex("\\[/[0-9]{2}")), r.producer("/12"))));
    });
    c.response(r -> {
        r.status(r.OK());
        r.body(ContractVerifierUtil.map().entry("id",
r.$(r.anyNumber())).entry("surname",
            r.$(r.consumer("Kowalsky"), r.producer(r.regex("[a-zA-Z]+"))));
        r.headers(h -> {
            h.header("Content-Type", "text/plain");
        });
    });
});
```

Kotlin

```
contract {
    request {
        method = method("GET")
        url = url(v(consumer(regex("\\/[0-9]{2}")), producer("/12")))
    }
    response {
        status = OK
        body(mapOf(
            "id" to v(anyNumber),
            "surname" to v(consumer("Kowalsky"), producer(regex("[a-zA-Z]+"))))
        ))
        headers {
            header("Content-Type", "text/plain")
        }
    }
}
```

You can also provide only one side of the communication with a regular expression. If you do so, then the contract engine automatically provides the generated string that matches the provided regular expression. The following code shows an example for Groovy:

```
org.springframework.cloud.contract.spec.Contract.make {
    request {
        method 'PUT'
        url value(consumer(regex('/foo/[0-9]{5}')))
        body([
            requestElement: $(consumer(regex('[0-9]{5}'))
        ])
        headers {
            header('header',
$(consumer(regex('application/vnd\\.fraud\\.v1\\.json;.*'))))
        }
    }
    response {
        status OK()
        body([
            responseElement: $(producer(regex('[0-9]{7}'))
        ])
        headers {
            contentType("application/vnd.fraud.v1+json")
        }
    }
}
```

In the preceding example, the opposite side of the communication has the respective data generated for request and response.

Spring Cloud Contract comes with a series of predefined regular expressions that you can use in your contracts, as the following example shows:

```
public static RegexProperty onlyAlphaUnicode() {
    return new RegexProperty(ONLY_ALPHA_UNICODE).asString();
}

public static RegexProperty alphaNumeric() {
    return new RegexProperty(ALPHA_NUMERIC).asString();
}

public static RegexProperty number() {
    return new RegexProperty(NUMBER).asDouble();
}

public static RegexProperty positiveInt() {
    return new RegexProperty(POSITIVE_INT).asInteger();
}

public static RegexProperty anyBoolean() {
    return new RegexProperty(TRUE_OR_FALSE).asBooleanType();
}

public static RegexProperty anInteger() {
    return new RegexProperty(INTEGER).asInteger();
}

public static RegexProperty aDouble() {
    return new RegexProperty(DOUBLE).asDouble();
}

public static RegexProperty ipAddress() {
    return new RegexProperty(IP_ADDRESS).asString();
}

public static RegexProperty hostname() {
    return new RegexProperty(HOSTNAME_PATTERN).asString();
}

public static RegexProperty email() {
    return new RegexProperty(EMAIL).asString();
}

public static RegexProperty url() {
    return new RegexProperty(URL).asString();
}

public static RegexProperty httpsUrl() {
    return new RegexProperty(HTTPS_URL).asString();
}
```



```
public static RegexProperty uuid() {
    return new RegexProperty(UUID).asString();
}

public static RegexProperty uuid4() {
    return new RegexProperty(UUID4).asString();
}

public static RegexProperty isoDate() {
    return new RegexProperty(ANY_DATE).asString();
}

public static RegexProperty isoDateTime() {
    return new RegexProperty(ANY_DATE_TIME).asString();
}

public static RegexProperty isoTime() {
    return new RegexProperty(ANY_TIME).asString();
}

public static RegexProperty iso8601WithOffset() {
    return new RegexProperty(ISO8601_WITH_OFFSET).asString();
}

public static RegexProperty nonEmpty() {
    return new RegexProperty(NON_EMPTY).asString();
}

public static RegexProperty nonBlank() {
    return new RegexProperty(NON_BLANK).asString();
}
```

In your contract, you can use it as follows (example for the Groovy DSL):

```

Contract dslWithOptionalsInString = Contract.make {
  priority 1
  request {
    method POST()
    url '/users/password'
    headers {
      contentType(applicationJson())
    }
    body(
      email: $(consumer(optional(regex(email()))), producer('abc@abc.com')),
      callback_url: $(consumer(regex(hostname()))),
      producer('http://partners.com')
    )
  }
  response {
    status 404
    headers {
      contentType(applicationJson())
    }
    body(
      code: value(consumer("123123"), producer(optional("123123"))),
      message: "User not found by email = [${value(producer(regex(email()))),
      consumer('not.existing@user.com'))}]"
    )
  }
}

```

To make matters even simpler, you can use a set of predefined objects that automatically assume that you want a regular expression to be passed. All of those methods start with the **any** prefix, as follows:

```
T anyAlphaUnicode();
T anyAlphaNumeric();
T anyNumber();
T anyInteger();
T anyPositiveInt();
T anyDouble();
T anyHex();
T aBoolean();
T anyIpAddress();
T anyHostname();
T anyEmail();
T anyUrl();
T anyHttpsUrl();
T anyUuid();
T anyDate();
T anyDateTime();
T anyTime();
T anyIso8601WithOffset();
T anyNonBlankString();
T anyNonEmptyString();
T anyOf(String... values);
```

The following example shows how you can reference those methods:

Groovy

```
Contract contractDsl = Contract.make {
    name "foo"
    label 'trigger_event'
    input {
        triggeredBy('toString()')
    }
    outputMessage {
        sentTo 'topic.rateablequote'
        body([
            alpha          : $(anyAlphaUnicode()),
            number         : $(anyNumber()),
            anInteger      : $(anyInteger()),
            positiveInt    : $(anyPositiveInt()),
            aDouble        : $(anyDouble()),
            aBoolean       : $(aBoolean()),
            ip              : $(anyIpAddress()),
            hostname       : $(anyHostname()),
            email          : $(anyEmail()),
            url             : $(anyUrl()),
            httpsUrl       : $(anyHttpsUrl()),
            uuid           : $(anyUuid()),
            date           : $(anyDate()),
            dateTime       : $(anyDateTime()),
            time           : $(anyTime()),
            iso8601WithOffset: $(anyIso8601WithOffset()),
            nonBlankString : $(anyNonBlankString()),
            nonEmptyString : $(anyNonEmptyString()),
            anyOf          : $(anyOf('foo', 'bar'))
        ])
    }
}
```

Kotlin

```
contract {
    name = "foo"
    label = "trigger_event"
    input {
        triggeredBy = "toString()"
    }
    outputMessage {
        sentTo = sentTo("topic.rateablequote")
        body(mapOf(
            "alpha" to v(anyAlphaUnicode),
            "number" to v(anyNumber),
            "anInteger" to v(anyInteger),
            "positiveInt" to v(anyPositiveInt),
            "aDouble" to v(anyDouble),
            "aBoolean" to v(aBoolean),
            "ip" to v(anyIpAddress),
            "hostname" to v(anyAlphaUnicode),
            "email" to v(anyEmail),
            "url" to v(anyUrl),
            "httpsUrl" to v(anyHttpsUrl),
            "uuid" to v(anyUuid),
            "date" to v(anyDate),
            "dateTime" to v(anyDateTime),
            "time" to v(anyTime),
            "iso8601WithOffset" to v(anyIso8601WithOffset),
            "nonBlankString" to v(anyNonBlankString),
            "nonEmptyString" to v(anyNonEmptyString),
            "anyOf" to v(anyOf('foo', 'bar'))
        ))
        headers {
            header("Content-Type", "text/plain")
        }
    }
}
```

Limitations



Due to certain limitations of the [Xeger](#) library that generates a string out of a regex, do not use the `$` and `^` signs in your regex if you rely on automatic generation. See [Issue 899](#).



Do not use a `LocalDate` instance as a value for `$` (for example, `$(consumer(LocalDate.now()))`). It causes a `java.lang.StackOverflowError`. Use `$(consumer(LocalDate.now().toString()))` instead. See [Issue 900](#).

Passing Optional Parameters



This section is valid only for Groovy DSL. See the [Dynamic Properties in the Matchers Sections](#) section for YAML examples of a similar feature.

You can provide optional parameters in your contract. However, you can provide optional parameters only for the following:

- The STUB side of the Request
- The TEST side of the Response

The following example shows how to provide optional parameters:

Groovy

```
org.springframework.cloud.contract.spec.Contract.make {
    priority 1
    name "optionals"
    request {
        method 'POST'
        url '/users/password'
        headers {
            contentType(applicationJson())
        }
        body(
            email: $(consumer(optional(regex(email()))),
producer('abc@abc.com')),
            callback_url: $(consumer(regex(hostname())),
producer('https://partners.com'))
        )
    }
    response {
        status 404
        headers {
            header 'Content-Type': 'application/json'
        }
        body(
            code: value(consumer("123123"), producer(optional("123123")))
        )
    }
}
```

Java

```
org.springframework.cloud.contract.spec.Contract.make(c -> {
    c.priority(1);
    c.name("optionals");
    c.request(r -> {
        r.method("POST");
        r.url("/users/password");
        r.headers(h -> {
            h.contentType(h.applicationJson());
        });
        r.body(ContractVerifierUtil.map()
            .entry("email", r.$(r.consumer(r.optional(r.regex(r.email())))),
r.producer("abc@abc.com")))
            .entry("callback_url",
                r.$(r.consumer(r.regex(r.hostname()))),
r.producer("https://partners.com"))));
    });
    c.response(r -> {
        r.status(404);
        r.headers(h -> {
            h.header("Content-Type", "application/json");
        });
        r.body(ContractVerifierUtil.map().entry("code",
            r.value(r.consumer("123123"), r.producer(r.optional("123123")))));
    });
});
```


Kotlin

```
contract { c ->
    priority = 1
    name = "optionals"
    request {
        method = POST
        url = url("/users/password")
        headers {
            contentType = APPLICATION_JSON
        }
        body = body(mapOf(
            "email" to v(consumer(optional(regex(email))),
producer("abc@abc.com")),
            "callback_url" to v(consumer(regex(hostname)),
producer("https://partners.com"))
        ))
    }
    response {
        status = NOT_FOUND
        headers {
            header("Content-Type", "application/json")
        }
        body(mapOf(
            "code" to value(consumer("123123"), producer(optional("123123")))
        ))
    }
}
```

By wrapping a part of the body with the `optional()` method, you create a regular expression that must be present 0 or more times.

If you use Spock, the following test would be generated from the previous example:

Groovy

```
package com.example

import com.jayway.jsonpath.DocumentContext
import com.jayway.jsonpath.JsonPath
import spock.lang.Specification
import io.restassured.module.mockmvc.specification.MockMvcRequestSpecification
import io.restassured.response.ResponseOptions

import static
org.springframework.cloud.contract.verifier.assertion.SpringCloudContractAssertion
s.assertThat
import static
org.springframework.cloud.contract.verifier.util.ContractVerifierUtil.*
import static com.toomuchcoding.jsonassert.JsonAssertion.assertThatJson
import static io.restassured.module.mockmvc.RestAssuredMockMvc.*

@SuppressWarnings("rawtypes")
class FooSpec extends Specification {

    \tdef validate_optionals() throws Exception {
    \t\tgiven:
    \t\t\tMockMvcRequestSpecification request = given()
    \t\t\t\t.header("Content-Type", "application/json")
    \t\t\t\t.body('{"email":"abc@abc.com","callback_url":"https://partners.com}''
    \t\t\t\t')

    \t\twhen:
    \t\t\tResponseOptions response = given().spec(request)
    \t\t\t\t.post("/users/password")

    \t\tthen:
    \t\t\tresponse.statusCode() == 404
    \t\t\tresponse.header("Content-Type") == 'application/json'

    \t\tand:
    \t\t\tDocumentContext parsedJson = JsonPath.parse(response.body.asString())
    \t\t\tassertThatJson(parsedJson).field("[code]").matches("(123123)?")
    \t\t}

    }
}
```

The following stub would also be generated:

```

    ...
  {
    "request" : {
      "url" : "/users/password",
      "method" : "POST",
      "bodyPatterns" : [ {
        "matchesJsonPath" : "$[?(@['email'] =~ /([a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-
]+\\\\.\\\\[a-zA-Z]{2,6})?/)]"
      }, {
        "matchesJsonPath" : "$[?(@['callback_url'] =~
/((http[s]?|ftp):\\\\/\\\\/)?(?:[^\:\\\\/\\\\s]+)(:[0-9]{1,5})?/)]"
      } ],
      "headers" : {
        "Content-Type" : {
          "equalTo" : "application/json"
        }
      }
    },
    "response" : {
      "status" : 404,
      "body" : "{\\\"code\\\":\\\"123123\\\",\\\"message\\\":\\\"User not found by email ==
[not.existing@user.com]\\\"}",
      "headers" : {
        "Content-Type" : "application/json"
      }
    },
    "priority" : 1
  }
  ...

```

Calling Custom Methods on the Server Side



This section is valid only for the Groovy DSL. See the [Dynamic Properties in the Matchers Sections](#) section for YAML examples of a similar feature.

You can define a method call that runs on the server side during the test. Such a method can be added to the class defined as `baseClassForTests` in the configuration. The following code shows an example of the contract portion of the test case:

Groovy

```
method GET()
```

Java

```
r.method(r.GET());
```

Kotlin

```
method = GET
```

The following code shows the base class portion of the test case:

```
abstract class BaseMockMvcSpec extends Specification {  
  
    def setup() {  
        RestAssuredMockMvc.standaloneSetup(new PairIdController())  
    }  
  
    void isProperCorrelationId(Integer correlationId) {  
        assert correlationId == 123456  
    }  
  
    void isEmpty(String value) {  
        assert value == null  
    }  
  
}
```



You cannot use both a `String` and `execute` to perform concatenation. For example, calling `header('Authorization', 'Bearer ' + execute('authToken()'))` leads to improper results. Instead, call `header('Authorization', execute('authToken()'))` and ensure that the `authToken()` method returns everything you need.

The type of the object read from the JSON can be one of the following, depending on the JSON path:

- `String`: If you point to a `String` value in the JSON.
- `JSONArray`: If you point to a `List` in the JSON.
- `Map`: If you point to a `Map` in the JSON.
- `Number`: If you point to `Integer`, `Double`, and other numeric type in the JSON.
- `Boolean`: If you point to a `Boolean` in the JSON.

In the request part of the contract, you can specify that the `body` should be taken from a method.



You must provide both the consumer and the producer side. The `execute` part is applied for the whole body, not for parts of it.

The following example shows how to read an object from JSON:

```
Contract contractDsl = Contract.make {
  request {
    method 'GET'
    url '/something'
    body(
      $(c('foo'), p(execute('hashCode()')))
    )
  }
  response {
    status OK()
  }
}
```

The preceding example results in calling the `hashCode()` method in the request body. It should resemble the following code:

```
// given:
MockMvcRequestSpecification request = given()
  .body(hashCode());

// when:
ResponseOptions response = given().spec(request)
  .get("/something");

// then:
assertThat(response.statusCode()).isEqualTo(200);
```

Referencing the Request from the Response

The best situation is to provide fixed values, but sometimes you need to reference a request in your response.

If you write contracts in the Groovy DSL, you can use the `fromRequest()` method, which lets you reference a bunch of elements from the HTTP request. You can use the following options:

- `fromRequest().url()`: Returns the request URL and query parameters.
- `fromRequest().query(String key)`: Returns the first query parameter with the given name.
- `fromRequest().query(String key, int index)`: Returns the nth query parameter with the given name.
- `fromRequest().path()`: Returns the full path.
- `fromRequest().path(int index)`: Returns the nth path element.

- `fromRequest().header(String key)`: Returns the first header with the given name.
- `fromRequest().header(String key, int index)`: Returns the nth header with the given name.
- `fromRequest().body()`: Returns the full request body.
- `fromRequest().body(String jsonPath)`: Returns the element from the request that matches the JSON Path.

If you use the YAML contract definition or the Java one, you have to use the [Handlebars](#) `{{{ }}` notation with custom Spring Cloud Contract functions to achieve this. In that case, you can use the following options:

- `{{{ request.url }}}`: Returns the request URL and query parameters.
- `{{{ request.query.key.[index] }}}`: Returns the nth query parameter with the given name. For example, for a key of `thing`, the first entry is `{{{ request.query.thing.[0] }}}`
- `{{{ request.path }}}`: Returns the full path.
- `{{{ request.path.[index] }}}`: Returns the nth path element. For example, the first entry is `'{{{ request.path.[0] }}}'`
- `{{{ request.headers.key }}}`: Returns the first header with the given name.
- `{{{ request.headers.key.[index] }}}`: Returns the nth header with the given name.
- `{{{ request.body }}}`: Returns the full request body.
- `{{{ jsonpath this 'your.json.path' }}}`: Returns the element from the request that matches the JSON Path. For example, for a JSON path of `$.here`, use `{{{ jsonpath this '$.here' }}}`

Consider the following contract:

Groovy

```
Contract contractDsl = Contract.make {
    request {
        method 'GET'
        url('/api/v1/xxxx') {
            queryParameters {
                parameter('foo', 'bar')
                parameter('foo', 'bar2')
            }
        }
        headers {
            header(authorization(), 'secret')
            header(authorization(), 'secret2')
        }
        body(foo: 'bar', baz: 5)
    }
    response {
        status OK()
        headers {
            header(authorization(), "foo ${fromRequest().header(authorization())}
bar")
        }
        body(
            url: fromRequest().url(),
            path: fromRequest().path(),
            pathIndex: fromRequest().path(1),
            param: fromRequest().query('foo'),
            paramIndex: fromRequest().query('foo', 1),
            authorization: fromRequest().header('Authorization'),
            authorization2: fromRequest().header('Authorization', 1),
            fullBody: fromRequest().body(),
            responseFoo: fromRequest().body('$.foo'),
            responseBaz: fromRequest().body('$.baz'),
            responseBaz2: "Bla bla ${fromRequest().body('$.foo')} bla bla",
            rawUrl: fromRequest().rawUrl(),
            rawPath: fromRequest().rawPath(),
            rawPathIndex: fromRequest().rawPath(1),
            rawParam: fromRequest().rawQuery('foo'),
            rawParamIndex: fromRequest().rawQuery('foo', 1),
            rawAuthorization: fromRequest().rawHeader('Authorization'),
            rawAuthorization2: fromRequest().rawHeader('Authorization', 1),
            rawResponseFoo: fromRequest().rawBody('$.foo'),
            rawResponseBaz: fromRequest().rawBody('$.baz'),
            rawResponseBaz2: "Bla bla ${fromRequest().rawBody('$.foo')} bla
bla"
        )
    }
}
Contract contractDsl = Contract.make {
```

```

request {
  method 'GET'
  url('/api/v1/xxxx') {
    queryParams {
      parameter('foo', 'bar')
      parameter('foo', 'bar2')
    }
  }
  headers {
    header(authorization(), 'secret')
    header(authorization(), 'secret2')
  }
  body(foo: "bar", baz: 5)
}
response {
  status OK()
  headers {
    contentType(applicationJson())
  }
  body(''
    {
      "responseFoo": "{{{ jsonPath request.body '$.foo' }}}",
      "responseBaz": {{{ jsonPath request.body '$.baz' }}}},
      "responseBaz2": "Bla bla {{{ jsonPath request.body '$.foo' }}}}
bla bla"
    }
    ''.toString()
  }
}
}

```


YAML

```
request:
  method: GET
  url: /api/v1/xxxx
  queryParameters:
    foo:
      - bar
      - bar2
  headers:
    Authorization:
      - secret
      - secret2
  body:
    foo: bar
    baz: 5
response:
  status: 200
  headers:
    Authorization: "foo {{{ request.headers.Authorization.0 }}} bar"
  body:
    url: "{{{ request.url }}"
    path: "{{{ request.path }}"
    pathIndex: "{{{ request.path.1 }}"
    param: "{{{ request.query.foo }}"
    paramIndex: "{{{ request.query.foo.1 }}"
    authorization: "{{{ request.headers.Authorization.0 }}"
    authorization2: "{{{ request.headers.Authorization.1 }}"
    fullBody: "{{{ request.body }}"
    responseFoo: "{{{ jsonpath this '$.foo' }}"
    responseBaz: "{{{ jsonpath this '$.baz' }}"
    responseBaz2: "Bla bla {{{ jsonpath this '$.foo' }}} bla bla"
```

Java

```
package contracts.beer.rest;

import java.util.function.Supplier;

import org.springframework.cloud.contract.spec.Contract;

import static
org.springframework.cloud.contract.verifier.util.ContractVerifierUtil.map;

class shouldReturnStatsForAUser implements Supplier<Contract> {

    @Override
    public Contract get() {
        return Contract.make(c -> {
            c.request(r -> {
                r.method("POST");
                r.url("/stats");
                r.body(map().entry("name", r.anyAlphaUnicode()));
                r.headers(h -> {
                    h.contentType(h.applicationJson());
                });
            });
            c.response(r -> {
                r.status(r.OK());
                r.body(map()
                    .entry("text",
                        "Dear {{{jsonPath request.body '$.name'}}} thanks
for your interested in drinking beer")
                    .entry("quantity", r.$(r.c(5), r.p(r.anyNumber()))));
                r.headers(h -> {
                    h.contentType(h.applicationJson());
                });
            });
        });
    }
}
```

Kotlin

```
package contracts.beer.rest

import org.springframework.cloud.contract.spec.ContractDsl.Companion.contract

contract {
    request {
        method = method("POST")
        url = url("/stats")
        body(mapOf(
            "name" to anyAlphaUnicode
        ))
        headers {
            contentType = APPLICATION_JSON
        }
    }
    response {
        status = OK
        body(mapOf(
            "text" to "Don't worry ${fromRequest().body("$.name")} thanks for your
interested in drinking beer",
            "quantity" to v(c(5), p(anyNumber))
        ))
        headers {
            contentType = fromRequest().header(CONTENT_TYPE)
        }
    }
}
```

Running a JUnit test generation leads to a test that resembles the following example:

```

// given:
MockMvcRequestSpecification request = given()
    .header("Authorization", "secret")
    .header("Authorization", "secret2")
    .body("{\"foo\":\"bar\",\"baz\":5}");

// when:
ResponseOptions response = given().spec(request)
    .queryParams("foo","bar")
    .queryParams("foo","bar2")
    .get("/api/v1/xxxx");

// then:
assertThat(response.statusCode()).isEqualTo(200);
assertThat(response.header("Authorization")).isEqualTo("foo secret bar");
// and:
DocumentContext parsedJson = JsonPath.parse(response.getBody().asString());
assertThatJson(parsedJson).field("[ 'fullBody' ]").isEqualTo("{\"foo\":\"bar\",\"baz\":5}");
assertThatJson(parsedJson).field("[ 'authorization' ]").isEqualTo("secret");
assertThatJson(parsedJson).field("[ 'authorization2' ]").isEqualTo("secret2");
assertThatJson(parsedJson).field("[ 'path' ]").isEqualTo("/api/v1/xxxx");
assertThatJson(parsedJson).field("[ 'param' ]").isEqualTo("bar");
assertThatJson(parsedJson).field("[ 'paramIndex' ]").isEqualTo("bar2");
assertThatJson(parsedJson).field("[ 'pathIndex' ]").isEqualTo("v1");
assertThatJson(parsedJson).field("[ 'responseBaz' ]").isEqualTo(5);
assertThatJson(parsedJson).field("[ 'responseFoo' ]").isEqualTo("bar");
assertThatJson(parsedJson).field("[ 'url' ]").isEqualTo("/api/v1/xxxx?foo=bar&foo=bar2");
assertThatJson(parsedJson).field("[ 'responseBaz2' ]").isEqualTo("Bla bla bar bla bla");

```

As you can see, elements from the request have been properly referenced in the response.

The generated WireMock stub should resemble the following example:

```

{
  "request" : {
    "urlPath" : "/api/v1/xxxx",
    "method" : "POST",
    "headers" : {
      "Authorization" : {
        "equalTo" : "secret2"
      }
    },
    "queryParameters" : {
      "foo" : {
        "equalTo" : "bar2"
      }
    },
    "bodyPatterns" : [ {
      "matchesJsonPath" : "$[?(@.['baz'] == 5)]"
    }, {
      "matchesJsonPath" : "$[?(@.['foo'] == 'bar')]"
    } ]
  },
  "response" : {
    "status" : 200,
    "body" :
    "{\\"authorization\\":\\"{{{request.headers.Authorization.[0]}}}\\",\\"path\\":\\"{{{request.path}}}\\",\\"responseBaz\\":{{{jsonpath this '$.baz'}}},\\"param\\":\\"{{{request.query.foo.[0]}}}\\",\\"pathIndex\\":\\"{{{request.path.[1]}}}\\",\\"responseBaz2\\":\\"Bla bla {{{jsonpath this '$.foo'}}} bla bla\\",\\"responseFoo\\":\\"{{{jsonpath this '$.foo'}}}\\",\\"authorization2\\":\\"{{{request.headers.Authorization.[1]}}}\\",\\"full Body\\":\\"{{{escapejsonbody}}}\\",\\"url\\":\\"{{{request.url}}}\\",\\"paramIndex\\":\\"{{{request.query.foo.[1]}}}\\""},
    "headers" : {
      "Authorization" : "{{{{request.headers.Authorization.[0]}}};foo"
    },
    "transformers" : [ "response-template" ]
  }
}

```

Sending a request such as the one presented in the **request** part of the contract results in sending the following response body:

```

{
  "url" : "/api/v1/xxxx?foo=bar&foo=bar2",
  "path" : "/api/v1/xxxx",
  "pathIndex" : "v1",
  "param" : "bar",
  "paramIndex" : "bar2",
  "authorization" : "secret",
  "authorization2" : "secret2",
  "fullBody" : "{\"foo\":\"bar\",\"baz\":5}",
  "responseFoo" : "bar",
  "responseBaz" : 5,
  "responseBaz2" : "Bla bla bar bla bla"
}

```



This feature works only with WireMock versions greater than or equal to 2.5.1. The Spring Cloud Contract Verifier uses WireMock's `response-template` response transformer. It uses Handlebars to convert the Mustache `{{{ }}}` templates into proper values. Additionally, it registers two helper functions:

- `escapejsonbody`: Escapes the request body in a format that can be embedded in JSON.
- `jsonpath`: For a given parameter, finds an object in the request body.

Dynamic Properties in the Matchers Sections

If you work with [Pact](#), the following discussion may seem familiar. Quite a few users are used to having a separation between the body and setting the dynamic parts of a contract.

You can use the `bodyMatchers` section for two reasons:

- Define the dynamic values that should end up in a stub. You can set it in the `request` part of your contract.
- Verify the result of your test. This section is present in the `response` or `outputMessage` side of the contract.

Currently, Spring Cloud Contract Verifier supports only JSON path-based matchers with the following matching possibilities:

Coded DSL

For the stubs (in tests on the consumer's side):

- `byEquality()`: The value taken from the consumer's request in the provided JSON path must be equal to the value provided in the contract.
- `byRegex(...)`: The value taken from the consumer's request in the provided JSON path must match the regex. You can also pass the type of the expected matched value (for example, `asString()`, `asLong()`, and so on).

- `byDate()`: The value taken from the consumer's request in the provided JSON path must match the regex for an ISO Date value.
- `byTimestamp()`: The value taken from the consumer's request in the provided JSON path must match the regex for an ISO DateTime value.
- `byTime()`: The value taken from the consumer's request in the provided JSON path must match the regex for an ISO Time value.

For the verification (in generated tests on the Producer's side):

- `byEquality()`: The value taken from the producer's response in the provided JSON path must be equal to the provided value in the contract.
- `byRegex(...)`: The value taken from the producer's response in the provided JSON path must match the regex.
- `byDate()`: The value taken from the producer's response in the provided JSON path must match the regex for an ISO Date value.
- `byTimestamp()`: The value taken from the producer's response in the provided JSON path must match the regex for an ISO DateTime value.
- `byTime()`: The value taken from the producer's response in the provided JSON path must match the regex for an ISO Time value.
- `byType()`: The value taken from the producer's response in the provided JSON path needs to be of the same type as the type defined in the body of the response in the contract. `byType` can take a closure, in which you can set `minOccurrence` and `maxOccurrence`. For the request side, you should use the closure to assert size of the collection. That way, you can assert the size of the flattened collection. To check the size of an unflattened collection, use a custom method with the `byCommand(...)` `testMatcher`.
- `byCommand(...)`: The value taken from the producer's response in the provided JSON path is passed as an input to the custom method that you provide. For example, `byCommand('thing($it)')` results in calling a `thing` method to which the value matching the JSON Path gets passed. The type of the object read from the JSON can be one of the following, depending on the JSON path:
 - `String`: If you point to a `String` value.
 - `JSONArray`: If you point to a `List`.
 - `Map`: If you point to a `Map`.
 - `Number`: If you point to `Integer`, `Double`, or another kind of number.
 - `Boolean`: If you point to a `Boolean`.
- `byNull()`: The value taken from the response in the provided JSON path must be null.

YAML



See the Groovy section for a detailed explanation of what the types mean.

For YAML, the structure of a matcher resembles the following example:

```
- path: $.thing1
  type: by_regex
  value: thing2
  regexType: as_string
```

Alternatively, if you want to use one of the predefined regular expressions [`only_alpha_unicode`, `number`, `any_boolean`, `ip_address`, `hostname`, `email`, `url`, `uuid`, `iso_date`, `iso_date_time`, `iso_time`, `iso_8601_with_offset`, `non_empty`, `non_blank`], you can use something similar to the following example:

```
- path: $.thing1
  type: by_regex
  predefined: only_alpha_unicode
```

The following list shows the allowed list of `type` values:

- For `stubMatchers`:
 - `by_equality`
 - `by_regex`
 - `by_date`
 - `by_timestamp`
 - `by_time`
 - `by_type`
 - Two additional fields (`minOccurrence` and `maxOccurrence`) are accepted.
- For `testMatchers`:
 - `by_equality`
 - `by_regex`
 - `by_date`
 - `by_timestamp`
 - `by_time`
 - `by_type`
 - Two additional fields (`minOccurrence` and `maxOccurrence`) are accepted.
 - `by_command`
 - `by_null`

You can also define which type the regular expression corresponds to in the `regexType` field. The following list shows the allowed regular expression types:

- `as_integer`
- `as_double`

- `as_float`
- `as_long`
- `as_short`
- `as_boolean`
- `as_string`

Consider the following example:

Groovy

```
Contract contractDsl = Contract.make {
    request {
        method 'GET'
        urlPath '/get'
        body([
            duck           : 123,
            alpha          : 'abc',
            number         : 123,
            aBoolean       : true,
            date           : '2017-01-01',
            dateTime       : '2017-01-01T01:23:45',
            time           : '01:02:34',
            valueWithoutAMatcher: 'foo',
            valueWithTypeMatch : 'string',
            key            : [
                'complex.key': 'foo'
            ]
        ])
        bodyMatchers {
            jsonPath('$.duck', byRegex("[0-9]{3}").asInteger())
            jsonPath('$.duck', byEquality())
            jsonPath('$.alpha', byRegex(onlyAlphaUnicode()).asString())
            jsonPath('$.alpha', byEquality())
            jsonPath('$.number', byRegex(number()).asInteger())
            jsonPath('$.aBoolean', byRegex(anyBoolean()).asBooleanType())
            jsonPath('$.date', byDate())
            jsonPath('$.dateTime', byTimestamp())
            jsonPath('$.time', byTime())
            jsonPath("\$.['key'].['complex.key']", byEquality())
        }
        headers {
            contentType(applicationJson())
        }
    }
    response {
        status OK()
        body([
            duck           : 123,
            alpha          : 'abc',
            number         : 123,
            positiveInteger : 1234567890,
            negativeInteger : -1234567890,
            positiveDecimalNumber: 123.4567890,
            negativeDecimalNumber: -123.4567890,
            aBoolean       : true,
            date           : '2017-01-01',
            dateTime       : '2017-01-01T01:23:45',
            time           : "01:02:34",
```

```

        valueWithoutAMatcher : 'foo',
        valueWithTypeMatch   : 'string',
        valueWithMin         : [
            1, 2, 3
        ],
        valueWithMax         : [
            1, 2, 3
        ],
        valueWithMinMax     : [
            1, 2, 3
        ],
        valueWithMinEmpty   : [],
        valueWithMaxEmpty   : [],
        key                  : [
            'complex.key': 'foo'
        ],
        nullValue           : null
    ])
    bodyMatchers {
        // asserts the jsonpath value against manual regex
        jsonPath('$.duck', byRegex("[0-9]{3}").asInteger())
        // asserts the jsonpath value against the provided value
        jsonPath('$.duck', byEquality())
        // asserts the jsonpath value against some default regex
        jsonPath('$.alpha', byRegex(onlyAlphaUnicode()).asString())
        jsonPath('$.alpha', byEquality())
        jsonPath('$.number', byRegex(number()).asInteger())
        jsonPath('$.positiveInteger', byRegex(anInteger()).asInteger())
        jsonPath('$.negativeInteger', byRegex(anInteger()).asInteger())
        jsonPath('$.positiveDecimalNumber', byRegex(aDouble()).asDouble())
        jsonPath('$.negativeDecimalNumber', byRegex(aDouble()).asDouble())
        jsonPath('$.aBoolean', byRegex(anyBoolean()).asBooleanType())
        // asserts vs inbuilt time related regex
        jsonPath('$.date', byDate())
        jsonPath('$.dateTime', byTimestamp())
        jsonPath('$.time', byTime())
        // asserts that the resulting type is the same as in response body
        jsonPath('$.valueWithTypeMatch', byType())
        jsonPath('$.valueWithMin', byType {
            // results in verification of size of array (min 1)
            minOccurrence(1)
        })
        jsonPath('$.valueWithMax', byType {
            // results in verification of size of array (max 3)
            maxOccurrence(3)
        })
        jsonPath('$.valueWithMinMax', byType {
            // results in verification of size of array (min 1 & max 3)
            minOccurrence(1)
            maxOccurrence(3)
        })
    }
}

```

```

    jsonPath('$.valueWithMinEmpty', byType {
        // results in verification of size of array (min 0)
        minOccurrence(0)
    })
    jsonPath('$.valueWithMaxEmpty', byType {
        // results in verification of size of array (max 0)
        maxOccurrence(0)
    })
    // will execute a method `assertThatValueIsANumber`
    jsonPath('$.duck', byCommand('assertThatValueIsANumber($it)'))
    jsonPath("\$.['key']['complex.key']", byEquality())
    jsonPath('$.nullValue', byNull())
}
headers {
    contentType(applicationJson())
    header('Some-Header', $(c('someValue'), p(regex('[a-zA-Z]{9}'))))
}
}
}

```

YAML

```

request:
  method: GET
  urlPath: /get/1
  headers:
    Content-Type: application/json
  cookies:
    foo: 2
    bar: 3
  queryParameters:
    limit: 10
    offset: 20
    filter: 'email'
    sort: name
    search: 55
    age: 99
    name: John.Doe
    email: 'bob@email.com'
  body:
    duck: 123
    alpha: "abc"
    number: 123
    aBoolean: true
    date: "2017-01-01"
    dateTime: "2017-01-01T01:23:45"
    time: "01:02:34"
    valueWithoutAMatcher: "foo"
    valueWithTypeMatch: "string"
  key:
    "complex.key": 'foo'

```

```
nullValue: null
valueWithMin:
  - 1
  - 2
  - 3
valueWithMax:
  - 1
  - 2
  - 3
valueWithMinMax:
  - 1
  - 2
  - 3
valueWithMinEmpty: []
valueWithMaxEmpty: []
matchers:
  url:
    regex: /get/[0-9]
    # predefined:
    # execute a method
    #command: 'equals($it)'
  queryParameters:
    - key: limit
      type: equal_to
      value: 20
    - key: offset
      type: containing
      value: 20
    - key: sort
      type: equal_to
      value: name
    - key: search
      type: not_matching
      value: '^[[0-9]]{2}$'
    - key: age
      type: not_matching
      value: '^\\w*$'
    - key: name
      type: matching
      value: 'John.*'
    - key: hello
      type: absent
  cookies:
    - key: foo
      regex: '[0-9]'
    - key: bar
      command: 'equals($it)'
  headers:
    - key: Content-Type
      regex: "application/json.*"
  body:
```

- path: \$.duck
type: by_regex
value: "[0-9]{3}"
- path: \$.duck
type: by_equality
- path: \$.alpha
type: by_regex
predefined: only_alpha_unicode
- path: \$.alpha
type: by_equality
- path: \$.number
type: by_regex
predefined: number
- path: \$.aBoolean
type: by_regex
predefined: any_boolean
- path: \$.date
type: by_date
- path: \$.dateTime
type: by_timestamp
- path: \$.time
type: by_time
- path: "\$.['key']['complex.key']"
type: by_equality
- path: \$.nullvalue
type: by_null
- path: \$.valueWithMin
type: by_type
minOccurrence: 1
- path: \$.valueWithMax
type: by_type
maxOccurrence: 3
- path: \$.valueWithMinMax
type: by_type
minOccurrence: 1
maxOccurrence: 3

response:

status: 200

cookies:

foo: 1

bar: 2

body:

duck: 123

alpha: "abc"

number: 123

aBoolean: true

date: "2017-01-01"

dateTime: "2017-01-01T01:23:45"

time: "01:02:34"

valueWithoutAMatcher: "foo"

valueWithTypeMatch: "string"

```
valueWithMin:
  - 1
  - 2
  - 3
valueWithMax:
  - 1
  - 2
  - 3
valueWithMinMax:
  - 1
  - 2
  - 3
valueWithMinEmpty: []
valueWithMaxEmpty: []
key:
  'complex.key': 'foo'
nulValue: null
matchers:
  headers:
    - key: Content-Type
      regex: "application/json.*"
  cookies:
    - key: foo
      regex: '[0-9]'
    - key: bar
      command: 'equals($it)'
  body:
    - path: $.duck
      type: by_regex
      value: "[0-9]{3}"
    - path: $.duck
      type: by_equality
    - path: $.alpha
      type: by_regex
      predefined: only_alpha_unicode
    - path: $.alpha
      type: by_equality
    - path: $.number
      type: by_regex
      predefined: number
    - path: $.aBoolean
      type: by_regex
      predefined: any_boolean
    - path: $.date
      type: by_date
    - path: $.dateTime
      type: by_timestamp
    - path: $.time
      type: by_time
    - path: $.valueWithTypeMatch
      type: by_type
```

```

- path: $.valueWithMin
  type: by_type
  minOccurrence: 1
- path: $.valueWithMax
  type: by_type
  maxOccurrence: 3
- path: $.valueWithMinMax
  type: by_type
  minOccurrence: 1
  maxOccurrence: 3
- path: $.valueWithMinEmpty
  type: by_type
  minOccurrence: 0
- path: $.valueWithMaxEmpty
  type: by_type
  maxOccurrence: 0
- path: $.duck
  type: by_command
  value: assertThatValueIsANumber($it)
- path: $.nullValue
  type: by_null
  value: null
headers:
  Content-Type: application/json

```

In the preceding example, you can see the dynamic portions of the contract in the `matchers` sections. For the request part, you can see that, for all fields but `valueWithoutAMatcher`, the values of the regular expressions that the stub should contain are explicitly set. For `valueWithoutAMatcher`, the verification takes place in the same way as without the use of matchers. In that case, the test performs an equality check.

For the response side in the `bodyMatchers` section, we define the dynamic parts in a similar manner. The only difference is that the `byType` matchers are also present. The verifier engine checks four fields to verify whether the response from the test has a value for which the JSON path matches the given field, is of the same type as the one defined in the response body, and passes the following check (based on the method being called):

- For `$.valueWithTypeMatch`, the engine checks whether the type is the same.
- For `$.valueWithMin`, the engine checks the type and asserts whether the size is greater than or equal to the minimum occurrence.
- For `$.valueWithMax`, the engine checks the type and asserts whether the size is smaller than or equal to the maximum occurrence.
- For `$.valueWithMinMax`, the engine checks the type and asserts whether the size is between the minimum and maximum occurrence.

The resulting test resembles the following example (note that an `and` section separates the autogenerated assertions and the assertion from matchers):


```

// given:
MockMvcRequestSpecification request = given()
    .header("Content-Type", "application/json")

    .body("{\"duck\":123,\"alpha\":\"abc\",\"number\":123,\"aBoolean\":true,\"date\":\"2017-01-01\",\"dateTime\":\"2017-01-01T01:23:45\",\"time\":\"01:02:34\",\"valueWithoutAMatcher\":\"foo\",\"valueWithTypeMatch\":\"string\",\"key\":{\"complex.key\":\"foo\"}}");

// when:
ResponseOptions response = given().spec(request)
    .get("/get");

// then:
assertThat(response.statusCode()).isEqualTo(200);
assertThat(response.header("Content-Type")).matches("application/json.*");
// and:
DocumentContext parsedJson = JsonPath.parse(response.getBody().asString());
assertThatJson(parsedJson).field("'valueWithoutAMatcher']").isEqualTo("foo");
// and:
assertThat(parsedJson.read("$.duck", String.class)).matches("[0-9]{3}");
assertThat(parsedJson.read("$.duck", Integer.class)).isEqualTo(123);
assertThat(parsedJson.read("$.alpha", String.class)).matches("[\\p{L}]*");
assertThat(parsedJson.read("$.alpha", String.class)).isEqualTo("abc");
assertThat(parsedJson.read("$.number", String.class)).matches("-?(\\d*\\.\\d+|\\d+)");
assertThat(parsedJson.read("$.aBoolean", String.class)).matches("(true|false)");
assertThat(parsedJson.read("$.date", String.class)).matches("(\\d\\d\\d\\d\\d)-(0[1-9]|1[012])-(0[1-9]|[12][0-9]|3[01])");
assertThat(parsedJson.read("$.dateTime", String.class)).matches("([0-9]{4})-(1[0-2]|0[1-9])-(3[01]|0[1-9]|[12][0-9])T(2[0-3]|[01][0-9]):([0-5][0-9]):([0-5][0-9])");
assertThat(parsedJson.read("$.time", String.class)).matches("(2[0-3]|[01][0-9]):([0-5][0-9]):([0-5][0-9])");
assertThat((Object)
    parsedJson.read("$.valueWithTypeMatch")).isInstanceOf(java.lang.String.class);
assertThat((Object)
    parsedJson.read("$.valueWithMin")).isInstanceOf(java.util.List.class);
assertThat((java.lang.Iterable) parsedJson.read("$.valueWithMin",
    java.util.Collection.class)).as("$.valueWithMin").hasSizeGreaterThanOrEqualTo(1);
assertThat((Object)
    parsedJson.read("$.valueWithMax")).isInstanceOf(java.util.List.class);
assertThat((java.lang.Iterable) parsedJson.read("$.valueWithMax",
    java.util.Collection.class)).as("$.valueWithMax").hasSizeLessThanOrEqualTo(3);
assertThat((Object)
    parsedJson.read("$.valueWithMinMax")).isInstanceOf(java.util.List.class);
assertThat((java.lang.Iterable) parsedJson.read("$.valueWithMinMax",
    java.util.Collection.class)).as("$.valueWithMinMax").hasSizeBetween(1, 3);
assertThat((Object)
    parsedJson.read("$.valueWithMinEmpty")).isInstanceOf(java.util.List.class);
assertThat((java.lang.Iterable) parsedJson.read("$.valueWithMinEmpty",
    java.util.Collection.class)).as("$.valueWithMinEmpty").hasSizeGreaterThanOrEqualTo(0);

```

```

assertThat((Object)
parsedJson.read("$.valueWithMaxEmpty")).assertInstanceOf(java.util.List.class);
assertThat((java.lang.Iterable) parsedJson.read("$.valueWithMaxEmpty",
java.util.Collection.class)).as("$.valueWithMaxEmpty").hasSizeLessThanOrEqualTo(0);
assertThatValueIsANumber(parsedJson.read("$.duck"));
assertThat(parsedJson.read("$.['key'].['complex.key']",
String.class)).isEqualTo("foo");

```



Notice that, for the `byCommand` method, the example calls the `assertThatValueIsANumber`. This method must be defined in the test base class or be statically imported to your tests. Notice that the `byCommand` call was converted to `assertThatValueIsANumber(parsedJson.read("$.duck"))`; That means that the engine took the method name and passed the proper JSON path as a parameter to it.

The resulting WireMock stub is in the following example:

```

'''
{
  "request" : {
    "urlPath" : "/get",
    "method" : "POST",
    "headers" : {
      "Content-Type" : {
        "matches" : "application/json.*"
      }
    },
    "bodyPatterns" : [ {
      "matchesJsonPath" : "$.['list'].['some'].['nested'][?(@.['anothervalue'] == 4)]"
    }, {
      "matchesJsonPath" : "$[?(@.['valueWithoutAMatcher'] == 'foo')]"
    }, {
      "matchesJsonPath" : "$[?(@.['valueWithTypeMatch'] == 'string')]"
    }, {
      "matchesJsonPath" : "$.['list'].['someother'].['nested'][?(@.['json'] == 'with
value')]"
    }, {
      "matchesJsonPath" : "$.['list'].['someother'].['nested'][?(@.['anothervalue'] ==
4)]"
    }, {
      "matchesJsonPath" : "$[?(@.duck =~ /[0-9]{3})/]"
    }, {
      "matchesJsonPath" : "$[?(@.duck == 123)]"
    }, {
      "matchesJsonPath" : "$[?(@.alpha =~ /([\\p{L}]+)/]"
    }, {
      "matchesJsonPath" : "$[?(@.alpha == 'abc')]"
    }, {
      "matchesJsonPath" : "$[?(@.number =~ /(-?(\\d*\\.\\d+|\\d+))/]"
    }, {

```

```

    "matchesJsonPath" : "$[?(@.aBoolean =~ /((true|false))/)]"
  }, {
    "matchesJsonPath" : "$[?(@.date =~ /((\\d\\d\\d\\d\\d\\d\\d\\d\\d\\d)-(0[1-9]|1[012])-(0[1-9]|1[12][0-9]|3[01]))/)]"
  }, {
    "matchesJsonPath" : "$[?(@.dateTime =~ /((([0-9]{4})-(1[0-2]|0[1-9])-(3[01]|0[1-9]|1[12][0-9])T(2[0-3]|[01][0-9]):([0-5][0-9]):([0-5][0-9]))/)]"
  }, {
    "matchesJsonPath" : "$[?(@.time =~ /((2[0-3]|[01][0-9]):([0-5][0-9]):([0-5][0-9]))/)]"
  }, {
    "matchesJsonPath" : "$.list.some.nested[?(@.json =~ /(.*)/)]"
  }, {
    "matchesJsonPath" : "$[?(@.valueWithMin.size() >= 1)]"
  }, {
    "matchesJsonPath" : "$[?(@.valueWithMax.size() <= 3)]"
  }, {
    "matchesJsonPath" : "$[?(@.valueWithMinMax.size() >= 1 && @.valueWithMinMax.size() <= 3)]"
  }, {
    "matchesJsonPath" : "$[?(@.valueWithOccurrence.size() >= 4 && @.valueWithOccurrence.size() <= 4)]"
  } ]
},
"response" : {
  "status" : 200,
  "body" :
"{\\\"duck\\\":123,\\\"alpha\\\":\\\"abc\\\",\\\"number\\\":123,\\\"aBoolean\\\":true,\\\"date\\\":\\\"2017-01-01\\\",\\\"dateTime\\\":\\\"2017-01-01T01:23:45\\\",\\\"time\\\":\\\"01:02:34\\\",\\\"valueWithoutAMatcher\\\":\\\"foo\\\",\\\"valueWithTypeMatch\\\":\\\"string\\\",\\\"valueWithMin\\\":[1,2,3],\\\"valueWithMax\\\":[1,2,3],\\\"valueWithMinMax\\\":[1,2,3],\\\"valueWithOccurrence\\\":[1,2,3,4]}",
  "headers" : {
    "Content-Type" : "application/json"
  },
  "transformers" : [ "response-template", "spring-cloud-contract" ]
}
}
...

```



If you use a **matcher**, the part of the request and response that the **matcher** addresses with the JSON Path gets removed from the assertion. In the case of verifying a collection, you must create matchers for **all** the elements of the collection.

Consider the following example:

```

Contract.make {
  request {
    method 'GET'
    url("/foo")
  }
  response {
    status OK()
    body(events: [[
      [
        operation      : 'EXPORT',
        eventId        : '16f1ed75-0bcc-4f0d-a04d-
3121798faf99',
        status         : 'OK'
      ], [
        operation      : 'INPUT_PROCESSING',
        eventId        : '3bb4ac82-6652-462f-b6d1-
75e424a0024a',
        status         : 'OK'
      ]
    ]
    )
    bodyMatchers {
      jsonPath('$.events[0].operation', byRegex('.+'))
      jsonPath('$.events[0].eventId', byRegex('^([a-fA-F0-9]{8}-[a-fA-F0-
9]{4}-[a-fA-F0-9]{4}-[a-fA-F0-9]{4}-[a-fA-F0-9]{12})$'))
      jsonPath('$.events[0].status', byRegex('.+'))
    }
  }
}

```

The preceding code leads to creating the following test (the code block shows only the assertion section):

```

and:
    DocumentContext parsedJson = JsonPath.parse(response.body.asString())

assertThatJson(parsedJson).array(["'events'"]).contains(["'eventId'"]).isEqualTo("
16f1ed75-0bcc-4f0d-a04d-3121798faf99")

assertThatJson(parsedJson).array(["'events'"]).contains(["'operation'"]).isEqualTo(
"EXPORT")

assertThatJson(parsedJson).array(["'events'"]).contains(["'operation'"]).isEqualTo(
"INPUT_PROCESSING")

assertThatJson(parsedJson).array(["'events'"]).contains(["'eventId'"]).isEqualTo("
3bb4ac82-6652-462f-b6d1-75e424a0024a")

assertThatJson(parsedJson).array(["'events'"]).contains(["'status'"]).isEqualTo("0
K")
and:
    assertThat(parsedJson.read("\$.events[0].operation",
String.class)).matches(".*")
    assertThat(parsedJson.read("\$.events[0].eventId",
String.class)).matches("^[a-fA-F0-9]{8}-[a-fA-F0-9]{4}-[a-fA-F0-9]{4}-[a-fA-F0-
9]{4}-[a-fA-F0-9]{12}$")
    assertThat(parsedJson.read("\$.events[0].status", String.class)).matches(".*")

```

Note that the assertion is malformed. Only the first element of the array got asserted. To fix this, apply the assertion to the whole `$.events` collection and assert it with the `byCommand(...)` method.

3.2.5. Asynchronous Support

If you use asynchronous communication on the server side (your controllers are returning `Callable`, `DeferredResult`, and so on), then, inside your contract, you must provide an `async()` method in the `response` section. The following code shows an example:

Groovy

```
org.springframework.cloud.contract.spec.Contract.make {
    request {
        method GET()
        url '/get'
    }
    response {
        status OK()
        body 'Passed'
        async()
    }
}
```

YAML

```
response:
  async: true
```

Java

```
class contract implements Supplier<Collection<Contract>> {

    @Override
    public Collection<Contract> get() {
        return Collections.singletonList(Contract.make(c -> {
            c.request(r -> {
                // ...
            });
            c.response(r -> {
                r.async();
                // ...
            });
        }));
    }
}
```

Kotlin

```
import org.springframework.cloud.contract.spec.ContractDsl.Companion.contract

contract {
    request {
        // ...
    }
    response {
        async = true
        // ...
    }
}
```

You can also use the `fixedDelayMilliseconds` method or property to add delay to your stubs. The following example shows how to do so:

Groovy

```
org.springframework.cloud.contract.spec.Contract.make {
    request {
        method GET()
        url '/get'
    }
    response {
        status 200
        body 'Passed'
        fixedDelayMilliseconds 1000
    }
}
```

YAML

```
response:
  fixedDelayMilliseconds: 1000
```

Java

```
class contract implements Supplier<Collection<Contract>> {

    @Override
    public Collection<Contract> get() {
        return Collections.singletonList(Contract.make(c -> {
            c.request(r -> {
                // ...
            });
            c.response(r -> {
                r.fixedDelayMilliseconds(1000);
                // ...
            });
        }));
    }
}
```


Kotlin

```
import org.springframework.cloud.contract.spec.ContractDsl.Companion.contract

contract {
    request {
        // ...
    }
    response {
        delay = fixedMilliseconds(1000)
        // ...
    }
}
```

3.2.6. XML Support for HTTP

For HTTP contracts, we also support using XML in the request and response body. The XML body has to be passed within the `body` element as a `String` or `GString`. Also, body matchers can be provided for both the request and the response. In place of the `jsonPath(...)` method, the `org.springframework.cloud.contract.spec.internal.BodyMatchers.xpath` method should be used, with the desired `xPath` provided as the first argument and the appropriate `MatchingType` as the second argument. All the body matchers apart from `byType()` are supported.

The following example shows a Groovy DSL contract with XML in the response body:

```

Contract.make {
    request {
        method GET()
        urlPath '/get'
        headers {
            contentType(applicationXml())
        }
    }
    response {
        status(OK())
        headers {
            contentType(applicationXml())
        }
        body ""
    }
}

<test>
<duck type='xtype'>123</duck>
<alpha>abc</alpha>
<list>
<elem>abc</elem>
<elem>def</elem>
<elem>ghi</elem>
</list>
<number>123</number>
<aBoolean>true</aBoolean>
<date>2017-01-01</date>
<dateTime>2017-01-01T01:23:45</dateTime>
<time>01:02:34</time>
<valueWithoutAMatcher>foo</valueWithoutAMatcher>
<key><complex>foo</complex></key>
</test>""

    bodyMatchers {
        XPath('/test/duck/text()', byRegexp("[0-9]{3}"))
        XPath('/test/duck/text()',
byCommand('equals($it)'))
        XPath('/test/duck/xxx', byNull())
        XPath('/test/duck/text()', byEquality())
        XPath('/test/alpha/text()',
byRegexp(onlyAlphaUnicode()))
        XPath('/test/alpha/text()', byEquality())
        XPath('/test/number/text()', byRegexp(number()))
        XPath('/test/date/text()', byDate())
        XPath('/test/dateTime/text()', byTimestamp())
        XPath('/test/time/text()', byTime())
        XPath('/test/*/complex/text()', byEquality())
        XPath('/test/duck/@type', byEquality())
    }
}

```

```

Contract.make {
  request {
    method GET()
    urlPath '/get'
    headers {
      contentType(applicationXml())
    }
  }
  response {
    status(OK())
    headers {
      contentType(applicationXml())
    }
    body ""
  }
}

<ns1:test xmlns:ns1="http://demo.com/testns">
  <ns1:header>
    <duck-bucket type='bigbucket'>
      <duck>duck5150</duck>
    </duck-bucket>
  </ns1:header>
</ns1:test>
""

bodyMatchers {
  xpath('/test/duck/text()', byRegex("[0-9]{3}"))
  xpath('/test/duck/text()',
byCommand('equals($it)'))

  xpath('/test/duck/xxx', byNull())
  xpath('/test/duck/text()', byEquality())
  xpath('/test/alpha/text()',
byRegex(onlyAlphaUnicode()))

  xpath('/test/alpha/text()', byEquality())
  xpath('/test/number/text()', byRegex(number()))
  xpath('/test/date/text()', byDate())
  xpath('/test/dateTime/text()', byTimestamp())
  xpath('/test/time/text()', byTime())
  xpath('/test/duck/@type', byEquality())
}
}

Contract.make {
  request {
    method GET()
    urlPath '/get'
    headers {
      contentType(applicationXml())
    }
  }
  response {
    status(OK())
    headers {
      contentType(applicationXml())
    }
  }
}

```

```

    }
    body ""
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
  <SOAP-ENV:Header>
    <RsHeader xmlns="http://schemas.xmlsoap.org/soap/custom">
      <MsgSeqId>1234</MsgSeqId>
    </RsHeader>
  </SOAP-ENV:Header>
</SOAP-ENV:Envelope>
"""

    bodyMatchers {
      xPath('//*[local-name()=\'RsHeader\' and
namespace-uri()=\'http://schemas.xmlsoap.org/soap/custom\' ]/*[local-
name()=\'MsgSeqId\' ]/text()', byEquality())
    }
  }
}
Contract.make {
  request {
    method GET()
    urlPath '/get'
    headers {
      contentType(applicationXml())
    }
  }
  response {
    status(OK())
    headers {
      contentType(applicationXml())
    }
    body ""
  }
}
<ns1:customer xmlns:ns1="http://demo.com/customer"
xmlns:addr="http://demo.com/address">
  <email>customer@test.com</email>
  <contact-info xmlns="http://demo.com/contact-info">
    <name>Krombopulous</name>
    <address>
      <addr:gps>
        <lat>51</lat>
        <addr:lon>50</addr:lon>
      </addr:gps>
    </address>
  </contact-info>
</ns1:customer>
"""
}
}

```

YAML

```
request:
```

```
method: GET
url: /getyamlResponse
headers:
  Content-Type: application/xml
body: |
  <test>
  <duck type='xtype'>123</duck>
  <alpha>abc</alpha>
  <list>
  <elem>abc</elem>
  <elem>def</elem>
  <elem>ghi</elem>
  </list>
  <number>123</number>
  <aBoolean>>true</aBoolean>
  <date>2017-01-01</date>
  <dateTime>2017-01-01T01:23:45</dateTime>
  <time>01:02:34</time>
  <valueWithoutAMatcher>foo</valueWithoutAMatcher>
  <valueWithTypeMatch>string</valueWithTypeMatch>
  <key><complex>foo</complex></key>
  </test>
matchers:
  body:
    - path: /test/duck/text()
      type: by_regex
      value: "[0-9]{10}"
    - path: /test/duck/text()
      type: by_equality
    - path: /test/time/text()
      type: by_time
response:
  status: 200
  headers:
    Content-Type: application/xml
  body: |
    <test>
    <duck type='xtype'>123</duck>
    <alpha>abc</alpha>
    <list>
    <elem>abc</elem>
    <elem>def</elem>
    <elem>ghi</elem>
    </list>
    <number>123</number>
    <aBoolean>>true</aBoolean>
    <date>2017-01-01</date>
    <dateTime>2017-01-01T01:23:45</dateTime>
    <time>01:02:34</time>
    <valueWithoutAMatcher>foo</valueWithoutAMatcher>
    <valueWithTypeMatch>string</valueWithTypeMatch>
```

```

<key><complex>foo</complex></key>
</test>
matchers:
  body:
    - path: /test/duck/text()
      type: by_regex
      value: "[0-9]{10}"
    - path: /test/duck/text()
      type: by_command
      value: "test($it)"
    - path: /test/duck/xxx
      type: by_null
    - path: /test/duck/text()
      type: by_equality
    - path: /test/time/text()
      type: by_time

```

Java

```

import java.util.function.Supplier;

import org.springframework.cloud.contract.spec.Contract;

class contract_xml implements Supplier<Contract> {

    @Override
    public Contract get() {
        return Contract.make(c -> {
            c.request(r -> {
                r.method(r.GET());
                r.urlPath("/get");
                r.headers(h -> {
                    h.contentType(h.applicationXml());
                });
            });
            c.response(r -> {
                r.status(r.OK());
                r.headers(h -> {
                    h.contentType(h.applicationXml());
                });
                r.body("<test>\n" + "<duck type='xtype'>123</duck>\n" +
"<alpha>abc</alpha>\n" + "<list>\n"
                    + "<elem>abc</elem>\n" + "<elem>def</elem>\n" +
"<elem>ghi</elem>\n" + "</list>\n"
                    + "<number>123</number>\n" + "<aBoolean>true</aBoolean>\n"
+ "<date>2017-01-01</date>\n"
                    + "<dateTime>2017-01-01T01:23:45</dateTime>\n" +
"<time>01:02:34</time>\n"
                    + "<valueWithoutAMatcher>foo</valueWithoutAMatcher>\n" +
"<key><complex>foo</complex></key>\n"
                    + "</test>");
            });
        });
    }
}

```

```
    r.bodyMatchers(m -> {
      m.xpath("/test/duck/text()", m.byRegex("[0-9]{3}"));
      m.xpath("/test/duck/text()", m.byCommand("equals($it)"));
      m.xpath("/test/duck/xxx", m.byNull());
      m.xpath("/test/duck/text()", m.byEquality());
      m.xpath("/test/alpha/text()",
m.byRegex(r.onlyAlphaUnicode()));
      m.xpath("/test/alpha/text()", m.byEquality());
      m.xpath("/test/number/text()", m.byRegex(r.number()));
      m.xpath("/test/date/text()", m.byDate());
      m.xpath("/test/dateTime/text()", m.byTimestamp());
      m.xpath("/test/time/text()", m.byTime());
      m.xpath("/test/*/complex/text()", m.byEquality());
      m.xpath("/test/duck/@type", m.byEquality());
    });
  });
};
}
```

```

import org.springframework.cloud.contract.spec.ContractDsl.Companion.contract

contract {
    request {
        method = GET
        urlPath = path("/get")
        headers {
            contentType = APPLICATION_XML
        }
    }
    response {
        status = OK
        headers {
            contentType =APPLICATION_XML
        }
        body = body("<test>\n" + "<duck type='xtype'>123</duck>\n"
            + "<alpha>abc</alpha>\n" + "<list>\n" + "<elem>abc</elem>\n"
            + "<elem>def</elem>\n" + "<elem>ghi</elem>\n" + "</list>\n"
            + "<number>123</number>\n" + "<aBoolean>true</aBoolean>\n"
            + "<date>2017-01-01</date>\n"
            + "<dateTime>2017-01-01T01:23:45</dateTime>\n"
            + "<time>01:02:34</time>\n"
            + "<valueWithoutAMatcher>foo</valueWithoutAMatcher>\n"
            + "<key><complex>foo</complex></key>\n" + "</test>")
        bodyMatchers {
            xPath("/test/duck/text()", byRegex("[0-9]{3}"))
            xPath("/test/duck/text()", byCommand("equals(\$it)"))
            xPath("/test/duck/xxx", byNull)
            xPath("/test/duck/text()", byEquality)
            xPath("/test/alpha/text()", byRegex(onlyAlphaUnicode))
            xPath("/test/alpha/text()", byEquality)
            xPath("/test/number/text()", byRegex(number))
            xPath("/test/date/text()", byDate)
            xPath("/test/dateTime/text()", byTimestamp)
            xPath("/test/time/text()", byTime)
            xPath("/test/*/complex/text()", byEquality)
            xPath("/test/duck/@type", byEquality)
        }
    }
}

```

The following example shows an automatically generated test for XML in the response body:


```

@Test
public void validate_xmlMatches() throws Exception {
    // given:
    MockMvcRequestSpecification request = given()
        .header("Content-Type", "application/xml");

    // when:
    ResponseOptions response = given().spec(request).get("/get");

    // then:
    assertThat(response.statusCode()).isEqualTo(200);
    // and:
    DocumentBuilder documentBuilder = DocumentBuilderFactory.newInstance()
        .newDocumentBuilder();
    Document parsedXml = documentBuilder.parse(new InputSource(
        new StringReader(response.getBody().asString())));
    // and:
    assertThat(valueFromXPath(parsedXml,
"/test/list/elm/text()")).isEqualTo("abc");

    assertThat(valueFromXPath(parsedXml, "/test/list/elm[2]/text()")).isEqualTo("def")
;
    assertThat(valueFromXPath(parsedXml, "/test/duck/text()").matches("[0-9]{3}"));
    assertThat(nodeFromXPath(parsedXml, "/test/duck/xxx")).isNull();
    assertThat(valueFromXPath(parsedXml,
"/test/alpha/text()").matches("[\\p{L}]*"));
    assertThat(valueFromXPath(parsedXml,
"/test/*/complex/text()").isEqualTo("foo"));
    assertThat(valueFromXPath(parsedXml, "/test/duck/@type")).isEqualTo("xtype");
}

```

XML Support for Namespaces

Namespaced XML is supported. However, any XPath expressions used to select namespaced content must be updated.

Consider the following explicitly namespaced XML document:

```

<ns1:customer xmlns:ns1="http://demo.com/customer">
    <email>customer@test.com</email>
</ns1:customer>

```

The XPath expression to select the email address is: `/ns1:customer/email/text()`.



Beware as the unqualified expression (`/customer/email/text()`) results in "".

For content that uses an unqualified namespace, the expression is more verbose. Consider the following XML document that uses an unqualified namespace:

```
<customer xmlns="http://demo.com/customer">
  <email>customer@test.com</email>
</customer>
```

The XPath expression to select the email address is

```
*/[local-name()='customer' and namespace-uri()='http://demo.com/customer']/*[local-name()='email']/text()
```



Beware, as the unqualified expressions (`/customer/email/text()` or `*/[local-name()='customer' and namespace-uri()='http://demo.com/customer']/email/text()`) result in `""`. Even the child elements have to be referenced with the `local-name` syntax.

General Namespaced Node Expression Syntax

- Node using qualified namespace:

```
/<node-name>
```

- Node using and defining an unqualified namespace:

```
/*[local-name=()='<node-name>' and namespace-uri=()='<namespace-uri>']
```



In some cases, you can omit the `namespace_uri` portion, but doing so may lead to ambiguity.

- Node using an unqualified namespace (one of its ancestor's defines the `xmlns` attribute):

```
/*[local-name=()='<node-name>']
```

3.2.7. Multiple Contracts in One File

You can define multiple contracts in one file. Such a contract might resemble the following example:

Groovy

```
import org.springframework.cloud.contract.spec.Contract

[
  Contract.make {
    name("should post a user")
    request {
      method 'POST'
      url('/users/1')
    }
    response {
      status OK()
    }
  },
  Contract.make {
    request {
      method 'POST'
      url('/users/2')
    }
    response {
      status OK()
    }
  }
]
```

YAML

```
---
name: should post a user
request:
  method: POST
  url: /users/1
response:
  status: 200
---
request:
  method: POST
  url: /users/2
response:
  status: 200
---
request:
  method: POST
  url: /users/3
response:
  status: 200
```

Java

```
class contract implements Supplier<Collection<Contract>> {

    @Override
    public Collection<Contract> get() {
        return Arrays.asList(
            Contract.make(c -> {
                c.name("should post a user");
                // ...
            }), Contract.make(c -> {
                // ...
            }), Contract.make(c -> {
                // ...
            })
        );
    }
}
```

Kotlin

```
import org.springframework.cloud.contract.spec.ContractDsl.Companion.contract

arrayOf(
    contract {
        name("should post a user")
        // ...
    },
    contract {
        // ...
    },
    contract {
        // ...
    }
}
```

In the preceding example, one contract has the `name` field and the other does not. This leads to generation of two tests that look like the following:

```

package org.springframework.cloud.contract.verifier.tests.com.hello;

import com.example.TestBase;
import com.jayway.jsonpath.DocumentContext;
import com.jayway.jsonpath.JsonPath;
import
com.jayway.restassured.module.mockmvc.specification.MockMvcRequestSpecification;
import com.jayway.restassured.response.ResponseOptions;
import org.junit.Test;

import static com.jayway.restassured.module.mockmvc.RestAssuredMockMvc.*;
import static com.toomuchcoding.jsonassert.JsonAssertion.assertThatJson;
import static org.assertj.core.api.Assertions.assertThat;

public class V1Test extends TestBase {

    @Test
    public void validate_should_post_a_user() throws Exception {
        // given:
        MockMvcRequestSpecification request = given();

        // when:
        ResponseOptions response = given().spec(request)
            .post("/users/1");

        // then:
        assertThat(response.statusCode()).isEqualTo(200);
    }

    @Test
    public void validate_withList_1() throws Exception {
        // given:
        MockMvcRequestSpecification request = given();

        // when:
        ResponseOptions response = given().spec(request)
            .post("/users/2");

        // then:
        assertThat(response.statusCode()).isEqualTo(200);
    }
}

```

Notice that, for the contract that has the `name` field, the generated test method is named `validate_should_post_a_user`. The one that does not have the `name` field is called `validate_withList_1`. It corresponds to the name of the file `WithList.groovy` and the index of the contract in the list.

The generated stubs are shown in the following example:

```
should post a user.json
1_WithList.json
```

The first file got the `name` parameter from the contract. The second got the name of the contract file (`WithList.groovy`) prefixed with the index (in this case, the contract had an index of `1` in the list of contracts in the file).



It is much better to name your contracts, because doing so makes your tests far more meaningful.

3.2.8. Stateful Contracts

Stateful contracts (also known as scenarios) are contract definitions that should be read in order. This might be useful in the following situations:

- You want to invoke the contract in a precisely defined order, since you use Spring Cloud Contract to test your stateful application.



We really discourage you from doing that, since contract tests should be stateless.

- You want the same endpoint to return different results for the same request.

To create stateful contracts (or scenarios), you need to use the proper naming convention while creating your contracts. The convention requires including an order number followed by an underscore. This works regardless of whether you work with YAML or Groovy. The following listing shows an example:

```
my_contracts_dir\  
  scenario1\  
    1_login.groovy  
    2_showCart.groovy  
    3_logout.groovy
```

Such a tree causes Spring Cloud Contract Verifier to generate WireMock's scenario with a name of `scenario1` and the three following steps:

1. `login`, marked as `Started` pointing to...
2. `showCart`, marked as `Step1` pointing to...
3. `logout`, marked as `Step2` (which closes the scenario).

You can find more details about WireMock scenarios at <https://wiremock.org/docs/stateful->

[behaviour/](#).

3.3. Integrations

3.3.1. JAX-RS

The Spring Cloud Contract supports the JAX-RS 2 Client API. The base class needs to define `protected WebTarget webTarget` and server initialization. The only option for testing JAX-RS API is to start a web server. Also, a request with a body needs to have a content type be set. Otherwise, the default of `application/octet-stream` gets used.

To use JAX-RS mode, use the following setting:

```
testMode = 'JAXRSCLIENT'
```

The following example shows a generated test API:

```

package com.example;

import com.jayway.jsonpath.DocumentContext;
import com.jayway.jsonpath.JsonPath;
import org.junit.Test;
import org.junit.Rule;
import javax.ws.rs.client.Entity;
import javax.ws.rs.core.Response;

import static
org.springframework.cloud.contract.verifier.assertion.SpringCloudContractAssertion
s.assertThat;
import static
org.springframework.cloud.contract.verifier.util.ContractVerifierUtil.*;
import static com.toomuchcoding.jsonassert.JsonAssertion.assertThatJson;
import static javax.ws.rs.client.Entity.*;

@SuppressWarnings("rawtypes")
public class FooTest {
    WebTarget webTarget;

    @Test
    public void validate_() throws Exception {

        // when:
        Response response = webTarget
            .path("/users")
            .queryParams("limit", "10")
            .queryParams("offset", "20")
            .queryParams("filter", "email")
            .queryParams("sort", "name")
            .queryParams("search", "55")
            .queryParams("age", "99")
            .queryParams("name", "Denis.Stepanov")
            .queryParams("email", "bob@email.com")
            .request()
            .build("GET")
            .invoke();
        String responseAsString = response.readEntity(String.class);

        // then:
        assertThat(response.getStatus()).isEqualTo(200);

        // and:
        DocumentContext parsedJson = JsonPath.parse(responseAsString);
        assertThatJson(parsedJson).field("[ 'property1' ]").isEqualTo("a");
    }
}

```


3.3.2. WebFlux with WebTestClient

You can work with WebFlux by using WebTestClient. The following listing shows how to configure WebTestClient as the test mode:

Maven

```
<plugin>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-contract-maven-plugin</artifactId>
  <version>${spring-cloud-contract.version}</version>
  <extensions>true</extensions>
  <configuration>
    <testMode>WEBTESTCLIENT</testMode>
  </configuration>
</plugin>
```

Gradle

```
contracts {
    testMode = 'WEBTESTCLIENT'
}
```

The following example shows how to set up a WebTestClient base class and RestAssured for WebFlux:

```
import io.restassured.module.webtestclient.RestAssuredWebTestClient;
import org.junit.Before;

public abstract class BeerRestBase {

    @Before
    public void setup() {
        RestAssuredWebTestClient.standaloneSetup(
            new ProducerController(personToCheck -> personToCheck.age >= 20));
    }
}
```



The `WebTestClient` mode is faster than the `EXPLICIT` mode.

3.3.3. WebFlux with Explicit Mode

You can also use WebFlux with the explicit mode in your generated tests to work with WebFlux. The following example shows how to configure using explicit mode:

Maven

```
<plugin>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-contract-maven-plugin</artifactId>
  <version>${spring-cloud-contract.version}</version>
  <extensions>>true</extensions>
  <configuration>
    <testMode>EXPLICIT</testMode>
  </configuration>
</plugin>
```

Gradle

```
contracts {
    testMode = 'EXPLICIT'
}
```

The following example shows how to set up a base class and RestAssured for Web Flux:

```
@SpringBootTest(classes = BeerRestBase.Config.class,
    webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT,
    properties = "server.port=0")
public abstract class BeerRestBase {

    // your tests go here

    // in this config class you define all controllers and mocked services
    @Configuration
    @EnableAutoConfiguration
    static class Config {

        @Bean
        PersonCheckingService personCheckingService() {
            return personToCheck -> personToCheck.age >= 20;
        }

        @Bean
        ProducerController producerController() {
            return new ProducerController(personCheckingService());
        }
    }
}
```

3.3.4. Custom Mode



This mode is experimental and can change in the future.

The Spring Cloud Contract lets you provide your own, custom, implementation of the `org.springframework.cloud.contract.verifier.http.HttpVerifier`. That way, you can use any client you want to send and receive a request. The default implementation in Spring Cloud Contract is `OkHttpHttpVerifier` and it uses OkHttp3 http client.

To get started, set `testMode` to `CUSTOM`:

```
testMode = 'CUSTOM'
```

The following example shows a generated test:

```

package com.example.beer;

import com.example.BeerRestBase;
import javax.inject.Inject;
import org.springframework.cloud.contract.verifier.http.HttpVerifier;
import org.springframework.cloud.contract.verifier.http.Request;
import org.springframework.cloud.contract.verifier.http.Response;
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.extension.ExtendWith;

import static
org.springframework.cloud.contract.verifier.assertion.SpringCloudContractAssertion
s.assertThat;
import static
org.springframework.cloud.contract.verifier.util.ContractVerifierUtil.*;
import static org.springframework.cloud.contract.verifier.http.Request.given;

@SuppressWarnings("rawtypes")
public class RestTest extends BeerRestBase {
    @Inject HttpVerifier httpVerifier;

    @Test
    public void validate_shouldGrantABeerIfOldEnough() throws Exception {
        // given:
        Request request = given()
            .post("/beer.BeerService/check")
            .scheme("HTTP")
            .protocol("h2_prior_knowledge")
            .header("Content-Type", "application/grpc")
            .header("te", "trailers")
            .body(fileToBytes(this,
"shouldGrantABeerIfOldEnough_request_PersonToCheck_old_enough.bin"))
            .build();

        // when:
        Response response = httpVerifier.exchange(request);

        // then:
        assertThat(response.statusCode()).isEqualTo(200);
        assertThat(response.header("Content-
Type")).matches("application/grpc.*");
        assertThat(response.header("grpc-encoding")).isEqualTo("identity");
        assertThat(response.header("grpc-accept-encoding")).isEqualTo("gzip");

        // and:

assertThat(response.getBody().asByteArray()).isEqualTo(fileToBytes(this,

```

```
"shouldGrantABeerIfOldEnough_response_Response_old_enough.bin"));
    }
}
```

The following example shows a corresponding base class:

```
@SpringBootTest(classes = BeerRestBase.Config.class,
    webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT)
public abstract class BeerRestBase {

    @Configuration
    @EnableAutoConfiguration
    static class Config {

        @Bean
        ProducerController producerController(PersonCheckingService
personCheckingService) {
            return new ProducerController(personCheckingService);
        }

        @Bean
        PersonCheckingService testPersonCheckingService() {
            return argument -> argument.getAge() >= 20;
        }

        @Bean
        HttpVerifier httpOkVerifier(@LocalServerPort int port) {
            return new OkHttpHttpVerifier("localhost:" + port);
        }
    }
}
```

3.3.5. Working with Context Paths

Spring Cloud Contract supports context paths.

The only change needed to fully support context paths is the switch on the producer side. Also, the autogenerated tests must use explicit mode. The consumer side remains untouched. In order for the generated test to pass, you must use explicit mode. The following example shows how to set the test mode to **EXPLICIT**:



Maven

```
<plugin>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-contract-maven-plugin</artifactId>
  <version>${spring-cloud-contract.version}</version>
  <extensions>>true</extensions>
  <configuration>
    <testMode>EXPLICIT</testMode>
  </configuration>
</plugin>
```

Gradle

```
contracts {
    testMode = 'EXPLICIT'
}
```

That way, you generate a test that does not use MockMvc. It means that you generate real requests and you need to set up your generated test's base class to work on a real socket.

Consider the following contract:

```
org.springframework.cloud.contract.spec.Contract.make {
    request {
        method 'GET'
        url '/my-context-path/url'
    }
    response {
        status OK()
    }
}
```

The following example shows how to set up a base class and RestAssured:

```

import io.restassured.RestAssured;
import org.junit.Before;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.boot.test.web.server.LocalServerPort;

@SpringBootTest(classes = ContextPathTestingBaseClass.class, webEnvironment =
SpringBootTest.WebEnvironment.RANDOM_PORT)
class ContextPathTestingBaseClass {

    @LocalServerPort int port;

    @Before
    public void setup() {
        RestAssured.baseURI = "http://localhost";
        RestAssured.port = this.port;
    }
}

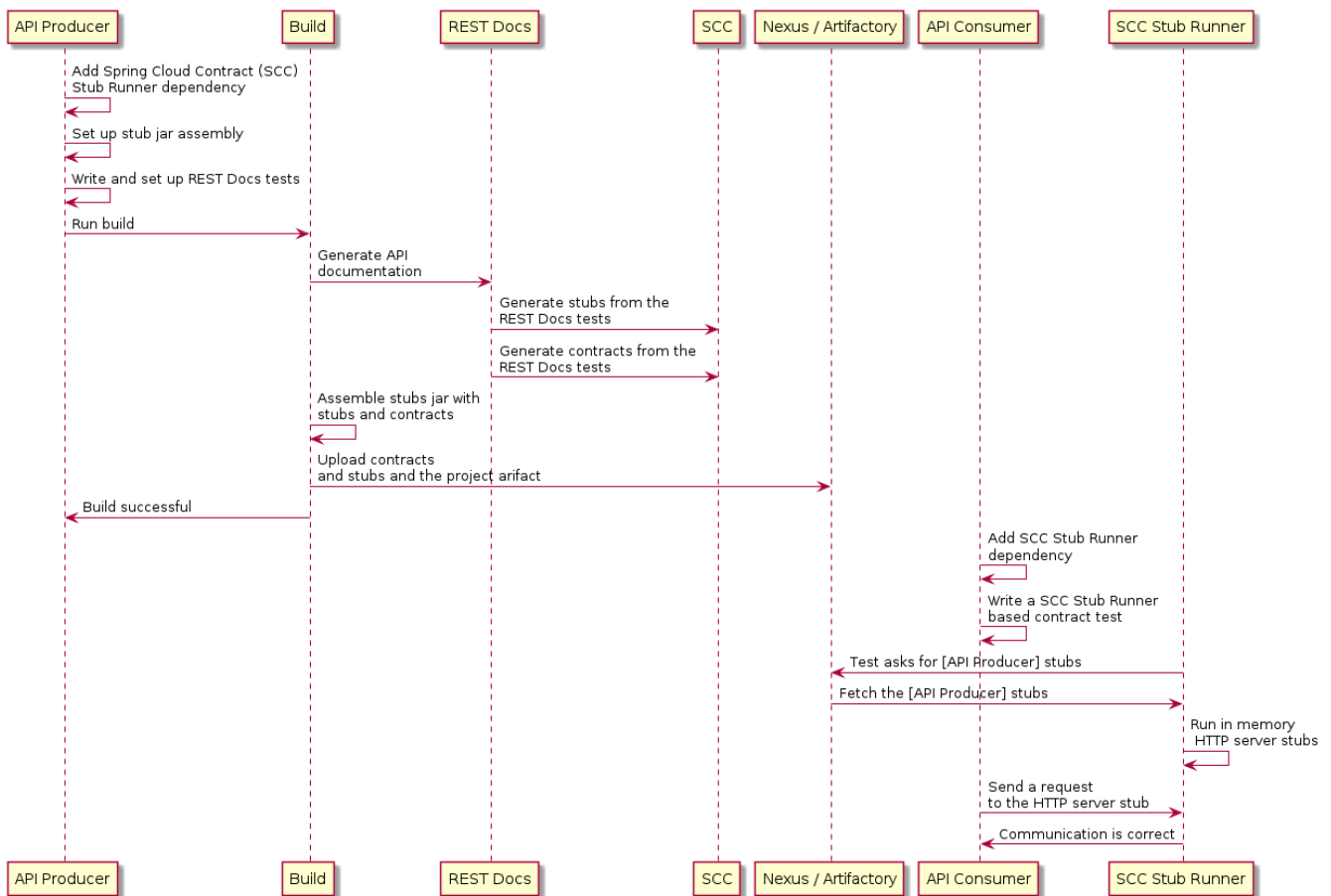
```

If you do it this way:

- All of your requests in the autogenerated tests are sent to the real endpoint with your context path included (for example, `/my-context-path/url`).
- Your contracts reflect that you have a context path. Your generated stubs also have that information (for example, in the stubs, you have to call `/my-context-path/url`).

3.3.6. Working with REST Docs

You can use [Spring REST Docs](#) to generate documentation (for example, in AsciiDoc format) for an HTTP API with Spring MockMvc, WebTestClient, or RestAssured. At the same time that you generate documentation for your API, you can also generate WireMock stubs by using Spring Cloud Contract WireMock. To do so, write your normal REST Docs test cases and use `@AutoConfigureRestDocs` to have stubs be automatically generated in the REST Docs output directory. The following UML diagram shows the REST Docs flow:



The following example uses `MockMvc`:

```

@RunWith(SpringRunner.class)
@SpringBootTest
@AutoConfigureRestDocs(outputDir = "target/snippets")
@AutoConfigureMockMvc
public class ApplicationTests {

    @Autowired
    private MockMvc mockMvc;

    @Test
    public void contextLoads() throws Exception {
        mockMvc.perform(get("/resource"))
            .andExpect(content().string("Hello World"))
            .andDo(document("resource"));
    }
}

```

This test generates a WireMock stub at `target/snippets/stubs/resource.json`. It matches all `GET` requests to the `/resource` path. The same example with `WebTestClient` (used for testing Spring WebFlux applications) would be as follows:


```

@RunWith(SpringRunner.class)
@SpringBootTest
@AutoConfigureRestDocs(outputDir = "target/snippets")
@AutoConfigureWebTestClient
public class ApplicationTests {

    @Autowired
    private WebTestClient client;

    @Test
    public void contextLoads() throws Exception {
        client.get().uri("/resource").exchange()
            .expectBody(String.class).isEqualTo("Hello World")
            .consumeWith(document("resource"));
    }
}

```

Without any additional configuration, these tests create a stub with a request matcher for the HTTP method and all headers except `host` and `content-length`. To match the request more precisely (for example, to match the body of a POST or PUT), we need to explicitly create a request matcher. Doing so has two effects:

- Creating a stub that matches only in the way you specify.
- Asserting that the request in the test case also matches the same conditions.

The main entry point for this feature is `WireMockRestDocs.verify()`, which can be used as a substitute for the `document()` convenience method, as the following example shows:

```

import static
org.springframework.cloud.contract.wiremock.restdocs.WireMockRestDocs.verify;

@RunWith(SpringRunner.class)
@SpringBootTest
@AutoConfigureRestDocs(outputDir = "target/snippets")
@AutoConfigureMockMvc
public class ApplicationTests {

    @Autowired
    private MockMvc mockMvc;

    @Test
    public void contextLoads() throws Exception {
        mockMvc.perform(post("/resource")
            .content("{\"id\":\"123456\",\"message\":\"Hello World\"}"))
            .andExpect(status().isOk())
            .andDo(verify().jsonPath("$.id"))
            .andDo(document("resource"));
    }
}

```

The preceding contract specifies that any valid POST with an `id` field receives the response defined in this test. You can chain together calls to `.jsonPath()` to add additional matchers. If JSON Path is unfamiliar, the [JayWay documentation](#) can help you get up to speed. The `WebTestClient` version of this test has a similar `verify()` static helper that you insert in the same place.

Instead of the `jsonPath` and `contentType` convenience methods, you can also use the WireMock APIs to verify that the request matches the created stub, as the following example shows:

```

@Test
public void contextLoads() throws Exception {
    mockMvc.perform(post("/resource")
        .content("{\"id\":\"123456\",\"message\":\"Hello World\"}"))
        .andExpect(status().isOk())
        .andDo(verify()
            .wiremock(WireMock.post(urlPathEquals("/resource"))
                .withRequestBody(matchingJsonPath("$.id"))
                .andDo(document("post-resource"))));
}

```

The WireMock API is rich. You can match headers, query parameters, and the request body by regex as well as by JSON path. You can use these features to create stubs with a wider range of parameters. The preceding example generates a stub resembling the following example:

post-resource.json

```
{
  "request" : {
    "url" : "/resource",
    "method" : "POST",
    "bodyPatterns" : [ {
      "matchesJsonPath" : "$.id"
    }
  ],
  "response" : {
    "status" : 200,
    "body" : "Hello World",
    "headers" : {
      "X-Application-Context" : "application:-1",
      "Content-Type" : "text/plain"
    }
  }
}
```



You can use either the `wiremock()` method or the `jsonPath()` and `contentType()` methods to create request matchers, but you cannot use both approaches.

On the consumer side, you can make the `resource.json` generated earlier in this section available on the classpath (by [Publishing Stubs as JARs](#), for example). After that, you can create a stub that uses WireMock in a number of different ways, including by using `@AutoConfigureWireMock(stubs="classpath:resource.json")`, as described earlier in this document.

Generating Contracts with REST Docs

You can also generate Spring Cloud Contract DSL files and documentation with Spring REST Docs. If you do so in combination with Spring Cloud WireMock, you get both the contracts and the stubs.

Why would you want to use this feature? Some people in the community asked questions about a situation in which they would like to move to DSL-based contract definition, but they already have a lot of Spring MVC tests. Using this feature lets you generate the contract files that you can later modify and move to folders (defined in your configuration) so that the plugin finds them.



You might wonder why this functionality is in the WireMock module. The functionality is there because it makes sense to generate both the contracts and the stubs.

Consider the following test:

```

this.mockMvc

.perform(post("/foo").accept(MediaType.APPLICATION_PDF).accept(MediaType.APPLICATION_JSON)

.contentType(MediaType.APPLICATION_JSON).content("{\"foo\": 23, \"bar\" : \"baz\"
}"))

    .andExpect(status().isOk()).andExpect(content().string("bar"))
    // first WireMock
    .andDo(WireMockRestDocs.verify().jsonPath("$.foo >= 20")
        .jsonPath("$.bar in ['baz', 'bazz', 'bazzz']")
        .contentType(MediaType.valueOf("application/json")))
    // then Contract DSL documentation
    .andDo(document("index",
SpringCloudContractRestDocs.dslContract(Maps.of("priority", 1))));

```

The preceding test creates the stub presented in the previous section, generating both the contract and a documentation file.

The contract is called `index.groovy` and might resemble the following example:

```

import org.springframework.cloud.contract.spec.Contract

Contract.make {
    request {
        method 'POST'
        url '/foo'
        body(''
            {"foo": 23 }
        '')
        headers {
            header(''Accept'', ''application/json'')
            header(''Content-Type'', ''application/json'')
        }
    }
    response {
        status OK()
        body(''
            bar
        '')
        headers {
            header(''Content-Type'', ''application/json;charset=UTF-8'')
            header(''Content-Length'', ''3'')
        }
        bodyMatchers {
            jsonPath('$[?(@.foo >= 20)]', byType())
        }
    }
}

```

The generated document (formatted in AsciiDoc in this case) contains a formatted contract. The location of this file would be [index/dsl-contract.adoc](#).

Specifying the priority attribute

The method `SpringCloudContractRestDocs.dslContract()` takes an optional `Map` parameter that allows you to specify additional attributes in the template.

One of these attributes is the [priority](#) field that you may specify as follows:

```
SpringCloudContractRestDocs.dslContract(Map.of("priority", 1))
```

Overriding the DSL contract template

By default, the output of the contract is based on a file named `default-dsl-contract-only.snippet`.

You may provide a custom template file instead by overriding the `getTemplate()` method as follows:

```
new ContractDslSnippet(){
    @Override
    protected String getTemplate() {
        return "custom-dsl-contract";
    }
});
```

so the example above showing this line

```
.andDo(document("index", SpringCloudContractRestDocs.dslContract()));
```

should be changed to:

```
.andDo(document("index", new ContractDslSnippet(){
    @Override
    protected String getTemplate() {
        return "custom-dsl-template";
    }
}));
```

Templates are resolved by looking for resources on the classpath. The following locations are checked in order:

- `org/springframework/restdocs/templates/${templateFormatId}/${name}.snippet`
- `org/springframework/restdocs/templates/${name}.snippet`
- `org/springframework/restdocs/templates/${templateFormatId}/default-${name}.snippet`

Therefore in the example above you should place a file named `custom-dsl-template.snippet` in `src/test/resources/org/springframework/restdocs/templates/custom-dsl-template.snippet`

3.3.7. GraphQL

Since [GraphQL](#) is essentially HTTP you can write a contract for it by creating a standard HTTP contract with an additional `metadata` entry with key `verifier` and a mapping `tool=graphql`.

Groovy

```
import org.springframework.cloud.contract.spec.Contract

Contract.make {

    request {
        method(POST())
        url("/graphql")
        headers {
            contentType("application/json")
        }
        body('''
{
    "query":"query queryName($personName: String!) {\n  personToCheck(name:
$personName) {\n    name\n    age\n  }\n}\n\n\n\n",
    "variables":{"personName":"Old Enough"},
    "operationName":"queryName"
}
''')
    }

    response {
        status(200)
        headers {
            contentType("application/json")
        }
        body('''\
{
  "data": {
    "personToCheck": {
      "name": "Old Enough",
      "age": "40"
    }
  }
}
''')
    }

    metadata(verifier: [
        tool: "graphql"
    ])
}
```

YAML

```
---
request:
  method: "POST"
  url: "/graphql"
  headers:
    Content-Type: "application/json"
  body:
    query: "query queryName($personName: String!) { personToCheck(name:
$personName)
      {      name      age  } }"
    variables:
      personName: "Old Enough"
      operationName: "queryName"
  matchers:
    headers:
      - key: "Content-Type"
        regex: "application/json.*"
        regexType: "as_string"
response:
  status: 200
  headers:
    Content-Type: "application/json"
  body:
    data:
      personToCheck:
        name: "Old Enough"
        age: "40"
  matchers:
    headers:
      - key: "Content-Type"
        regex: "application/json.*"
        regexType: "as_string"
name: "shouldRetrieveOldEnoughPerson"
metadata:
  verifier:
    tool: "graphql"
```

Adding the metadata section will change the way the default, WireMock stub is built. It will now use the Spring Cloud Contract request matcher, so that e.g. the **query** part of the GraphQL request gets compared against the real request by ignoring whitespaces.

Producer Side Setup

On the producer side your configuration can look as follows.

Maven

```
<plugin>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-contract-maven-plugin</artifactId>
  <version>${spring-cloud-contract.version}</version>
  <extensions>>true</extensions>
  <configuration>
    <testMode>EXPLICIT</testMode>
    <baseClassForTests>com.example.BaseClass</baseClassForTests>
  </configuration>
</plugin>
```

Gradle

```
contracts {
  testMode = "EXPLICIT"
  baseClassForTests = "com.example.BaseClass"
}
```

The base class would set up the application running on a random port.

Base Class

```
@SpringBootTest(classes = ProducerApplication.class,
  properties = "graphql.servlet.websocket.enabled=false",
  webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT)
public abstract class BaseClass {

  @LocalServerPort int port;

  @BeforeEach
  public void setup() {
    RestAssured.baseURI = "http://localhost:" + port;
  }
}
```

Consumer Side Setup

Example of a consumer side test of the GraphQL API.

Consumer Side Test

```
@SpringBootTest(webEnvironment = WebEnvironment.NONE)
public class BeerControllerGraphQLTest {

    @RegisterExtension
    static StubRunnerExtension rule = new StubRunnerExtension()
        .downloadStub("com.example","beer-api-producer-graphql")
        .stubsMode(StubRunnerProperties.StubsMode.LOCAL);

    private static final String REQUEST_BODY = "{\n"
        + "\"query\": \"query queryName($personName: String!) {\n"
        + "  personToCheck(name: $personName) {\n"
        + "    name\n"
        + "    age\n"
        + "  }\n"
        + "}"

    @Test
    public void should_send_a_graphql_request() {
        ResponseEntity<String> responseEntity = new RestTemplate()
            .exchange(RequestEntity
                .post(URI.create("http://localhost:" +
                    rule.findStubUrl("beer-api-producer-graphql").getPort() + "/graphql"))
                .contentType(MediaType.APPLICATION_JSON)
                .body(REQUEST_BODY), String.class);

        BDDAssertions.then(responseEntity.getStatusCodeValue()).isEqualTo(200);
    }
}
```

3.3.8. GRPC

GRPC is an RPC framework built on top of HTTP/2 for which Spring Cloud Contract has basic support.



Spring Cloud Contract has an experimental support for basic use cases of GRPC. Unfortunately, due to GRPC's tweaking of the HTTP/2 Header frames, it's impossible to assert the `grpc-status` header.

Let's look at the following contract.

Groovy contract

```
package contracts.beer.rest

import org.springframework.cloud.contract.spec.Contract
import org.springframework.cloud.contract.verifier.http.ContractVerifierHttpMetaData

Contract.make {
    description("""
Represents a successful scenario of getting a beer
...
given:
    client is old enough
when:
    he applies for a beer
then:
    we'll grant him the beer
...

""")
    request {
        method 'POST'
        url '/beer.BeerService/check'
        body(fileAsBytes("PersonToCheck_old_enough.bin"))
        headers {
            contentType("application/grpc")
            header("te", "trailers")
        }
    }
    response {
        status 200
        body(fileAsBytes("Response_old_enough.bin"))
        headers {
            contentType("application/grpc")
            header("grpc-encoding", "identity")
            header("grpc-accept-encoding", "gzip")
        }
    }
    metadata([
        "verifierHttp": [
            "protocol":
ContractVerifierHttpMetaData.Protocol.H2_PRIOR_KNOWLEDGE.toString()
        ]
    ])
}
```

Producer Side Setup

In order to leverage the HTTP/2 support you must set the **CUSTOM** test mode as follow.

Maven

```
<plugin>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-contract-maven-plugin</artifactId>
  <version>${spring-cloud-contract.version}</version>
  <extensions>>true</extensions>
  <configuration>
    <testMode>CUSTOM</testMode>
    <packageWithBaseClasses>com.example</packageWithBaseClasses>
  </configuration>
</plugin>
```

Gradle

```
contracts {
  packageWithBaseClasses = 'com.example'
  testMode = "CUSTOM"
}
```

The base class would set up the application running on a random port. It will also set the **HttpVerifier** implementation to one that can use the HTTP/2 protocol. Spring Cloud Contract comes with the **OkHttpHttpVerifier** implementation.

Base Class

```
@SpringBootTest(classes = BeerRestBase.Config.class,
    webEnvironment = SpringBootTest.WebEnvironment.NONE,
    properties = {
        "grpc.server.port=0"
    })
public abstract class BeerRestBase {

    @Autowired
    GrpcServerProperties properties;

    @Configuration
    @EnableAutoConfiguration
    static class Config {

        @Bean
        ProducerController producerController(PersonCheckingService
personCheckingService) {
            return new ProducerController(personCheckingService);
        }

        @Bean
        PersonCheckingService testPersonCheckingService() {
            return argument -> argument.getAge() >= 20;
        }

        @Bean
        HttpVerifier httpOkVerifier(GrpcServerProperties properties) {
            return new OkHttpHttpVerifier("localhost:" + properties.getPort());
        }
    }
}
```

Consumer Side Setup

Example of GRPC consumer side test. Due to the unusual behaviour of the GRPC server side, the stub is unable to return the `grpc-status` header in the proper moment. This is why we need to manually set the return status.

Consumer Side Test

```
@SpringBootTest(webEnvironment = WebEnvironment.NONE, classes =
GrpcTests.TestConfiguration.class, properties = {
    "grpc.client.beerService.address=static://localhost:5432",
    "grpc.client.beerService.negotiationType=TLS"
})
public class GrpcTests {

    @GrpcClient(value = "beerService", interceptorNames =
"fixedStatusSendingClientInterceptor")
    BeerServiceGrpc.BeerServiceBlockingStub beerServiceBlockingStub;

    int port;

    @RegisterExtension
    static StubRunnerExtension rule = new StubRunnerExtension()
        .downloadStub("com.example", "beer-api-producer-grpc")
        // With WireMock PlainText mode you can just set an HTTP port
        .withPort(5432)
        .stubsMode(StubRunnerProperties.StubsMode.LOCAL)
        .withHttpServerStubConfigurer(MyWireMockConfigurer.class);

    @BeforeEach
    public void setupPort() {
        this.port = rule.findStubUrl("beer-api-producer-grpc").getPort();
    }

    @Test
    public void should_give_me_a_beer_when_im_old_enough() throws Exception {
        Response response =
beerServiceBlockingStub.check(PersonToCheck.newBuilder().setAge(23).build());

        BDDAssertions.then(response.getStatus()).isEqualTo(Response.BeerCheckStatus.OK);
    }

    @Test
    public void should_reject_a_beer_when_im_too_young() throws Exception {
        Response response =
beerServiceBlockingStub.check(PersonToCheck.newBuilder().setAge(17).build());
        response = response == null ? Response.newBuilder().build() : response;

        BDDAssertions.then(response.getStatus()).isEqualTo(Response.BeerCheckStatus.NOT_OK
);
    }

    // Not necessary with WireMock PlainText mode
    static class MyWireMockConfigurer extends WireMockHttpServerStubConfigurer {
```

```

@Override
public WireMockConfiguration configure(WireMockConfiguration
httpStubConfiguration, HttpServerStubConfiguration httpServerStubConfiguration) {
    return httpStubConfiguration
        .httpsPort(5432);
}
}

@Configuration
@ImportAutoConfiguration(GrpcClientAutoConfiguration.class)
static class TestConfiguration {

    // Not necessary with WireMock PlainText mode
    @Bean
    public GrpcChannelConfigurer keepAliveClientConfigurer() {
        return (channelBuilder, name) -> {
            if (channelBuilder instanceof NettyChannelBuilder) {
                try {
                    ((NettyChannelBuilder) channelBuilder)
                        .sslContext(GrpcSslContexts.forClient()

.trustManager(InsecureTrustManagerFactory.INSTANCE)
                            .build());
                }
                catch (SSLException e) {
                    throw new IllegalStateException(e);
                }
            }
        };
    }

    /**
     * GRPC client interceptor that sets the returned status always to OK.
     * You might want to change the return status depending on the received
stub payload.
     *
     * Hopefully in the future this will be unnecessary and will be removed.
     */
    @Bean
    ClientInterceptor fixedStatusSendingClientInterceptor() {
        return new ClientInterceptor() {
            @Override
            public <ReqT, RespT> ClientCall<ReqT, RespT>
interceptCall(MethodDescriptor<ReqT, RespT> method, CallOptions callOptions,
Channel next) {
                ClientCall<ReqT, RespT> call = next.newCall(method,
callOptions);
                return new ClientCall<ReqT, RespT>() {
                    @Override
                    public void start(Listener<RespT> responseListener,
Metadata headers) {

```

```

        Listener<RespT> listener = new Listener<RespT>() {
            @Override
            public void onHeaders(Metadata headers) {
                responselistener.onHeaders(headers);
            }

            @Override
            public void onMessage(RespT message) {
                responselistener.onMessage(message);
            }

            @Override
            public void onClose(Status status, Metadata
trailers) {
                // TODO: This must be fixed somehow either in
Jetty (WireMock) or somewhere else
                responselistener.onClose(Status.OK, trailers);
            }

            @Override
            public void onReady() {
                responselistener.onReady();
            }
        };
        call.start(listener, headers);
    }

    @Override
    public void request(int numMessages) {
        call.request(numMessages);
    }

    @Override
    public void cancel(@Nullable String message, @Nullable
Throwable cause) {
        call.cancel(message, cause);
    }

    @Override
    public void halfClose() {
        call.halfClose();
    }

    @Override
    public void sendMessage(ReqT message) {
        call.sendMessage(message);
    }
};
    }
};
}

```



```
}  
}
```

3.4. Messaging

Spring Cloud Contract lets you verify applications that use messaging as a means of communication. All of the integrations shown in this document work with Spring, but you can also create one of your own and use that.

3.4.1. Messaging DSL Top-level Elements

The DSL for messaging looks a little bit different than the one that focuses on HTTP. The following sections explain the differences:

- [Output Triggered by a Method](#)
- [Consumer/Producer](#)
- [Common](#)

Output Triggered by a Method

The output message can be triggered by calling a method (such as a `Scheduler` when a contract was started and when a message was sent), as shown in the following example:

Groovy

```
def dsl = Contract.make {
    // Human readable description
    description 'Some description'
    // Label by means of which the output message can be triggered
    label 'some_label'
    // input to the contract
    input {
        // the contract will be triggered by a method
        triggeredBy('bookReturnedTriggered()')
    }
    // output message of the contract
    outputMessage {
        // destination to which the output message will be sent
        sentTo('output')
        // the body of the output message
        body('{ "bookName" : "foo" }')
        // the headers of the output message
        headers {
            header('BOOK-NAME', 'foo')
        }
    }
}
```

YAML

```
# Human readable description
description: Some description
# Label by means of which the output message can be triggered
label: some_label
input:
  # the contract will be triggered by a method
  triggeredBy: bookReturnedTriggered()
# output message of the contract
outputMessage:
  # destination to which the output message will be sent
  sentTo: output
  # the body of the output message
  body:
    bookName: foo
  # the headers of the output message
  headers:
    BOOK-NAME: foo
```

In the previous example case, the output message is sent to `output` if a method called `bookReturnedTriggered` is invoked. On the message publisher's side, we generate a test that calls that method to trigger the message. On the consumer side, you can use `some_label` to trigger the

message.

Consumer/Producer



This section is valid only for the Groovy DSL.

In HTTP, you have a notion of `client/stub` and ``server/test` notation. You can also use those paradigms in messaging. In addition, Spring Cloud Contract Verifier also provides the `consumer` and `producer` methods (note that you can use either `$` or `value` methods to provide `consumer` and `producer` parts).

Common

In the `input` or `outputMessage` section, you can call `assertThat` with the name of a `method` (for example, `assertThatMessageIsOnTheQueue()`) that you have defined in the base class or in a static import. Spring Cloud Contract runs that method in the generated test.

3.4.2. Integrations

You can use one of the following integration configurations:

- Apache Camel
- Spring Integration
- Spring Cloud Stream
- Spring JMS

Since we use Spring Boot, if you have added one of these libraries to the classpath, all the messaging configuration is automatically set up.



Remember to put `@AutoConfigureMessageVerifier` on the base class of your generated tests. Otherwise, the messaging part of Spring Cloud Contract does not work.

If you want to use Spring Cloud Stream, remember to add a test dependency on `org.springframework.cloud:spring-cloud-stream`, as follows:

Maven

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-stream</artifactId>
  <type>test-jar</type>
  <scope>test</scope>
  <classifier>test-binder</classifier>
</dependency>
```

Gradle

```
testImplementation(group: 'org.springframework.cloud', name:
'spring-cloud-stream', classifier: 'test-binder')
```

Manual Integration Testing

The main interface used by the tests is `org.springframework.cloud.contract.verifier.messaging.MessageVerifierSender` and `org.springframework.cloud.contract.verifier.messaging.MessageVerifierReceiver`. It defines how to send and receive messages.

In a test, you can inject a `ContractVerifierMessageExchange` to send and receive messages that follow the contract. Then add `@AutoConfigureMessageVerifier` to your test. The following example shows how to do so:

```
@RunWith(SpringTestRunner.class)
@SpringBootTest
@AutoConfigureMessageVerifier
public static class MessagingContractTests {

    @Autowired
    private MessageVerifier verifier;
    ...
}
```



If your tests require stubs as well, then `@AutoConfigureStubRunner` includes the messaging configuration, so you only need the one annotation.

3.4.3. Producer Side Messaging Test Generation

Having the `input` or `outputMessage` sections in your DSL results in creation of tests on the publisher's side. By default, JUnit 4 tests are created. However, there is also a possibility to create JUnit 5, TestNG, or Spock tests.



The destination passed to `messageFrom` or `sentTo` can have different meanings for different messaging implementations. For Stream and Integration, it is first resolved as a `destination` of a channel. Then, if there is no such `destination`, it is resolved as a channel name. For Camel, that's a certain component (for example, `jms`).

Consider the following contract:

Groovy

```
def contractDsl = Contract.make {
  name "foo"
  label 'some_label'
  input {
    triggeredBy('bookReturnedTriggered()')
  }
  outputMessage {
    sentTo('activemq:output')
    body('{ "bookName" : "foo" }')
    headers {
      header('BOOK-NAME', 'foo')
      messagingContentType(applicationJson())
    }
  }
}
```

YAML

```
label: some_label
input:
  triggeredBy: bookReturnedTriggered
outputMessage:
  sentTo: activemq:output
  body:
    bookName: foo
  headers:
    BOOK-NAME: foo
  contentType: application/json
```

For the preceding example, the following test would be created:

JUnit

```
package com.example;

import com.jayway.jsonpath.DocumentContext;
import com.jayway.jsonpath.JsonPath;
import org.junit.Test;
import org.junit.Rule;
import javax.inject.Inject;
import
org.springframework.cloud.contract.verifier.messaging.internal.ContractVerifierObj
ectMapper;
import
org.springframework.cloud.contract.verifier.messaging.internal.ContractVerifierMes
sage;
import
org.springframework.cloud.contract.verifier.messaging.internal.ContractVerifierMes
saging;

import static
org.springframework.cloud.contract.verifier.assertion.SpringCloudContractAssertion
s.assertThat;
import static
org.springframework.cloud.contract.verifier.util.ContractVerifierUtil.*;
import static com.toomuchcoding.jsonassert.JsonAssertion.assertThatJson;
import static
org.springframework.cloud.contract.verifier.messaging.util.ContractVerifierMessagi
ngUtil.headers;
import static
org.springframework.cloud.contract.verifier.util.ContractVerifierUtil.fileToBytes;

@SuppressWarnings("rawtypes")
public class FooTest {
    @Inject ContractVerifierMessaging contractVerifierMessaging;
    @Inject ContractVerifierObjectMapper contractVerifierObjectMapper;

    @Test
    public void validate_foo() throws Exception {
        // when:
        bookReturnedTriggered();

        // then:
        ContractVerifierMessage response =
contractVerifierMessaging.receive("activemq:output",
contract(this, "foo.yml"));
assertThat(response).isNotNull();

        // and:
assertThat(response.getHeader("BOOK-NAME")).isNotNull();
assertThat(response.getHeader("BOOK-
```

```

NAME").toString()).isEqualTo("foo");
        assertThat(response.getHeader("contentType")).isNotNull();

assertThat(response.getHeader("contentType").toString()).isEqualTo("application/js
on");

        // and:
        DocumentContext parsedJson =
JsonPath.parse(contractVerifierObjectMapper.writeValueAsString(response.getPayload
()));
        assertThatJson(parsedJson).field("[ 'bookName' ]").isEqualTo("foo");
    }
}

```

Spock

```

package com.example

import com.jayway.jsonpath.DocumentContext
import com.jayway.jsonpath.JsonPath
import spock.lang.Specification
import javax.inject.Inject
import
org.springframework.cloud.contract.verifier.messaging.internal.ContractVerifierObj
ectMapper
import
org.springframework.cloud.contract.verifier.messaging.internal.ContractVerifierMes
sage
import
org.springframework.cloud.contract.verifier.messaging.internal.ContractVerifierMes
saging

import static
org.springframework.cloud.contract.verifier.assertion.SpringCloudContractAssertion
s.assertThat
import static
org.springframework.cloud.contract.verifier.util.ContractVerifierUtil.*
import static com.toomuchcoding.jsonassert.JsonAssertion.assertThatJson
import static
org.springframework.cloud.contract.verifier.messaging.util.ContractVerifierMessagi
ngUtil.headers
import static
org.springframework.cloud.contract.verifier.util.ContractVerifierUtil.fileToBytes

@SuppressWarnings("rawtypes")
class FooSpec extends Specification {
    @Inject ContractVerifierMessaging contractVerifierMessaging
    @Inject ContractVerifierObjectMapper contractVerifierObjectMapper

    def validate_foo() throws Exception {

```

```

when:
    bookReturnedTriggered()

then:
    ContractVerifierMessage response =
contractVerifierMessaging.receive("activemq:output",
    contract(this, "foo.yml"))
    response != null

and:
    response.getHeader("BOOK-NAME") != null
    response.getHeader("BOOK-NAME").toString() == 'foo'
    response.getHeader("contentType") != null
    response.getHeader("contentType").toString() == 'application/json'

and:
    DocumentContext parsedJson =
JsonPath.parse(contractVerifierObjectMapper.writeValueAsString(response.getPayload
()))
        assertThatJson(parsedJson).field("[ 'bookName' ]").isEqualTo("foo")
    }
}

```

3.4.4. Consumer Stub Generation

Unlike in the HTTP part, in messaging, we need to publish the contract definition inside the JAR with a stub. Then it is parsed on the consumer side, and proper stubbed routes are created.



If you have multiple frameworks on the classpath, Stub Runner needs to define which one should be used. Assume that you have AMQP, Spring Cloud Stream, and Spring Integration on the classpath and that you want to use Spring AMQP. Then you need to set `stubrunner.stream.enabled=false` and `stubrunner.integration.enabled=false`. That way, the only remaining framework is Spring AMQP.

Stub triggering

To trigger a message, use the `StubTrigger` interface, as the following example shows:


```

package org.springframework.cloud.contract.stubrunner;

import java.util.Collection;
import java.util.Map;

/**
 * Contract for triggering stub messages.
 *
 * @author Marcin Grzejszczak
 */
public interface StubTrigger {

    /**
     * Triggers an event by a given label for a given {@code groupid:artifactid}
     notation.
     * You can use only {@code artifactId} too.
     *
     * Feature related to messaging.
     * @param ivyNotation ivy notation of a stub
     * @param labelName name of the label to trigger
     * @return true - if managed to run a trigger
     */
    boolean trigger(String ivyNotation, String labelName);

    /**
     * Triggers an event by a given label.
     *
     * Feature related to messaging.
     * @param labelName name of the label to trigger
     * @return true - if managed to run a trigger
     */
    boolean trigger(String labelName);

    /**
     * Triggers all possible events.
     *
     * Feature related to messaging.
     * @return true - if managed to run a trigger
     */
    boolean trigger();

    /**
     * Feature related to messaging.
     * @return a mapping of ivy notation of a dependency to all the labels it has.
     */
    Map<String, Collection<String>> labels();
}

```

For convenience, the `StubFinder` interface extends `StubTrigger`, so you need only one or the other in your tests.

`StubTrigger` gives you the following options to trigger a message:

- [Trigger by Label](#)
- [Trigger by Group and Artifact IDs](#)
- [Trigger by Artifact IDs](#)
- [Trigger All Messages](#)

Trigger by Label

The following example shows how to trigger a message with a label:

```
stubFinder.trigger('return_book_1')
```

Trigger by Group and Artifact IDs

The following example shows how to trigger a message by group and artifact IDs:

```
stubFinder.trigger('org.springframework.cloud.contract.verifier.stubs:streamService', 'return_book_1')
```

Trigger by Artifact IDs

The following example shows how to trigger a message from artifact IDs:

```
stubFinder.trigger('streamService', 'return_book_1')
```

Trigger All Messages

The following example shows how to trigger all messages:

```
stubFinder.trigger()
```

3.4.5. Consumer Side Messaging With Apache Camel

Spring Cloud Contract Stub Runner's messaging module gives you an easy way to integrate with Apache Camel. For the provided artifacts, it automatically downloads the stubs and registers the required routes.

Adding Apache Camel to the Project

You can have both Apache Camel and Spring Cloud Contract Stub Runner on the classpath. Remember to annotate your test class with `@AutoConfigureStubRunner`.

Disabling the Functionality

If you need to disable this functionality, set the `stubrunner.camel.enabled=false` property.

Examples

Assume that we have the following Maven repository with deployed stubs for the `camelService` application:

```
├── .m2
│   ├── repository
│   │   ├── io
│   │   │   ├── codearte
│   │   │   │   ├── accurest
│   │   │   │   │   ├── stubs
│   │   │   │   │   │   ├── camelService
│   │   │   │   │   │   │   ├── 0.0.1-SNAPSHOT
│   │   │   │   │   │   │   │   ├── camelService-0.0.1-SNAPSHOT.pom
│   │   │   │   │   │   │   │   ├── camelService-0.0.1-SNAPSHOT-stubs.jar
│   │   │   │   │   │   │   │   ├── maven-metadata-local.xml
│   │   │   │   │   │   │   └── maven-metadata-local.xml
```

Further, assume that the stubs contain the following structure:

```
├── META-INF
│   └── MANIFEST.MF
├── repository
│   ├── accurest
│   │   └── bookReturned1.groovy
│   └── mappings
```

Now consider the following contract:

```
Contract.make {
  label 'return_book_1'
  input {
    triggeredBy('bookReturnedTriggered()')
  }
  outputMessage {
    sentTo('rabbitmq:output?queue=output')
    body(''{ "bookName" : "foo" }'')
    headers {
      header('BOOK-NAME', 'foo')
    }
  }
}
```

To trigger a message from the `return_book_1` label, we use the `StubTrigger` interface, as follows:

```
stubFinder.trigger("return_book_1")
```

That will send out a message to the destination described in the output message of the contract.

3.4.6. Consumer Side Messaging with Spring Integration

Spring Cloud Contract Stub Runner's messaging module gives you an easy way to integrate with Spring Integration. For the provided artifacts, it automatically downloads the stubs and registers the required routes.

Adding the Runner to the Project

You can have both Spring Integration and Spring Cloud Contract Stub Runner on the classpath. Remember to annotate your test class with `@AutoConfigureStubRunner`.

Disabling the Functionality

If you need to disable this functionality, set the `stubrunner.integration.enabled=false` property.

Examples

Assume that you have the following Maven repository with deployed stubs for the `integrationService` application:

```

└── .m2
    ├── repository
    │   ├── io
    │   │   ├── codearte
    │   │   │   ├── accurest
    │   │   │   │   ├── stubs
    │   │   │   │   │   ├── integrationService
    │   │   │   │   │   │   ├── 0.0.1-SNAPSHOT
    │   │   │   │   │   │   │   ├── integrationService-0.0.1-SNAPSHOT.pom
    │   │   │   │   │   │   │   ├── integrationService-0.0.1-SNAPSHOT-
    │   │   │   │   │   │   │   └── maven-metadata-local.xml
    │   │   │   │   │   └── maven-metadata-local.xml
    │   │   └── stubs.jar
    └── stubs.jar

```

Further assume the stubs contain the following structure:

```

└── META-INF
    ├── MANIFEST.MF
    ├── repository
    │   ├── accurest
    │   │   ├── bookReturned1.groovy
    │   └── mappings

```

Consider the following contract:

```

Contract.make {
    label 'return_book_1'
    input {
        triggeredBy('bookReturnedTriggered()')
    }
    outputMessage {
        sentTo('output')
        body('''{ "bookName" : "foo" }''')
        headers {
            header('BOOK-NAME', 'foo')
        }
    }
}

```

Now consider the following Spring Integration Route:

```

<?xml version="1.0" encoding="UTF-8"?>
<beans:beans xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:beans="http://www.springframework.org/schema/beans"
  xmlns="http://www.springframework.org/schema/integration"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
    https://www.springframework.org/schema/beans/spring-beans.xsd
    http://www.springframework.org/schema/integration
    http://www.springframework.org/schema/integration/spring-
integration.xsd">

  <!-- REQUIRED FOR TESTING -->
  <bridge input-channel="output"
    output-channel="outputTest"/>

  <channel id="outputTest">
    <queue/>
  </channel>

</beans:beans>

```

To trigger a message from the `return_book_1` label, use the `StubTrigger` interface, as follows:

```
stubFinder.trigger('return_book_1')
```

That will send out a message to the destination described in the output message of the contract.

3.4.7. Consumer Side Messaging With Spring Cloud Stream

Spring Cloud Contract Stub Runner's messaging module gives you an easy way to integrate with Spring Stream. For the provided artifacts, it automatically downloads the stubs and registers the required routes.



If Stub Runner's integration with the Stream `messageFrom` or `sentTo` strings are resolved first as the `destination` of a channel and no such `destination` exists, the destination is resolved as a channel name.

If you want to use Spring Cloud Stream, remember to add a dependency on `org.springframework.cloud:spring-cloud-stream` test support, as follows:



Maven

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-stream-test-binder</artifactId>
  <scope>test</scope>
</dependency>
```

Gradle

```
testImplementation('org.springframework.cloud:spring-cloud-stream-
test-binder')
```

Adding the Runner to the Project

You can have both Spring Cloud Stream and Spring Cloud Contract Stub Runner on the classpath. Remember to annotate your test class with `@AutoConfigureStubRunner`.

Disabling the Functionality

If you need to disable this functionality, set the `stubrunner.stream.enabled=false` property.

Examples

Assume that you have the following Maven repository with deployed stubs for the `streamService` application:

```
.m2
├── repository
│   ├── io
│   │   ├── codearte
│   │   │   ├── accurest
│   │   │   │   ├── stubs
│   │   │   │   │   ├── streamService
│   │   │   │   │   │   ├── 0.0.1-SNAPSHOT
│   │   │   │   │   │   │   ├── streamService-0.0.1-SNAPSHOT.pom
│   │   │   │   │   │   │   ├── streamService-0.0.1-SNAPSHOT-stubs.jar
│   │   │   │   │   │   │   ├── maven-metadata-local.xml
│   │   │   │   │   │   └── maven-metadata-local.xml
```

Further assume the stubs contain the following structure:

```

├── META-INF
│   └── MANIFEST.MF
└── repository
    ├── accurest
    │   └── bookReturned1.groovy
    └── mappings

```

Consider the following contract:

```

Contract.make {
    label 'return_book_1'
    input { triggeredBy('bookReturnedTriggered()') }
    outputMessage {
        sentTo('returnBook')
        body(''{ "bookName" : "foo" }''')
        headers { header('BOOK-NAME', 'foo') }
    }
}

```

Now consider the following Spring Cloud Stream function configuration:

```

@ImportAutoConfiguration(TestChannelBinderConfiguration.class)
@Configuration(proxyBeanMethods = true)
@EnableAutoConfiguration
protected static class Config {

    @Bean
    Function<String, String> test1() {
        return (input) -> {
            println "Test 1 [${input}]"
            return input
        }
    }
}

```

Now consider the following Spring configuration:


```
stubrunner.repositoryRoot: classpath:m2repo/repository/
stubrunner.ids:
org.springframework.cloud.contract.verifier.stubs:streamService:0.0.1-
SNAPSHOT:stubs
stubrunner.stubs-mode: remote
spring:
  cloud:
    stream:
      bindings:
        test1-in-0:
          destination: returnBook
        test1-out-0:
          destination: outputToAssertBook
      function:
        definition: test1

server:
  port: 0

debug: true
```

To trigger a message from the `return_book_1` label, use the `StubTrigger` interface as follows:

```
stubFinder.trigger('return_book_1')
```

That will send out a message to the destination described in the output message of the contract.

3.4.8. Consumer Side Messaging With Spring JMS

Spring Cloud Contract Stub Runner's messaging module provides an easy way to integrate with Spring JMS.

The integration assumes that you have a running instance of a JMS broker.

Adding the Runner to the Project

You need to have both Spring JMS and Spring Cloud Contract Stub Runner on the classpath. Remember to annotate your test class with `@AutoConfigureStubRunner`.

Examples

Assume that the stub structure looks as follows:

```
├── stubs
│   └── bookReturned1.groovy
```

Further assume the following test configuration:

```
stubrunner:
  repository-root: stubs:classpath:/stubs/
  ids: my:stubs
  stubs-mode: remote
spring:
  activemq:
    send-timeout: 1000
  jms:
    template:
      receive-timeout: 1000
```

Now consider the following contract:

```
Contract.make {
  label 'return_book_1'
  input {
    triggeredBy('bookReturnedTriggered()')
  }
  outputMessage {
    sentTo('output')
    body(''{ "bookName" : "foo" }''')
    headers {
      header('BOOKNAME', 'foo')
    }
  }
}
```

To trigger a message from the `return_book_1` label, we use the `StubTrigger` interface, as follows:

```
stubFinder.trigger('return_book_1')
```

That will send out a message to the destination described in the output message of the contract.

3.5. Spring Cloud Contract Stub Runner

One of the issues that you might encounter while using Spring Cloud Contract Verifier is passing the generated WireMock JSON stubs from the server side to the client side (or to various clients). The same takes place in terms of client-side generation for messaging.

Copying the JSON files and setting the client side for messaging manually is out of the question. That is why we introduced Spring Cloud Contract Stub Runner. It can automatically download and run the stubs for you.

3.5.1. Snapshot Versions

You can add the additional snapshot repository to your build file to use snapshot versions, which are automatically uploaded after every successful build, as follows:

Maven

```
<repositories>
  <repository>
    <id>spring-snapshots</id>
    <name>Spring Snapshots</name>
    <url>https://repo.spring.io/snapshot</url>
    <snapshots>
      <enabled>true</enabled>
    </snapshots>
  </repository>
  <repository>
    <id>spring-milestones</id>
    <name>Spring Milestones</name>
    <url>https://repo.spring.io/milestone</url>
    <snapshots>
      <enabled>false</enabled>
    </snapshots>
  </repository>
  <repository>
    <id>spring-releases</id>
    <name>Spring Releases</name>
    <url>https://repo.spring.io/release</url>
    <snapshots>
      <enabled>false</enabled>
    </snapshots>
  </repository>
</repositories>
<pluginRepositories>
  <pluginRepository>
    <id>spring-snapshots</id>
    <name>Spring Snapshots</name>
    <url>https://repo.spring.io/snapshot</url>
    <snapshots>
      <enabled>true</enabled>
    </snapshots>
  </pluginRepository>
  <pluginRepository>
    <id>spring-milestones</id>
    <name>Spring Milestones</name>
    <url>https://repo.spring.io/milestone</url>
    <snapshots>
      <enabled>false</enabled>
    </snapshots>
  </pluginRepository>
  <pluginRepository>
    <id>spring-releases</id>
    <name>Spring Releases</name>
    <url>https://repo.spring.io/release</url>
    <snapshots>
```

```
        <enabled>false</enabled>
    </snapshots>
</pluginRepository>
</pluginRepositories>
```

Gradle ([settings.xml](#))

```
pluginManagement {
    repositories {
        mavenLocal()
        maven { url "https://repo.spring.io/snapshot" }
        maven { url "https://repo.spring.io/milestone" }
        maven { url "https://repo.spring.io/release" }
        gradlePluginPortal()
    }
}
```

3.5.2. Publishing Stubs as JARs

The easiest approach to publishing stubs as jars is to centralize the way stubs are kept. For example, you can keep them as jars in a Maven repository.



For both Maven and Gradle, the setup comes ready to work. However, you can customize it if you want to.

The following example shows how to publish stubs as jars:

Maven

```
<!-- First disable the default jar setup in the properties section -->
<!-- we don't want the verifier to do a jar for us -->
<spring.cloud.contract.verifier.skip>true</spring.cloud.contract.verifier.skip>
```

```
<!-- Next add the assembly plugin to your build -->
<!-- we want the assembly plugin to generate the JAR -->
<plugin>
```

```
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-assembly-plugin</artifactId>
  <executions>
    <execution>
      <id>stub</id>
      <phase>prepare-package</phase>
      <goals>
        <goal>single</goal>
      </goals>
      <inherited>>false</inherited>
      <configuration>
        <attach>true</attach>
        <descriptors>
          $/tmp/releaser-1691636970130-0/spring-cloud-
contract/docs/src/assembly/stub.xml
        </descriptors>
      </configuration>
    </execution>
  </executions>
</plugin>
```

```
<!-- Finally setup your assembly. Below you can find the contents of
src/main/assembly/stub.xml -->
```

```
<assembly
  xmlns="http://maven.apache.org/plugins/maven-assembly-plugin/assembly/1.1.3"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/plugins/maven-assembly-
plugin/assembly/1.1.3 https://maven.apache.org/xsd/assembly-1.1.3.xsd">
  <id>stubs</id>
  <formats>
    <format>jar</format>
  </formats>
  <includeBaseDirectory>>false</includeBaseDirectory>
  <fileSets>
    <fileSet>
      <directory>src/main/java</directory>
      <outputDirectory>../outputDirectory</outputDirectory>
      <includes>
        <include>**com/example/model/*.*</include>
      </includes>
    </fileSet>
```

```

<fileSet>
  <directory>${project.build.directory}/classes</directory>
  <outputDirectory></outputDirectory>
  <includes>
    <include>**com/example/model/*. *</include>
  </includes>
</fileSet>
<fileSet>
  <directory>${project.build.directory}/snippets/stubs</directory>
  <outputDirectory>META-
INF/${project.groupId}/${project.artifactId}/${project.version}/mappings</outputDi
rectory>
  <includes>
    <include>**/*</include>
  </includes>
</fileSet>
<fileSet>
  <directory>$/tmp/releaser-1691636970130-0/spring-cloud-
contract/docs/src/test/resources/contracts</directory>
  <outputDirectory>META-
INF/${project.groupId}/${project.artifactId}/${project.version}/contracts</outputD
irectory>
  <includes>
    <include>**/*.groovy</include>
  </includes>
</fileSet>
</fileSets>
</assembly>

```

Gradle

```

ext {
  contractsDir = file("mappings")
  stubsOutputDirRoot = file("${project.buildDir}/production/${project.name}-
stubs/")
}

// Automatically added by plugin:
// copyContracts - copies contracts to the output folder from which JAR will be
created
// verifierStubsJar - JAR with a provided stub suffix

publishing {
  publications {
    stubs(MavenPublication) {
      artifactId "${project.name}-stubs"
      artifact verifierStubsJar
    }
  }
}
}

```

3.5.3. Stub Runner Core

The stub runner core runs stubs for service collaborators. Treating stubs as contracts of services lets you use stub-runner as an implementation of [Consumer-driven Contracts](#).

Stub Runner lets you automatically download the stubs of the provided dependencies (or pick those from the classpath), start WireMock servers for them, and feed them with proper stub definitions. For messaging, special stub routes are defined.

Retrieving stubs

You can pick from the following options of acquiring stubs:

- Aether-based solution that downloads JARs with stubs from Artifactory or Nexus
- Classpath-scanning solution that searches the classpath with a pattern to retrieve stubs
- Writing your own implementation of the `org.springframework.cloud.contract.stubrunner.StubDownloaderBuilder` for full customization

The latter example is described in the [Custom Stub Runner](#) section.

Downloading Stubs

You can control the downloading of stubs with the `stubsMode` switch. It picks value from the `StubRunnerProperties.StubsMode` enumeration. You can use the following options:

- `StubRunnerProperties.StubsMode.CLASSPATH` (default value): Picks stubs from the classpath
- `StubRunnerProperties.StubsMode.LOCAL`: Picks stubs from a local storage (for example, `.m2`)
- `StubRunnerProperties.StubsMode.REMOTE`: Picks stubs from a remote location

The following example picks stubs from a local location:

```
@AutoConfigureStubRunner(repositoryRoot="https://foo.bar", ids =
"com.example:beer-api-producer:+:stubs:8095", stubsMode =
StubRunnerProperties.StubsMode.LOCAL)
```

Classpath scanning

If you set the `stubsMode` property to `StubRunnerProperties.StubsMode.CLASSPATH` (or set nothing since `CLASSPATH` is the default value), the classpath is scanned. Consider the following example:

```
@AutoConfigureStubRunner(ids = {
    "com.example:beer-api-producer:+:stubs:8095",
    "com.example.foo:bar:1.0.0:superstubs:8096"
})
```


You can add the dependencies to your classpath, as follows:

Maven

```
<dependency>
  <groupId>com.example</groupId>
  <artifactId>beer-api-producer-restdocs</artifactId>
  <classifier>stubs</classifier>
  <version>0.0.1-SNAPSHOT</version>
  <scope>test</scope>
  <exclusions>
    <exclusion>
      <groupId>*</groupId>
      <artifactId>*</artifactId>
    </exclusion>
  </exclusions>
</dependency>
<dependency>
  <groupId>com.example.thing1</groupId>
  <artifactId>thing2</artifactId>
  <classifier>superstubs</classifier>
  <version>1.0.0</version>
  <scope>test</scope>
  <exclusions>
    <exclusion>
      <groupId>*</groupId>
      <artifactId>*</artifactId>
    </exclusion>
  </exclusions>
</dependency>
```

Gradle

```
testCompile("com.example:beer-api-producer-restdocs:0.0.1-SNAPSHOT:stubs") {
  transitive = false
}
testCompile("com.example.thing1:thing2:1.0.0:superstubs") {
  transitive = false
}
```

Then the specified locations on your classpath get scanned. For `com.example:beer-api-producer-restdocs`, the following locations are scanned:

- `/META-INF/com.example/beer-api-producer-restdocs/*/*`
- `/contracts/com.example/beer-api-producer-restdocs/*/*`
- `/mappings/com.example/beer-api-producer-restdocs/*/*`

For `com.example.thing1:thing2`, the following locations are scanned:

- `/META-INF/com.example.thing1/thing2/*/*`
- `/contracts/com.example.thing1/thing2/*/*`
- `/mappings/com.example.thing1/thing2/*/*`



You have to explicitly provide the group and artifact IDs when you package the producer stubs.

To achieve proper stub packaging, the producer would set up the contracts as follows:

```
├── src
│   ├── test
│   │   ├── resources
│   │   │   ├── contracts
│   │   │   │   ├── com.example
│   │   │   │   │   ├── beer-api-producer-restdocs
│   │   │   │   │   │   ├── nested
│   │   │   │   │   │   │   └── contract3.groovy
```

By using the [Maven assembly plugin](#) or the [Gradle Jar](#) task, you have to create the following structure in your stubs jar:

```
├── META-INF
│   ├── com.example
│   │   ├── beer-api-producer-restdocs
│   │   │   ├── 2.0.0
│   │   │   │   ├── contracts
│   │   │   │   │   ├── nested
│   │   │   │   │   │   └── contract2.groovy
│   │   │   ├── mappings
│   │   │   │   └── mapping.json
```

By maintaining this structure, the classpath gets scanned and you can profit from the messaging or HTTP stubs without the need to download artifacts.

Configuring HTTP Server Stubs

Stub Runner has a notion of a `HttpServerStub` that abstracts the underlying concrete implementation of the HTTP server (for example, WireMock is one of the implementations). Sometimes, you need to perform some additional tuning (which is concrete for the given implementation) of the stub servers. To do that, Stub Runner gives you the `httpServerStubConfigurer` property that is available in the annotation and the JUnit rule and is accessible through system properties, where you can provide your implementation of the

`org.springframework.cloud.contract.stubrunner.HttpServerStubConfigurer` interface. The implementations can alter the configuration files for the given HTTP server stub.

Spring Cloud Contract Stub Runner comes with an implementation that you can extend for WireMock:

`org.springframework.cloud.contract.stubrunner.provider.wiremock.WireMockHttpServerStubConfigurer`. In the `configure` method, you can provide your own custom configuration for the given stub. The use case might be starting WireMock for the given artifact ID, on an HTTPS port. The following example shows how to do so:

Example 1. WireMockHttpServerStubConfigurer implementation

```
@CompileStatic
static class HttpsForFraudDetection extends WireMockHttpServerStubConfigurer {

    private static final Log log = LogFactory.getLog(HttpsForFraudDetection)

    @Override
    WireMockConfiguration configure(WireMockConfiguration httpStubConfiguration,
HttpServerStubConfiguration httpServerStubConfiguration) {
        if (httpServerStubConfiguration.stubConfiguration.artifactId ==
"fraudDetectionServer") {
            int httpsPort = TestSocketUtils.findAvailableTcpPort()
            log.info("Will set HTTPS port [" + httpsPort + "] for fraud detection
server")
            return httpStubConfiguration
                .httpsPort(httpsPort)
        }
        return httpStubConfiguration
    }
}
```

You can then reuse it with the `@AutoConfigureStubRunner` annotation, as follows:

```
@AutoConfigureStubRunner(mappingsOutputFolder = "target/outputmappings/",
    httpServerStubConfigurer = HttpsForFraudDetection)
```

Whenever an HTTPS port is found, it takes precedence over the HTTP port.

Running stubs

This section describes how to run stubs. It contains the following topics:

- [HTTP Stubs](#)
- [Viewing Registered Mappings](#)

- [Messaging Stubs](#)

HTTP Stubs

Stubs are defined in JSON documents, whose syntax is defined in the [WireMock documentation](#).

The following example defines a stub in JSON:

```
{
  "request": {
    "method": "GET",
    "url": "/ping"
  },
  "response": {
    "status": 200,
    "body": "pong",
    "headers": {
      "Content-Type": "text/plain"
    }
  }
}
```

Viewing Registered Mappings

Every stubbed collaborator exposes a list of defined mappings under the `__/admin/` endpoint.

You can also use the `mappingsOutputFolder` property to dump the mappings to files. For the annotation-based approach, it would resemble the following example:

```
@AutoConfigureStubRunner(ids="a.b.c:loanIssuance,a.b.c:fraudDetectionServer",
mappingsOutputFolder = "target/outputmappings/")
```

For the JUnit approach, it resembles the following example:

```
@ClassRule @Shared StubRunnerRule rule = new StubRunnerRule()
    .repoRoot("https://some_url")
    .downloadStub("a.b.c", "loanIssuance")
    .downloadStub("a.b.c:fraudDetectionServer")
    .withMappingsOutputFolder("target/outputmappings")
```

Then, if you check out the `target/outputmappings` folder, you would see the following structure;

```
.
├── fraudDetectionServer_13705
└── loanIssuance_12255
```

That means that there were two stubs registered. `fraudDetectionServer` was registered at port `13705` and `loanIssuance` at port `12255`. If we take a look at one of the files, we would see (for WireMock) the mappings available for the given server:

```
[{
  "id" : "f9152eb9-bf77-4c38-8289-90be7d10d0d7",
  "request" : {
    "url" : "/name",
    "method" : "GET"
  },
  "response" : {
    "status" : 200,
    "body" : "fraudDetectionServer"
  },
  "uuid" : "f9152eb9-bf77-4c38-8289-90be7d10d0d7"
},
...
]
```

Messaging Stubs

Depending on the provided Stub Runner dependency and the DSL, the messaging routes are automatically set up.

3.5.4. Stub Runner JUnit Rule and Stub Runner JUnit5 Extension

Stub Runner comes with a JUnit rule that lets you can download and run stubs for a given group and artifact ID, as the following example shows:

```

@ClassRule
public static StubRunnerRule rule = new StubRunnerRule().repoRoot(repoRoot())
    .stubsMode(StubRunnerProperties.StubsMode.REMOTE)
    .downloadStub("org.springframework.cloud.contract.verifier.stubs",
"loanIssuance")

    .downloadStub("org.springframework.cloud.contract.verifier.stubs:fraudDetectionSer
ver");

@BeforeClass
@AfterClass
public static void setupProps() {
    System.clearProperty("stubrunner.repository.root");
    System.clearProperty("stubrunner.classifier");
}

```

A `StubRunnerExtension` is also available for JUnit 5. `StubRunnerRule` and `StubRunnerExtension` work in a very similar fashion. After the rule or extension is called, Stub Runner connects to your Maven repository and, for the given list of dependencies, tries to:

- Download them
- Cache them locally
- Unzip them to a temporary folder
- Start a WireMock server for each Maven dependency on a random port from the provided range of ports or the provided port
- Feed the WireMock server with all JSON files that are valid WireMock definitions
- Send messages (remember to pass an implementation of `MessageVerifierSender` interface)

Stub Runner uses the [Eclipse Aether](#) mechanism to download the Maven dependencies. Check their [docs](#) for more information.

Since the `StubRunnerRule` and `StubRunnerExtension` implement the `StubFinder`, they let you find the started stubs, as the following example shows:

```

package org.springframework.cloud.contract.stubrunner;

import java.net.URL;
import java.util.Collection;
import java.util.Map;

import org.springframework.cloud.contract.spec.Contract;

/**
 * Contract for finding registered stubs.
 *
 * @author Marcin Grzejszczak
 */
public interface StubFinder extends StubTrigger {

    /**
     * For the given groupId and artifactId tries to find the matching URL of the
     running
     * stub.
     * @param groupId - might be null. In that case a search only via artifactId
     takes
     * place
     * @param artifactId - artifact id of the stub
     * @return URL of a running stub or throws exception if not found
     * @throws StubNotFoundException in case of not finding a stub
     */
    URL findStubUrl(String groupId, String artifactId) throws
    StubNotFoundException;

    /**
     * For the given Ivy notation {@code
 [groupId]:artifactId:[version]:[classifier]}
     * tries to find the matching URL of the running stub. You can also pass only
     * {@code artifactId}.
     * @param ivyNotation - Ivy representation of the Maven artifact
     * @return URL of a running stub or throws exception if not found
     * @throws StubNotFoundException in case of not finding a stub
     */
    URL findStubUrl(String ivyNotation) throws StubNotFoundException;

    /**
     * @return all running stubs
     */
    RunningStubs findAllRunningStubs();

    /**
     * @return the list of Contracts
     */
    Map<StubConfiguration, Collection<Contract>> getContracts();

```

```
}
```

The following examples provide more detail about using Stub Runner:

Spock

```
@ClassRule
@Shared
StubRunnerRule rule = new StubRunnerRule()
    .stubsMode(StubRunnerProperties.StubsMode.REMOTE)

.repoRoot(StubRunnerRuleSpec.getResource("/m2repo/repository").toURI().toString())
    .downloadStub("org.springframework.cloud.contract.verifier.stubs",
"loanIssuance")

.downloadStub("org.springframework.cloud.contract.verifier.stubs:fraudDetectionServer")
    .withMappingsOutputFolder("target/outputmappingsforrule")

def 'should start WireMock servers'() {
    expect: 'WireMocks are running'
        rule.findStubUrl('org.springframework.cloud.contract.verifier.stubs',
'loanIssuance') != null
        rule.findStubUrl('loanIssuance') != null
        rule.findStubUrl('loanIssuance') ==
rule.findStubUrl('org.springframework.cloud.contract.verifier.stubs',
'loanIssuance')

rule.findStubUrl('org.springframework.cloud.contract.verifier.stubs:fraudDetectionServer') != null
    and:
        rule.findAllRunningStubs().isPresent('loanIssuance')

rule.findAllRunningStubs().isPresent('org.springframework.cloud.contract.verifier.stubs', 'fraudDetectionServer')

rule.findAllRunningStubs().isPresent('org.springframework.cloud.contract.verifier.stubs:fraudDetectionServer')
    and: 'Stubs were registered'
        "${rule.findStubUrl('loanIssuance').toString()}/name".toURL().text ==
'loanIssuance'
        "${rule.findStubUrl('fraudDetectionServer').toString()}/name".toURL().text
== 'fraudDetectionServer'
}

def 'should output mappings to output folder'() {
    when:
        def url = rule.findStubUrl('fraudDetectionServer')
    then:
        new File("target/outputmappingsforrule",
"fraudDetectionServer_${url.port}").exists()
}
```

JUnit 4

```
@Test
public void should_start_wiremock_servers() throws Exception {
    // expect: 'WireMocks are running'
    then(rule.findStubUrl("org.springframework.cloud.contract.verifier.stubs",
"loanIssuance")).isNotNull();
    then(rule.findStubUrl("loanIssuance")).isNotNull();
    then(rule.findStubUrl("loanIssuance"))

.isEqualTo(rule.findStubUrl("org.springframework.cloud.contract.verifier.stubs",
"loanIssuance"));

then(rule.findStubUrl("org.springframework.cloud.contract.verifier.stubs:fraudDe
tctionServer")).isNotNull();
    // and:
    then(rule.findAllRunningStubs().isPresent("loanIssuance")).isTrue();

then(rule.findAllRunningStubs().isPresent("org.springframework.cloud.contract.veri
fier.stubs",
    "fraudDetectionServer")).isTrue();
    then(rule.findAllRunningStubs()

.isPresent("org.springframework.cloud.contract.verifier.stubs:fraudDetectionServer
")).isTrue();
    // and: 'Stubs were registered'
    then(httpGet(rule.findStubUrl("loanIssuance").toString() +
"/name")).isEqualTo("loanIssuance");
    then(httpGet(rule.findStubUrl("fraudDetectionServer").toString() +
"/name")).isEqualTo("fraudDetectionServer");
}
```

JUnit 5

```
// Visible for JUnit
@RegisterExtension
static StubRunnerExtension stubRunnerExtension = new
StubRunnerExtension().repoRoot(repoRoot())
    .stubsMode(StubRunnerProperties.StubsMode.REMOTE)
    .downloadStub("org.springframework.cloud.contract.verifier.stubs",
"loanIssuance")

.downloadStub("org.springframework.cloud.contract.verifier.stubs:fraudDetectionSer
ver")
    .withMappingsOutputFolder("target/outputmappingsforrule");

@BeforeAll
@AfterAll
static void setupProps() {
    System.clearProperty("stubrunner.repository.root");
    System.clearProperty("stubrunner.classifier");
}

private static String repoRoot() {
    try {
        return
StubRunnerRuleJUnitTest.class.getResource("/m2repo/repository/").toURI().toString(
);
    }
    catch (Exception e) {
        return "";
    }
}
```

See the [Common Properties for JUnit and Spring](#) for more information on how to apply global configuration of Stub Runner.



To use the JUnit rule or JUnit 5 extension together with messaging, you have to provide an implementation of the `MessageVerifierSender` and `MessageVerifierReceiver` interface to the rule builder (for example, `rule.messageVerifierSender(new MyMessageVerifierSender())`). If you do not do this, then, whenever you try to send a message, an exception is thrown.

Maven Settings

The stub downloader honors Maven settings for a different local repository folder. Authentication details for repositories and profiles are currently not taken into account, so you need to specify it by using the properties mentioned above.

Providing Fixed Ports

You can also run your stubs on fixed ports. You can do it in two different ways. One is to pass it in the properties, and the other is to use the fluent API of JUnit rule.

Fluent API

When using the `StubRunnerRule` or `StubRunnerExtension`, you can add a stub to download and then pass the port for the last downloaded stub. The following example shows how to do so:

```
@ClassRule
public static StubRunnerRule rule = new StubRunnerRule().repoRoot(repoRoot())
    .stubsMode(StubRunnerProperties.StubsMode.REMOTE)
    .downloadStub("org.springframework.cloud.contract.verifier.stubs",
"loanIssuance").withPort(35465)

.downloadStub("org.springframework.cloud.contract.verifier.stubs:fraudDetectionServer:35466");

@BeforeClass
@AfterClass
public static void setupProps() {
    System.clearProperty("stubrunner.repository.root");
    System.clearProperty("stubrunner.classifier");
}
```

For the preceding example, the following test is valid:

```
then(rule.findStubUrl("loanIssuance")).isEqualTo(URI.create("http://localhost:35465").toURL());
then(rule.findStubUrl("fraudDetectionServer")).isEqualTo(URI.create("http://localhost:35466").toURL());
```

Stub Runner with Spring

Stub Runner with Spring sets up Spring configuration of the Stub Runner project.

By providing a list of stubs inside your configuration file, Stub Runner automatically downloads and registers in WireMock the selected stubs.

If you want to find the URL of your stubbed dependency, you can autowire the `StubFinder` interface and use its methods, as follows:

```

@SpringBootTest(classes = Config, properties = [" stubrunner.cloud.enabled=false",
    'foo=${stubrunner.runningstubs.fraudDetectionServer.port}',

    'fooWithGroup=${stubrunner.runningstubs.org.springframework.cloud.contract.verifier.stubs.fraudDetectionServer.port}'])
@AutoConfigureStubRunner(mappingsOutputFolder = "target/outputmappings/",
    httpServerStubConfigurer = HttpsForFraudDetection)
@ActiveProfiles("test")
class StubRunnerConfigurationSpec {

    @Autowired
    StubFinder stubFinder
    @Autowired
    Environment environment
    @StubRunnerPort("fraudDetectionServer")
    int fraudDetectionServerPort

    @StubRunnerPort("org.springframework.cloud.contract.verifier.stubs:fraudDetectionServer")
    int fraudDetectionServerPortWithGroupId
    @Value('${foo}')
    Integer foo

    @BeforeAll
    static void setupSpec() {
        System.clearProperty("stubrunner.repository.root")
        System.clearProperty("stubrunner.classifier")
        WireMockHttpServerStubAccessor.clear()
    }

    @AfterAll
    static void cleanupSpec() {
        setupSpec()
    }

    @Test
    void 'should mark all ports as random'() {
        expect:
        WireMockHttpServerStubAccessor.everyPortRandom()
    }

    @Test
    void 'should start WireMock servers'() {
        expect: 'WireMocks are running'
        assert
        stubFinder.findStubUrl('org.springframework.cloud.contract.verifier.stubs',
            'loanIssuance') != null
        assert stubFinder.findStubUrl('loanIssuance') != null
        assert stubFinder.findStubUrl('loanIssuance') ==

```

```

stubFinder.findStubUrl('org.springframework.cloud.contract.verifier.stubs',
'loanIssuance')
    assert stubFinder.findStubUrl('loanIssuance') ==
stubFinder.findStubUrl('org.springframework.cloud.contract.verifier.stubs:loanIssuance')
    assert
stubFinder.findStubUrl('org.springframework.cloud.contract.verifier.stubs:loanIssuance:0.0.1-SNAPSHOT') ==
stubFinder.findStubUrl('org.springframework.cloud.contract.verifier.stubs:loanIssuance:0.0.1-SNAPSHOT:stubs')
    assert
stubFinder.findStubUrl('org.springframework.cloud.contract.verifier.stubs:fraudDetectionServer') != null
    and:
    assert stubFinder.findAllRunningStubs().isPresent('loanIssuance')
    assert
stubFinder.findAllRunningStubs().isPresent('org.springframework.cloud.contract.verifier.stubs', 'fraudDetectionServer')
    assert
stubFinder.findAllRunningStubs().isPresent('org.springframework.cloud.contract.verifier.stubs:fraudDetectionServer')
    and: 'Stubs were registered'
    assert
"${stubFinder.findStubUrl('loanIssuance').toString()}/name".toURL().text ==
'loanIssuance'
    assert
"${stubFinder.findStubUrl('fraudDetectionServer').toString()}/name".toURL().text ==
'fraudDetectionServer'
    and: 'Fraud Detection is an HTTPS endpoint'
    assert
stubFinder.findStubUrl('fraudDetectionServer').toString().startsWith("https")
}

@Test
void 'should throw an exception when stub is not found'() {
    when:
        BDDAssertions.thenThrownBy(() ->
stubFinder.findStubUrl('nonExistingService')).isInstanceOf(StubNotFoundException)
    when:
        BDDAssertions.thenThrownBy(() ->
stubFinder.findStubUrl('nonExistingGroupId', 'nonExistingArtifactId'))
            .isInstanceOf(StubNotFoundException)
}

@Test
void 'should register started servers as environment variables'() {
    expect:
    assert
environment.getProperty("stubrunner.runningstubs.loanIssuance.port") != null
    assert stubFinder.findAllRunningStubs().getPort("loanIssuance") ==
(environment.getProperty("stubrunner.runningstubs.loanIssuance.port") as Integer)
}

```

```

        and:
        assert
environment.getProperty("stubrunner.runningstubs.fraudDetectionServer.port") !=
null
        assert stubFinder.findAllRunningStubs().getPort("fraudDetectionServer") ==
(environment.getProperty("stubrunner.runningstubs.fraudDetectionServer.port") as
Integer)
        and:
        assert
environment.getProperty("stubrunner.runningstubs.fraudDetectionServer.port") !=
null
        assert stubFinder.findAllRunningStubs().getPort("fraudDetectionServer") ==
(environment.getProperty("stubrunner.runningstubs.org.springframework.cloud.contra
ct.verifier.stubs.fraudDetectionServer.port") as Integer)
    }

    @Test
    void 'should be able to interpolate a running stub in the passed test
property'() {
        given:
        int fraudPort =
stubFinder.findAllRunningStubs().getPort("fraudDetectionServer")
        expect:
        assert fraudPort > 0
        assert environment.getProperty("foo", Integer) == fraudPort
        assert environment.getProperty("fooWithGroup", Integer) == fraudPort
        assert foo == fraudPort
    }

    // @Issue("#573")
    @Test
    void 'should be able to retrieve the port of a running stub via an
annotation'() {
        given:
        int fraudPort =
stubFinder.findAllRunningStubs().getPort("fraudDetectionServer")
        expect:
        assert fraudPort > 0
        assert fraudDetectionServerPort == fraudPort
        assert fraudDetectionServerPortWithGroupId == fraudPort
    }

    @Test
    void 'should dump all mappings to a file'() {
        when:
        def url = stubFinder.findStubUrl("fraudDetectionServer")
        then:
        assert new File("target/outputmappings/",
"fraudDetectionServer_${url.port}").exists()
    }

```

```

@Configuration
@EnableAutoConfiguration
static class Config {}

@CompileStatic
static class HttpsForFraudDetection extends WireMockHttpServerStubConfigurer {

    private static final Log log = LogFactory.getLog(HttpsForFraudDetection)

    @Override
    WireMockConfiguration configure(WireMockConfiguration
httpStubConfiguration, HttpServerStubConfiguration httpServerStubConfiguration) {
        if (httpServerStubConfiguration.stubConfiguration.artifactId ==
"fraudDetectionServer") {
            int httpsPort = TestSocketUtils.findAvailableTcpPort()
            log.info("Will set HTTPS port [" + httpsPort + "] for fraud
detection server")
            return httpStubConfiguration
                .httpsPort(httpsPort)
        }
        return httpStubConfiguration
    }
}
}
}

```

Doing so depends on the following configuration file:

```

stubrunner:
  repositoryRoot: classpath:m2repo/repository/
  ids:
    - org.springframework.cloud.contract.verifier.stubs:loanIssuance
    - org.springframework.cloud.contract.verifier.stubs:fraudDetectionServer
    - org.springframework.cloud.contract.verifier.stubs:bootService
  stubs-mode: remote

```

Instead of using the properties, you can also use the properties inside the `@AutoConfigureStubRunner`. The following example achieves the same result by setting values on the annotation:

```

@AutoConfigureStubRunner(ids =
["org.springframework.cloud.contract.verifier.stubs:loanIssuance",
"org.springframework.cloud.contract.verifier.stubs:fraudDetectionServer",
"org.springframework.cloud.contract.verifier.stubs:bootService"] ,
stubsMode = StubRunnerProperties.StubsMode.REMOTE ,
repositoryRoot = "classpath:m2repo/repository/" )

```


Stub Runner Spring registers environment variables in the following manner for every registered WireMock server. The following example shows Stub Runner IDs for `com.example:thing1` and `com.example:thing2`:

- `stubrunner.runningstubs.thing1.port`
- `stubrunner.runningstubs.com.example.thing1.port`
- `stubrunner.runningstubs.thing2.port`
- `stubrunner.runningstubs.com.example.thing2.port`

You can reference these values in your code.

You can also use the `@StubRunnerPort` annotation to inject the port of a running stub. The value of the annotation can be the `groupid:artifactid` or only the `artifactid`. The following example works shows Stub Runner IDs for `com.example:thing1` and `com.example:thing2`.

```
@StubRunnerPort("thing1")
int thing1Port;
@StubRunnerPort("com.example:thing2")
int thing2Port;
```

3.5.5. Stub Runner Spring Cloud

Stub Runner can integrate with Spring Cloud.

For real life examples, see:

- [The producer application sample](#)
- [The consumer application sample](#)

Stubbing Service Discovery

The most important feature of `Stub Runner Spring Cloud` is the fact that it stubs:

- `DiscoveryClient`
- `ReactorServiceInstanceLoadBalancer`

That means that, regardless of whether you use Zookeeper, Consul, Eureka, or anything else, you do not need that in your tests. We are starting WireMock instances of your dependencies and we are telling your application, whenever you use `Feign`, to load a balanced `RestTemplate` or `DiscoveryClient` directly, to call those stubbed servers instead of calling the real Service Discovery tool.

Test Profiles and Service Discovery

In your integration tests, you typically do not want to call either a discovery service (such as Eureka) or Config Server. That is why you create an additional test configuration in which you want

to disable these features.

Due to certain limitations of `spring-cloud-commons`, to achieve this, you have to disable these properties in a static block such as the following example (for Eureka):

```
//Hack to work around https://github.com/spring-cloud/spring-cloud-  
commons/issues/156  
static {  
    System.setProperty("eureka.client.enabled", "false");  
    System.setProperty("spring.cloud.config.failFast", "false");  
}
```

Additional Configuration

You can match the `artifactId` of the stub with the name of your application by using the `stubrunner.idsToServiceIds`: map.



By default, all service discovery is stubbed. This means that, regardless of whether you have an existing `DiscoveryClient`, its results are ignored. However, if you want to reuse it, you can set `stubrunner.cloud.delegate.enabled` to `true`, and then your existing `DiscoveryClient` results are merged with the stubbed ones.

The default Maven configuration used by Stub Runner can be tweaked either by setting the following system properties or by setting the corresponding environment variables:

- `maven.repo.local`: Path to the custom maven local repository location
- `org.apache.maven.user-settings`: Path to custom maven user settings location
- `org.apache.maven.global-settings`: Path to maven global settings location

3.5.6. Using the Stub Runner Boot Application

Spring Cloud Contract Stub Runner Boot is a Spring Boot application that exposes REST endpoints to trigger the messaging labels and to access WireMock servers.

Stub Runner Boot Security

The Stub Runner Boot application is not secured by design - securing it would require to add security to all stubs even if they don't actually require it. Since this is a testing utility - the server is **not intended** to be used in production environments.



It is expected that **only a trusted client** has access to the Stub Runner Boot server. You should not run this application as a Fat Jar or a [Docker Image](#) in untrusted locations.

Stub Runner Server

To use the Stub Runner Server, add the following dependency:

```
compile "org.springframework.cloud:spring-cloud-starter-stub-runner"
```

Then annotate a class with `@EnableStubRunnerServer`, build a fat jar, and it is ready to work.

For the properties, see the [Stub Runner Spring](#) section.

Stub Runner Server Fat Jar

You can download a standalone JAR from Maven (for example, for version 2.0.1.RELEASE) by running the following commands:

```
$ wget -O stub-runner.jar  
'https://search.maven.org/remotecontent?filepath=org/springframework/cloud/spring-  
cloud-contract-stub-runner-boot/2.0.1.RELEASE/spring-cloud-contract-stub-runner-  
boot-2.0.1.RELEASE.jar'  
$ java -jar stub-runner.jar --stubrunner.ids=... --stubrunner.repositoryRoot=...
```

Spring Cloud CLI

Starting from the 1.4.0.RELEASE version of the [Spring Cloud CLI](#) project, you can start Stub Runner Boot by running `spring cloud stubrunner`.

To pass the configuration, you can create a `stubrunner.yml` file in the current working directory, in a subdirectory called `config`, or in `~/.spring-cloud`. The file could resemble the following example for running stubs installed locally:

Example 2. stubrunner.yml

```
stubrunner:  
  stubsMode: LOCAL  
  ids:  
    - com.example:beer-api-producer:+:9876
```

Then you can call `spring cloud stubrunner` from your terminal window to start the Stub Runner server. It is available at port 8750.

Endpoints

Stub Runner Boot offers two endpoints:

- [HTTP](#)
- [Messaging](#)

HTTP

For HTTP, Stub Runner Boot makes the following endpoints available:

- GET `/stubs`: Returns a list of all running stubs in `ivy:integer` notation
- GET `/stubs/{ivy}`: Returns a port for the given `ivy` notation (when calling the endpoint `ivy` can also be `artifactId` only)

Messaging

For Messaging, Stub Runner Boot makes the following endpoints available:

- GET `/triggers`: Returns a list of all running labels in `ivy : [label1, label2 ...]` notation
- POST `/triggers/{label}`: Runs a trigger with `label`
- POST `/triggers/{ivy}/{label}`: Runs a trigger with a `label` for the given `ivy` notation (when calling the endpoint, `ivy` can also be `artifactId` only)

Example

The following example shows typical usage of Stub Runner Boot:

```
@SpringBootTest(classes = StubRunnerBoot, properties =
"spring.cloud.zookeeper.enabled=false")
@ActiveProfiles("test")
class StubRunnerBootSpec {

    @Autowired
    StubRunning stubRunning

    @BeforeEach
    void setup() {
        RestAssuredMockMvc.standaloneSetup(new HttpStubsController(stubRunning),
            new TriggerController(stubRunning))
    }

    @Test
    void 'should return a list of running stub servers in "full ivy port" notation'()
    {
        when:
            String response = RestAssuredMockMvc.get('/stubs').body.asString()
        then:
            def root = new JsonSlurper().parseText(response)
            assert
            root.'org.springframework.cloud.contract.verifier.stubs:bootService:0.0.1-
            SNAPSHOT:stubs' instanceof Integer
    }
}
```

```

@Test
void 'should return a port on which a #stubId stub is running'() {
    given:
        def stubIds =
['org.springframework.cloud.contract.verifier.stubs:bootService:++stubs',
'org.springframework.cloud.contract.verifier.stubs:bootService:0.0.1-SNAPSHOT:stubs',
        'org.springframework.cloud.contract.verifier.stubs:bootService:+',
        'org.springframework.cloud.contract.verifier.stubs:bootService',
        'bootService']
    stubIds.each {
        when:
            def response = RestAssuredMockMvc.get("/stubs/${it}")
        then:
            assert response.statusCode == 200
            assert Integer.valueOf(response.body.asString()) > 0
        }
    }
}

@Test
void 'should return 404 when missing stub was called'() {
    when:
        def response = RestAssuredMockMvc.get("/stubs/a:b:c:d")
    then:
        assert response.statusCode == 404
    }
}

@Test
void 'should return a list of messaging labels that can be triggered when version
and classifier are passed'() {
    when:
        String response = RestAssuredMockMvc.get('/triggers').body.asString()
    then:
        def root = new JsonSlurper().parseText(response)
        assert
root.'org.springframework.cloud.contract.verifier.stubs:bootService:0.0.1-
SNAPSHOT:stubs'?.containsAll(["return_book_1"])
    }
}

@Test
void 'should trigger a messaging label'() {
    given:
        StubRunning stubRunning = Mockito.mock(StubRunning)
        RestAssuredMockMvc.standaloneSetup(new HttpStubsController(stubRunning),
new TriggerController(stubRunning))
    when:
        def response = RestAssuredMockMvc.post("/triggers/delete_book")
    then:
        response.statusCode == 200
    and:

```

```

        Mockito.verify(stubRunning).trigger('delete_book')
    }

    @Test
    void 'should trigger a messaging label for a stub with #stubId ivy notation'() {
        given:
            StubRunning stubRunning = Mockito.mock(StubRunning)
            RestAssuredMockMvc.standaloneSetup(new HttpStubsController(stubRunning),
new TriggerController(stubRunning))
            and:
                def stubIds =
['org.springframework.cloud.contract.verifier.stubs:bootService:stubs',
'org.springframework.cloud.contract.verifier.stubs:bootService', 'bootService']
                stubIds.each {
                    when:
                        def response = RestAssuredMockMvc.post("/triggers/$it/delete_book")
                    then:
                        assert response.statusCode == 200
                    and:
                        Mockito.verify(stubRunning).trigger(it, 'delete_book')
                }
            }

    @Test
    void 'should throw exception when trigger is missing'() {
        when:
            BDDAssertions.thenThrownBy(() ->
RestAssuredMockMvc.post("/triggers/missing_label"))
                .hasMessageContaining("Exception occurred while trying to return
[missing_label] label.")
                .hasMessageContaining("Available labels are")

                .hasMessageContaining("org.springframework.cloud.contract.verifier.stubs:loanIssuance:
0.0.1-SNAPSHOT:stubs=[]")

                .hasMessageContaining("org.springframework.cloud.contract.verifier.stubs:bootService:0
.0.1-SNAPSHOT:stubs=")
            }
    }
}

```

Stub Runner Boot with Service Discovery

One way to use Stub Runner Boot is to use it as a feed of stubs for “smoke tests”. What does that mean? Assume that you do not want to deploy 50 microservices to a test environment in order to see whether your application works. You have already run a suite of tests during the build process, but you would also like to ensure that the packaging of your application works. You can deploy your application to an environment, start it, and run a couple of tests on it to see whether it works. We can call those tests “smoke tests”, because their purpose is to check only a handful of testing

scenarios.

The problem with this approach is that, if you use microservices, you most likely also use a service discovery tool. Stub Runner Boot lets you solve this issue by starting the required stubs and registering them in a service discovery tool.

Now assume that we want to start this application so that the stubs get automatically registered. We can do so by running the application with `java -jar ${SYSTEM_PROPS} stub-runner-boot-eureka-example.jar`, where `${SYSTEM_PROPS}`.

That way, your deployed application can send requests to started WireMock servers through service discovery. Most likely, points 1 through 3 could be set by default in `application.yml`, because they are not likely to change. That way, you can provide only the list of stubs to download whenever you start the Stub Runner Boot.

3.5.7. Consumer-Driven Contracts: Stubs Per Consumer

There are cases in which two consumers of the same endpoint want to have two different responses.



This approach also lets you immediately know which consumer uses which part of your API. You can remove part of a response that your API produces and see which of your autogenerated tests fails. If none fails, you can safely delete that part of the response, because nobody uses it.

Consider the following example of a contract defined for the producer called `producer`, which has two consumers (`foo-consumer` and `bar-consumer`):

Consumer `foo-service`

```
request {
  url '/foo'
  method GET()
}
response {
  status OK()
  body(
    foo: "foo"
  )
}
```

Consumer `bar-service`

```
request {
  url '/bar'
  method GET()
}
response {
  status OK()
  body(
    bar: "bar"
  )
}
```

You cannot produce two different responses for the same request. That is why you can properly package the contracts and then profit from the `stubsPerConsumer` feature.

On the producer side, the consumers can have a folder that contains contracts related only to them. By setting the `stubrunner.stubs-per-consumer` flag to `true`, we no longer register all stubs but only those that correspond to the consumer application's name. In other words, we scan the path of every stub and, if it contains a subfolder with name of the consumer in the path, only then is it registered.

On the `foo` producer side the contracts would look like this

```
.
├── contracts
│   ├── bar-consumer
│   │   ├── bookReturnedForBar.groovy
│   │   └── shouldCallBar.groovy
│   └── foo-consumer
│       ├── bookReturnedForFoo.groovy
│       └── shouldCallFoo.groovy
```

The `bar-consumer` consumer can either set the `spring.application.name` or the `stubrunner.consumer-`

name to `bar-consumer` Alternatively, you can set the test as follows:

```
@SpringBootTest(classes = Config, properties = ["spring.application.name=bar-
consumer"])
@AutoConfigureStubRunner(ids =
"org.springframework.cloud.contract.verifier.stubs:producerWithMultipleConsumers",
    repositoryRoot = "classpath:m2repo/repository/",
    stubsMode = StubRunnerProperties.StubsMode.REMOTE,
    stubsPerConsumer = true)
@ActiveProfiles("streamconsumer")
class StubRunnerStubsPerConsumerSpec {
    ...
}
```

Then only the stubs registered under a path that contains `bar-consumer` in its name (that is, those from the `src/test/resources/contracts/bar-consumer/some/contracts/...` folder) are allowed to be referenced.

You can also set the consumer name explicitly, as follows:

```
@SpringBootTest(classes = Config)
@AutoConfigureStubRunner(ids =
"org.springframework.cloud.contract.verifier.stubs:producerWithMultipleConsumers",
    repositoryRoot = "classpath:m2repo/repository/",
    consumerName = "foo-consumer",
    stubsMode = StubRunnerProperties.StubsMode.REMOTE,
    stubsPerConsumer = true)
@ActiveProfiles("streamconsumer")
class StubRunnerStubsPerConsumerWithConsumerNameSpec {
    ...
}
```

Then only the stubs registered under a path that contains the `foo-consumer` in its name (that is, those from the `src/test/resources/contracts/foo-consumer/some/contracts/...` folder) are allowed to be referenced.

For more information about the reasons behind this change, see [issue 224](#).

3.5.8. Fetching Stubs or Contract Definitions From A Location

Instead of picking the stubs or contract definitions from Artifactory, Nexus, or Git, you can point to a location on a drive or the classpath. Doing so can be especially useful in a multi-module project, where one module wants to reuse stubs or contracts from another module without the need to actually install those in a local maven repository to commit those changes to Git.

In order to achieve this, you can use the `stubs://` protocol when the repository root parameter is set either in Stub Runner or in a Spring Cloud Contract plugin.

In this example, the `producer` project has been successfully built and stubs were generated under the `target/stubs` folder. As a consumer, one can set up the Stub Runner to pick the stubs from that location by using the `stubs://` protocol.

Annotation

```
@AutoConfigureStubRunner(  
    stubsMode = StubRunnerProperties.StubsMode.REMOTE,  
    repositoryRoot = "stubs://file://location/to/the/producer/target/stubs/",  
    ids = "com.example:some-producer")
```

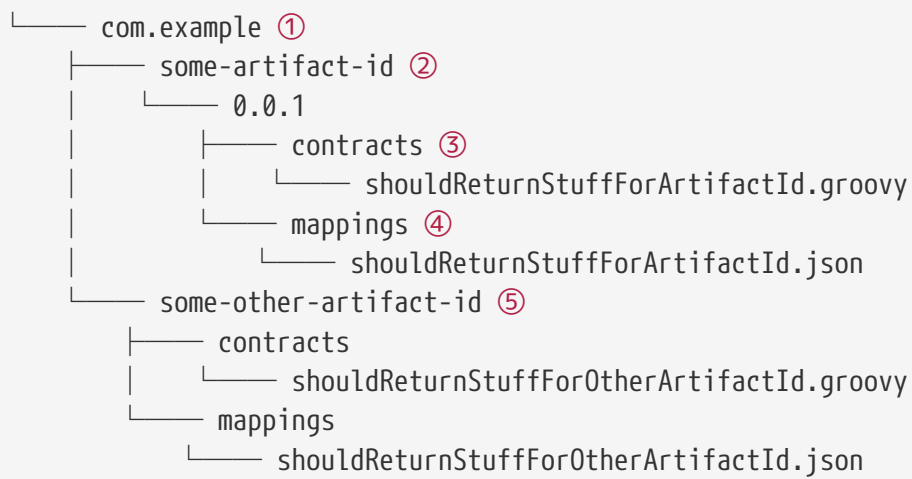
JUnit 4 Rule

```
@Rule  
public StubRunnerRule rule = new StubRunnerRule()  
    .downloadStub("com.example:some-producer")  
    .repoRoot("stubs://file://location/to/the/producer/target/stubs/")  
    .stubsMode(StubRunnerProperties.StubsMode.REMOTE);
```

JUnit 5 Extension

```
@RegisterExtension  
public StubRunnerExtension stubRunnerExtension = new StubRunnerExtension()  
    .downloadStub("com.example:some-producer")  
    .repoRoot("stubs://file://location/to/the/producer/target/stubs/")  
    .stubsMode(StubRunnerProperties.StubsMode.REMOTE);
```

Contracts and stubs may be stored in a location, where each producer has its own, dedicated folder for contracts and stub mappings. Under that folder, each consumer can have its own setup. To make Stub Runner find the dedicated folder from the provided IDs, you can pass the `stubs.find-producer=true` property or the `stubrunner.stubs.find-producer=true` system property. The following listing shows an arrangement of contracts and stubs:



① group ID of the consumers

② consumer with artifact id [some-artifact-id]

③ contracts for the consumer with artifact id [some-artifact-id]

④ mappings for the consumer with artifact id [some-artifact-id]

⑤ consumer with artifact id [some-other-artifact-id]

Annotation

```
@AutoConfigureStubRunner(  
    stubsMode = StubRunnerProperties.StubsMode.REMOTE,  
    repositoryRoot = "stubs://file://location/to/the/contracts/directory",  
    ids = "com.example:some-producer",  
    properties="stubs.find-producer=true")
```

JUnit 4 Rule

```
static Map<String, String> contractProperties() {  
    Map<String, String> map = new HashMap<>();  
    map.put("stubs.find-producer", "true");  
    return map;  
}  
  
@Rule  
public StubRunnerRule rule = new StubRunnerRule()  
    .downloadStub("com.example:some-producer")  
    .repoRoot("stubs://file://location/to/the/contracts/directory")  
    .stubsMode(StubRunnerProperties.StubsMode.REMOTE)  
    .properties(contractProperties());
```

JUnit 5 Extension

```
static Map<String, String> contractProperties() {  
    Map<String, String> map = new HashMap<>();  
    map.put("stubs.find-producer", "true");  
    return map;  
}  
  
@RegisterExtension  
public StubRunnerExtension stubRunnerExtension = new StubRunnerExtension()  
    .downloadStub("com.example:some-producer")  
    .repoRoot("stubs://file://location/to/the/contracts/directory")  
    .stubsMode(StubRunnerProperties.StubsMode.REMOTE)  
    .properties(contractProperties());
```

3.5.9. Generating Stubs at Runtime

As a consumer, you might not want to wait for the producer to finish its implementation and then publish their stubs. A solution to this problem can be generation of stubs at runtime.

As a producer, when a contract is defined, you are required to make the generated tests pass in order for the stubs to be published. There are cases where you would like to unblock the consumers so that they can fetch the stubs before your tests actually pass. In this case, you should set such contracts as in-progress. You can read more about this under the [Contracts in Progress](#) section. That

way, your tests are not generated, but the stubs are generated.

As a consumer, you can toggle a switch to generate stubs at runtime. Stub Runner ignores all the existing stub mappings and generates new ones for all the contract definitions. Another option is to pass the `stubrunner.generate-stubs` system property. The following example shows such a setup:

Annotation

```
@AutoConfigureStubRunner(  
    stubsMode = StubRunnerProperties.StubsMode.REMOTE,  
    repositoryRoot = "stubs://file://location/to/the/contracts",  
    ids = "com.example:some-producer",  
    generateStubs = true)
```

JUnit 4 Rule

```
@Rule  
public StubRunnerRule rule = new StubRunnerRule()  
    .downloadStub("com.example:some-producer")  
    .repoRoot("stubs://file://location/to/the/contracts")  
    .stubsMode(StubRunnerProperties.StubsMode.REMOTE)  
    .withGenerateStubs(true);
```

JUnit 5 Extension

```
@RegisterExtension  
public StubRunnerExtension stubRunnerExtension = new StubRunnerExtension()  
    .downloadStub("com.example:some-producer")  
    .repoRoot("stubs://file://location/to/the/contracts")  
    .stubsMode(StubRunnerProperties.StubsMode.REMOTE)  
    .withGenerateStubs(true);
```

3.5.10. Fail On No Stubs

By default, Stub Runner will fail if no stubs are found. In order to change that behavior, set the `failOnNoStubs` property to `false` in the annotation or call the `withFailOnNoStubs(false)` method on a JUnit Rule or Extension. The following example shows how to do so:

Annotation

```
@AutoConfigureStubRunner(  
    stubsMode = StubRunnerProperties.StubsMode.REMOTE,  
    repositoryRoot = "stubs://file://location/to/the/contracts",  
    ids = "com.example:some-producer",  
    failOnNoStubs = false)
```

JUnit 4 Rule

```
@Rule  
public StubRunnerRule rule = new StubRunnerRule()  
    .downloadStub("com.example:some-producer")  
    .repoRoot("stubs://file://location/to/the/contracts")  
    .stubsMode(StubRunnerProperties.StubsMode.REMOTE)  
    .withFailOnNoStubs(false);
```

JUnit 5 Extension

```
@RegisterExtension  
public StubRunnerExtension stubRunnerExtension = new StubRunnerExtension()  
    .downloadStub("com.example:some-producer")  
    .repoRoot("stubs://file://location/to/the/contracts")  
    .stubsMode(StubRunnerProperties.StubsMode.REMOTE)  
    .withFailOnNoStubs(false);
```

3.5.11. Common Properties

This section briefly describes common properties, including:

- [Common Properties for JUnit and Spring](#)
- [Stub Runner Stubs IDs](#)

Common Properties for JUnit and Spring

You can set repetitive properties by using system properties or Spring configuration properties. The following table shows their names with their default values:

Property name	Default value	Description
<code>stubrunner.minPort</code>	10000	Minimum value of a port for a started WireMock with stubs.
<code>stubrunner.maxPort</code>	15000	Maximum value of a port for a started WireMock with stubs.
<code>stubrunner.repositoryRoot</code>		Maven repository URL. If blank, then call the local Maven repo.

Property name	Default value	Description
<code>stubrunner.classifier</code>	<code>stubs</code>	Default classifier for the stub artifacts.
<code>stubrunner.stubsMode</code>	<code>CLASSPATH</code>	The way you want to fetch and register the stubs.
<code>stubrunner.ids</code>		Array of Ivy notation stubs to download.
<code>stubrunner.username</code>		Optional username to access the tool that stores the JARs with stubs.
<code>stubrunner.password</code>		Optional password to access the tool that stores the JARs with stubs.
<code>stubrunner.stubsPerConsumer</code>	<code>false</code>	Set to <code>true</code> if you want to use different stubs for each consumer instead of registering all stubs for every consumer.
<code>stubrunner.consumerName</code>		If you want to use a stub for each consumer and want to override the consumer name, change this value.

Stub Runner Stubs IDs

You can set the stubs to download in the `stubrunner.ids` system property. They use the following pattern:

```
groupId:artifactId:version:classifier:port
```

Note that `version`, `classifier`, and `port` are optional.

- If you do not provide the `port`, a random one is picked.
- If you do not provide the `classifier`, the default is used. (Note that you can pass an empty classifier this way: `groupId:artifactId:version:`).
- If you do not provide the `version`, then `+` is passed, and the latest one is downloaded.

`port` means the port of the WireMock server.



Starting with version 1.0.4, you can provide a range of versions that you would like the Stub Runner to take into consideration. You can read more about the Aether versioning ranges [here](#).

3.6. Spring Cloud Contract WireMock

The Spring Cloud Contract WireMock modules let you use [WireMock](#) in a Spring Boot application. For more detail, check out the [samples](#).

If you have a Spring Boot application that uses Tomcat as an embedded server (which is the default with `spring-boot-starter-web`), you can add `spring-cloud-starter-contract-stub-runner` to your classpath and add `@AutoConfigureWireMock` to use Wiremock in your tests. Wiremock runs as a stub server, and you can register stub behavior by using a Java API or by using static JSON declarations as part of your test. The following code shows an example:

```
@SpringBootTest(webEnvironment = WebEnvironment.RANDOM_PORT)
@AutoConfigureWireMock(port = 0)
public class WiremockForDocsTests {

    // A service that calls out over HTTP
    @Autowired
    private Service service;

    @BeforeEach
    public void setup() {
        this.service.setBase("http://localhost:"
            + this.environment.getProperty("wiremock.server.port"));
    }

    // Using the WireMock APIs in the normal way:
    @Test
    public void contextLoads() throws Exception {
        // Stubbing WireMock
        stubFor(get(urlEqualTo("/resource")).willReturn(aResponse()
            .withHeader("Content-Type", "text/plain").withBody("Hello
World!"))));
        // We're asserting if WireMock responded properly
        assertThat(this.service.go()).isEqualTo("Hello World!");
    }
}
```

To start the stub server on a different port, use (for example), `@AutoConfigureWireMock(port=9999)`. For a random port, use a value of `0`. The stub server port can be bound in the test application context with the `wiremock.server.port` property. Using `@AutoConfigureWireMock` adds a bean of type `WiremockConfiguration` to your test application context, where it is cached between methods and classes that have the same context. The same is true for Spring integration tests. Also, you can inject a bean of type `WireMockServer` into your test. The registered WireMock server is reset after each test class. However, if you need to reset it after each test method, set the `wiremock.reset-mappings-after-each-test` property to `true`.

3.6.1. Registering Stubs Automatically

If you use `@AutoConfigureWireMock`, it registers WireMock JSON stubs from the file system or classpath (by default, from `file:src/test/resources/mappings`). You can customize the locations by using the `stubs` attribute in the annotation, which can be an Ant-style resource pattern or a directory. In the case of a directory, `*/.json` is appended. The following code shows an example:

```
@RunWith(SpringRunner.class)
@SpringBootTest
@AutoConfigureWireMock(stubs="classpath:/stubs")
public class WiremockImportApplicationTests {

    @Autowired
    private Service service;

    @Test
    public void contextLoads() throws Exception {
        assertThat(this.service.go()).isEqualTo("Hello World!");
    }
}
```



Actually, WireMock always loads mappings from `src/test/resources/mappings` **as well as** the custom locations in the `stubs` attribute. To change this behavior, you can also specify a file root, as described in the [next section of this document](#).



Also, the mappings in the `stubs` location are not considered part of Wiremock's "default mappings" and calls to `com.github.tomakehurst.wiremock.client.WireMock.resetToDefaultMappings` during a test do not result in the mappings in the `stubs` location being included. However, the `org.springframework.cloud.contract.wiremock.WireMockTestExecutionListener` does reset the mappings (including adding the ones from the stubs location) after every test class and, optionally, after every test method (guarded by the `wiremock.reset-mappings-after-each-test` property).

If you use Spring Cloud Contract's default stub jars, your stubs are stored in the `/META-INF/group-id/artifact-id/versions/mappings/` folder. If you want to register all stubs from that location, from all embedded JARs, you can use the following syntax:

```
@AutoConfigureWireMock(port = 0, stubs = "classpath*/META-INF/**/mappings/**/*.*.json")
```

3.6.2. Using Files to Specify the Stub Bodies

WireMock can read response bodies from files on the classpath or the file system. In the case of the file system, you can see in the JSON DSL that the response has a `bodyFileName` instead of a (literal) `body`. The files are resolved relative to a root directory (by default, `src/test/resources/__files`). To customize this location, you can set the `files` attribute in the `@AutoConfigureWireMock` annotation to the location of the parent directory (in other words, `__files` is a subdirectory). You can use a Spring resource notation to refer to `file:...` or `classpath:...` locations. Generic URLs are not supported. A list of values can be given—in which case, WireMock resolves the first file that exists when it needs to find a response body.



When you configure the `files` root, it also affects the automatic loading of stubs, because they come from the root location in a subdirectory called `mappings`.



The value of `files` has no effect on the stubs loaded explicitly from the `stubs` attribute.

3.6.3. Alternative: Using JUnit Rules

For a more conventional WireMock experience, you can use JUnit `@Rules` to start and stop the server. To do so, use the `WireMockSpring` convenience class to obtain an `Options` instance, as the following example shows:

```

@SpringBootTest(webEnvironment = WebEnvironment.RANDOM_PORT)
public class WiremockForDocsClassRuleTests {

    // Start WireMock on some dynamic port
    // for some reason `dynamicPort()` is not working properly
    public static WireMockServer wiremock = new
WireMockServer(WireMockSpring.options().dynamicPort());

    @BeforeAll
    static void setupClass() {
        wiremock.start();
    }

    @AfterEach
    void after() {
        wiremock.resetAll();
    }

    @AfterAll
    static void clean() {
        wiremock.shutdown();
    }

    // A service that calls out over HTTP to wiremock's port
    @Autowired
    private Service service;

    @BeforeEach
    public void setup() {
        this.service.setBase("http://localhost:" + wiremock.port());
    }

    // Using the WireMock APIs in the normal way:
    @Test
    public void contextLoads() throws Exception {
        // Stubbing WireMock
        wiremock.stubFor(get(urlEqualTo("/resource")).willReturn(aResponse()
            .withHeader("Content-Type", "text/plain").withBody("Hello
World!"))));
        // We're asserting if WireMock responded properly
        assertThat(this.service.go()).isEqualTo("Hello World!");
    }
}

```

The `@ClassRule` means that the server shuts down after all the methods in this class have been run.

3.6.4. Relaxed SSL Validation for Rest Template

WireMock lets you stub a “secure” server with an `https` URL protocol. If your application wants to contact that stub server in an integration test, it finds that the SSL certificates are not valid (the usual problem with self-installed certificates). The best option is often to re-configure the client to use `http`. If that is not an option, you can ask Spring to configure an HTTP client that ignores SSL validation errors (do so only for tests, of course).

To make this work with minimum fuss, you need to use the Spring Boot `RestTemplateBuilder` in your application, as the following example shows:

```
@Bean
public RestTemplate restTemplate(RestTemplateBuilder builder) {
    return builder.build();
}
```

You need `RestTemplateBuilder` because the builder is passed through callbacks to initialize it, so the SSL validation can be set up in the client at that point. This happens automatically in your test if you use the `@AutoConfigureWireMock` annotation or the stub runner. If you use the JUnit `@Rule` approach, you need to add the `@AutoConfigureHttpClient` annotation as well, as the following example shows:

```
@RunWith(SpringRunner.class)
@SpringBootTest("app.baseUrl=https://localhost:6443")
@AutoConfigureHttpClient
public class WiremockHttpsServerApplicationTests {

    @ClassRule
    public static WireMockClassRule wiremock = new WireMockClassRule(
        WireMockSpring.options().httpsPort(6443));

    ...
}
```

If you use `spring-boot-starter-test`, you have the Apache HTTP client on the classpath, and it is selected by the `RestTemplateBuilder` and configured to ignore SSL errors. If you use the default `java.net` client, you do not need the annotation (but it does no harm). There is currently no support for other clients, but it may be added in future releases.

To disable the custom `RestTemplateBuilder`, set the `wiremock.rest-template-ssl-enabled` property to `false`.

3.6.5. WireMock and Spring MVC Mocks

Spring Cloud Contract provides a convenience class that can load JSON WireMock stubs into a

Spring `MockRestServiceServer`. The following code shows an example:

```
@SpringBootTest(webEnvironment = WebEnvironment.NONE)
public class WiremockForDocsMockServerApplicationTests {

    @Autowired
    private RestTemplate restTemplate;

    @Autowired
    private Service service;

    @Test
    public void contextLoads() throws Exception {
        // will read stubs classpath
        MockRestServiceServer server =
        WireMockRestServiceServer.with(this.restTemplate)

        .baseUrl("https://example.org").stubs("classpath:/stubs/resource.json")
            .build();
        // We're asserting if WireMock responded properly
        assertThat(this.service.go()).isEqualTo("Hello World");
        server.verify();
    }
}
```

The `baseUrl` value is prepended to all mock calls, and the `stubs()` method takes a stub path resource pattern as an argument. In the preceding example, the stub defined at `/stubs/resource.json` is loaded into the mock server. If the `RestTemplate` is asked to visit `example.org/`, it gets the responses as being declared at that URL. More than one stub pattern can be specified, and each one can be a directory (for a recursive list of all `.json`), a fixed filename (as in the preceding example), or an Ant-style pattern. The JSON format is the normal WireMock format, which you can read about at the [WireMock website](#).

Currently, the Spring Cloud Contract Verifier supports Tomcat, Jetty, and Undertow as Spring Boot embedded servers, and Wiremock itself has “native” support for a particular version of Jetty (currently 9.2). To use the native Jetty, you need to add the native Wiremock dependencies and exclude the Spring Boot container (if there is one).

3.7. Build Tools Integration

You can run test generation and stub invocation in various ways. The most common ones are as follows:

- [Maven](#)
- [Gradle](#)

- [Docker](#)

3.8. What to Read Next

If you want to learn more about any of the classes discussed in this section, you can browse the [source code directly](#). If you have specific questions, see the [how-to](#) section.

If you are comfortable with Spring Cloud Contract's core features, you can continue on and read about [Spring Cloud Contract's advanced features](#).

4. Maven Project

To learn how to set up the Maven project for Spring Cloud Contract Verifier, read the following sections:

- [Adding the Maven Plugin](#)
- [Maven and Rest Assured 2.0](#)
- [Using Snapshot and Milestone Versions for Maven](#)
- [Adding stubs](#)
- [Run Plugin](#)
- [Configure plugin](#)
- [Configuration Options](#)
- [Single Base Class for All Tests](#)
- [Using Different Base Classes for Contracts](#)
- [Invoking Generated Tests](#)
- [Pushing Stubs to SCM](#)
- [Maven Plugin and STS](#)

You can also check the plugin's documentation [here](#).

4.1. Adding the Maven Plugin

To add the Spring Cloud Contract BOM, include the following section in your `pom.xml` file:

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

Next, add the **Spring Cloud Contract Verifier** Maven plugin, as follows:

```
<plugin>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-contract-maven-plugin</artifactId>
  <version>${spring-cloud-contract.version}</version>
  <extensions>true</extensions>
  <configuration>

<packageWithBaseClasses>com.example.fraud</packageWithBaseClasses>
<!--           <convertToYaml>true</convertToYaml>-->
    </configuration>
</plugin>
```

You can read more in the [spring-cloud-contract-maven-plugin/index.html](https://docs.spring.io/spring-cloud-contract/maven-plugin/index.html) [Spring Cloud Contract Maven Plugin Documentation].

Sometimes, regardless of the picked IDE, you can see that the **target/generated-test-source** folder is not visible on the IDE's classpath. To ensure that it is always there, you can add the following entry to your **pom.xml**

```
<plugin>
  <groupId>org.codehaus.mojo</groupId>
  <artifactId>build-helper-maven-plugin</artifactId>
  <executions>
    <execution>
      <id>add-source</id>
      <phase>generate-test-sources</phase>
      <goals>
        <goal>add-test-source</goal>
      </goals>
      <configuration>
        <sources>
          <source>${project.build.directory}/generated-test-
sources/contracts</source>
        </sources>
      </configuration>
    </execution>
  </executions>
</plugin>
```

4.2. Maven and Rest Assured 2.0

By default, Rest Assured 3.x is added to the classpath. However, you can use Rest Assured 2.x by

adding it to the plugins classpath, as follows:

```
<plugin>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-contract-maven-plugin</artifactId>
  <version>${spring-cloud-contract.version}</version>
  <extensions>>true</extensions>
  <configuration>
    <packageWithBaseClasses>com.example</packageWithBaseClasses>
  </configuration>
  <dependencies>
    <dependency>
      <groupId>org.springframework.cloud</groupId>
      <artifactId>spring-cloud-contract-verifier</artifactId>
      <version>${spring-cloud-contract.version}</version>
    </dependency>
    <dependency>
      <groupId>com.jayway.restassured</groupId>
      <artifactId>rest-assured</artifactId>
      <version>2.5.0</version>
      <scope>compile</scope>
    </dependency>
    <dependency>
      <groupId>com.jayway.restassured</groupId>
      <artifactId>spring-mock-mvc</artifactId>
      <version>2.5.0</version>
      <scope>compile</scope>
    </dependency>
  </dependencies>
</plugin>

<dependencies>
  <!-- all dependencies -->
  <!-- you can exclude rest-assured from spring-cloud-contract-verifier -->
  <dependency>
    <groupId>com.jayway.restassured</groupId>
    <artifactId>rest-assured</artifactId>
    <version>2.5.0</version>
    <scope>test</scope>
  </dependency>
  <dependency>
    <groupId>com.jayway.restassured</groupId>
    <artifactId>spring-mock-mvc</artifactId>
    <version>2.5.0</version>
    <scope>test</scope>
  </dependency>
</dependencies>
```


That way, the plugin automatically sees that Rest Assured 2.x is present on the classpath and modifies the imports accordingly.

4.3. Using Snapshot and Milestone Versions for Maven

To use Snapshot and Milestone versions, you have to add the following section to your `pom.xml`:

```
<repositories>
  <repository>
    <id>spring-snapshots</id>
    <name>Spring Snapshots</name>
    <url>https://repo.spring.io/snapshot</url>
    <snapshots>
      <enabled>>true</enabled>
    </snapshots>
  </repository>
  <repository>
    <id>spring-milestones</id>
    <name>Spring Milestones</name>
    <url>https://repo.spring.io/milestone</url>
    <snapshots>
      <enabled>>false</enabled>
    </snapshots>
  </repository>
  <repository>
    <id>spring-releases</id>
    <name>Spring Releases</name>
    <url>https://repo.spring.io/release</url>
    <snapshots>
      <enabled>>false</enabled>
    </snapshots>
  </repository>
</repositories>
<pluginRepositories>
  <pluginRepository>
    <id>spring-snapshots</id>
    <name>Spring Snapshots</name>
    <url>https://repo.spring.io/snapshot</url>
    <snapshots>
      <enabled>>true</enabled>
    </snapshots>
  </pluginRepository>
  <pluginRepository>
    <id>spring-milestones</id>
    <name>Spring Milestones</name>
    <url>https://repo.spring.io/milestone</url>
    <snapshots>
      <enabled>>false</enabled>
    </snapshots>
  </pluginRepository>
  <pluginRepository>
    <id>spring-releases</id>
    <name>Spring Releases</name>
    <url>https://repo.spring.io/release</url>
    <snapshots>
      <enabled>>false</enabled>
    </snapshots>
  </pluginRepository>
</pluginRepositories>
```

```
</snapshots>
  </pluginRepository>
</pluginRepositories>
```

4.4. Adding stubs

By default, Spring Cloud Contract Verifier looks for stubs in the `src/test/resources/contracts` directory. The directory containing stub definitions is treated as a class name, and each stub definition is treated as a single test. We assume that it contains at least one directory to be used as the test class name. If there is more than one level of nested directories, all except the last one is used as the package name. Consider the following structure:

```
src/test/resources/contracts/myservice/shouldCreateUser.groovy
src/test/resources/contracts/myservice/shouldReturnUser.groovy
```

Given that structure, Spring Cloud Contract Verifier creates a test class named `defaultBasePackage.MyService` with two methods:

- `shouldCreateUser()`
- `shouldReturnUser()`

4.5. Run Plugin

The `generateTests` plugin goal is assigned to be invoked in the phase called `generate-test-sources`. If you want it to be part of your build process, you need not do anything. If you want only to generate tests, invoke the `generateTests` goal.

If you want to run stubs from Maven, call the `run` goal with the stubs to run as the `spring.cloud.contract.verifier.stubs` system property as follows:

```
mvn org.springframework.cloud:spring-cloud-contract-maven-plugin:run \
-Dspring.cloud.contract.verifier.stubs="com.acme:service-name"
```

4.6. Configure plugin

To change the default configuration, you can add a `configuration` section to the plugin definition or the `execution` definition, as follows:

```

<plugin>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-contract-maven-plugin</artifactId>
  <executions>
    <execution>
      <goals>
        <goal>convert</goal>
        <goal>generateStubs</goal>
        <goal>generateTests</goal>
      </goals>
    </execution>
  </executions>
  <configuration>

  <basePackageForTests>org.springframework.cloud.verifier.twitter.place</basePackage
  ForTests>

  <baseClassForTests>org.springframework.cloud.verifier.twitter.place.BaseMockMvcSpe
  c</baseClassForTests>
  </configuration>
</plugin>

```

4.7. Configuration Options

- **testMode**: Defines the mode for acceptance tests. By default, the mode is **MockMvc**, which is based on Spring's MockMvc. You can also change it to **WebTestClient**, **JaxRsClient**, or **Explicit** (for real HTTP calls).
- **basePackageForTests**: Specifies the base package for all generated tests. If not set, the value is picked from the package of **baseClassForTests** and from **packageWithBaseClasses**. If neither of these values are set, the value is set to **org.springframework.cloud.contract.verifier.tests**.
- **ruleClassForTests**: Specifies a rule that should be added to the generated test classes.
- **baseClassForTests**: Creates a base class for all generated tests. By default, if you use Spock classes, the class is **spock.lang.Specification**.
- **contractsDirectory**: Specifies a directory that contains contracts written with the Groovyn DSL. The default directory is **/src/test/resources/contracts**.
- **generatedTestSourcesDir**: Specifies the test source directory where tests generated from the Groovy DSL should be placed. By default, its value is **\$buildDir/generated-test-sources/contracts**.
- **generatedTestResourcesDir**: Specifies the test resource directory for resources used by the generated tests.
- **testFramework**: Specifies the target test framework to be used. Currently, Spock, JUnit 4 (**TestFramework.JUNIT**), and JUnit 5 are supported, with JUnit 4 being the default framework.
- **packageWithBaseClasses**: Defines a package where all the base classes reside. This setting takes

precedence over `baseClassForTests`. The convention is such that, if you have a contract under (for example) `src/test/resources/contract/foo/bar/baz/` and set the value of the `packageWithBaseClasses` property to `com.example.base`, Spring Cloud Contract Verifier assumes that there is a `BarBazBase` class under the `com.example.base` package. In other words, the system takes the last two parts of the package, if they exist, and forms a class with `Base` as a suffix.

- `baseClassMappings`: Specifies a list of base class mappings that provide `contractPackageRegex` (which is checked against the package where the contract is located) and `baseClassFQN` (which maps to the fully qualified name of the base class for the matched contract). For example, if you have a contract under `src/test/resources/contract/foo/bar/baz/` and map the `.* → com.example.base.BaseClass` property, the test class generated from these contracts extends `com.example.base.BaseClass`. This setting takes precedence over `packageWithBaseClasses` and `baseClassForTests`.
- `contractsProperties`: A map that contains properties to be passed to Spring Cloud Contract components. Those properties might be used by (for example) built-in or custom Stub Downloaders.
- `failOnNoContracts`: When enabled, will throw an exception when no contracts were found. Defaults to `true`.
- `failOnInProgress`: If set to `true`, then, if any contracts that are in progress are found, they break the build. On the producer side, you need to be explicit about the fact that you have contracts in progress and take into consideration that you might be causing false positive test results on the consumer side. Defaults to `true`.
- `incrementalContractTests`: When enabled, tests are created only when contracts have changed since last build. Defaults to `true`.
- `incrementalContractStubs`: When enabled, stubs are created only when contracts have changed since last build. Defaults to `true`.
- `incrementalContractStubsJar`: When enabled, stubs jar is created only when stubs have changed since last build. Defaults to `true`. `*httpPort`: HTTP port for the WireMock server that serves stubs. Currently `spring.cloud.contract.verifier.http.port` property works only when serving stubs from the directory. Otherwise, when providing stubs id, port have to be included in the id string. `*skip`: Set this to `true` to bypass the verifier execution. `*skipTestOnly`: Set this to `true` to bypass verifier test generation. `*stubs`: List of stubs to be downloaded and ran in a colon separated Ivy notation. `*minPort`: Specifies the minimal port at which the stub should start. `*maxPort`: Specifies the maximal port at which the stub should start. `*waitForKeyPressed`: Specifies if the plugin should wait for the user to press the key after starting the stubs. `*stubsClassifier`: Specifies the classifier used by stubs artifacts.

If you want to download your contract definitions from a Maven repository, you can use the following options:

- `contractDependency`: The contract dependency that contains all the packaged contracts.
- `contractsPath`: The path to the concrete contracts in the JAR with packaged contracts. Defaults to `groupid/artifactid` where `groupid` is slash separated.
- `contractsMode`: Picks the mode in which stubs are found and registered.
- `deleteStubsAfterTest`: If set to `false`, do not remove any downloaded contracts from temporary

directories.

- `contractsRepositoryUrl`: URL to a repository with the artifacts that have contracts. If it is not provided, use the current Maven ones.
- `contractsRepositoryUsername`: The user name to be used to connect to the repo with contracts.
- `contractsRepositoryPassword`: The password to be used to connect to the repo with contracts.
- `contractsRepositoryProxyHost`: The proxy host to be used to connect to the repo with contracts.
- `contractsRepositoryProxyPort`: The proxy port to be used to connect to the repo with contracts.

We cache only non-snapshot, explicitly provided versions (for example `+` or `1.0.0.BUILD-SNAPSHOT` do not get cached). By default, this feature is turned on.

The following list describes experimental features that you can turn on in the plugin:

- `convertToYaml`: Converts all DSLs to the declarative YAML format. This can be extremely useful when you use external libraries in your Groovy DSLs. By turning this feature on (by setting it to `true`), you need not add the library dependency on the consumer side.
- `assertJsonSize`: You can check the size of JSON arrays in the generated tests. This feature is disabled by default.

4.8. Single Base Class for All Tests

When using Spring Cloud Contract Verifier in the default (`MockMvc`), you need to create a base specification for all generated acceptance tests. In this class, you need to point to an endpoint, which should be verified. The following example shows how to do so:

```
package org.mycompany.tests

import org.mycompany.ExampleSpringController
import com.jayway.restassured.module.mockmvc.RestAssuredMockMvc
import spock.lang.Specification

class MvcSpec extends Specification {
    def setup() {
        RestAssuredMockMvc.standaloneSetup(new ExampleSpringController())
    }
}
```

If necessary, you can also setup the whole context, as the following example shows:

```

import io.restassured.module.mockmvc.RestAssuredMockMvc;
import org.junit.Before;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.test.context.junit4.SpringRunner;
import org.springframework.web.context.WebApplicationContext;

@RunWith(SpringRunner.class)
@SpringBootTest(webEnvironment = WebEnvironment.RANDOM_PORT, classes =
SomeConfig.class, properties="some=property")
public abstract class BaseTestClass {

    @Autowired
    WebApplicationContext context;

    @Before
    public void setup() {
        RestAssuredMockMvc.webAppContextSetup(this.context);
    }
}

```

If you use **EXPLICIT** mode, you can use a base class to initialize the whole tested app, similar to what you might do in regular integration tests. The following example shows how to do so:

```

import io.restassured.RestAssured;
import org.junit.Before;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.boot.web.server.LocalServerPort;
import org.springframework.test.context.junit4.SpringRunner;
import org.springframework.web.context.WebApplicationContext;

@RunWith(SpringRunner.class)
@SpringBootTest(webEnvironment = WebEnvironment.RANDOM_PORT, classes =
SomeConfig.class, properties="some=property")
public abstract class BaseTestClass {

    @LocalServerPort
    int port;

    @Before
    public void setup() {
        RestAssured.baseURI = "http://localhost:" + this.port;
    }
}

```

If you use the **JAXRSCLIENT** mode, this base class should also contain a **protected WebTarget webTarget** field. Right now, the only way to test the JAX-RS API is to start a web server.

4.9. Using Different Base Classes for Contracts

If your base classes differ between contracts, you can tell the Spring Cloud Contract plugin which class should get extended by the autogenerated tests. You have two options:

- Follow a convention by providing a value for **packageWithBaseClasses**
- Provide explicit mapping with **baseClassMappings**

4.9.1. By Convention

The convention is such that if you have a contract under (for example) **src/test/resources/contract/foo/bar/baz/** and set the value of the **packageWithBaseClasses** property to **com.example.base**, then Spring Cloud Contract Verifier assumes that there is a **BarBazBase** class under the **com.example.base** package. In other words, the system takes the last two parts of the package, if they exist, and forms a class with a **Base** suffix. This rule takes precedence over **baseClassForTests**. The following example shows how it works in the **contracts** closure:


```

<plugin>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-contract-maven-plugin</artifactId>
  <configuration>
    <packageWithBaseClasses>hello</packageWithBaseClasses>
  </configuration>
</plugin>

```

4.9.2. By Mapping

You can manually map a regular expression of the contract's package to the fully qualified name of the base class for the matched contract. You have to provide a list called `baseClassMappings` that consists of `baseClassMapping` objects that each take a `contractPackageRegex` to `baseClassFQN` mapping. Consider the following example:

```

<plugin>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-contract-maven-plugin</artifactId>
  <configuration>
    <baseClassForTests>com.example.FooBase</baseClassForTests>
    <baseClassMappings>
      <baseClassMapping>
        <contractPackageRegex>.*com.*</contractPackageRegex>
        <baseClassFQN>com.example.TestBase</baseClassFQN>
      </baseClassMapping>
    </baseClassMappings>
  </configuration>
</plugin>

```

Assume that you have contracts under these two locations:

- `src/test/resources/contract/com/`
- `src/test/resources/contract/foo/`

By providing the `baseClassForTests`, we have a fallback in case mapping did not succeed. (You can also provide the `packageWithBaseClasses` as a fallback.) That way, the tests generated from `src/test/resources/contract/com/` contracts extend the `com.example.ComBase`, whereas the rest of the tests extend `com.example.FooBase`.

4.10. Invoking Generated Tests

The Spring Cloud Contract Maven Plugin generates verification code in a directory called `/generated-test-sources/contractVerifier` and attaches this directory to `testCompile` goal.

For Groovy Spock code, you can use the following:

```
<plugin>
  <groupId>org.codehaus.gmavenplus</groupId>
  <artifactId>gmavenplus-plugin</artifactId>
  <version>1.5</version>
  <executions>
    <execution>
      <goals>
        <goal>testCompile</goal>
      </goals>
    </execution>
  </executions>
  <configuration>
    <testSources>
      <testSource>
        <directory>${project.basedir}/src/test/groovy</directory>
        <includes>
          <include>**/*.groovy</include>
        </includes>
      </testSource>
      <testSource>
        <directory>${project.build.directory}/generated-test-
sources/contractVerifier</directory>
        <includes>
          <include>**/*.groovy</include>
        </includes>
      </testSource>
    </testSources>
  </configuration>
</plugin>
```

To ensure that the provider side is compliant with defined contracts, you need to invoke `mvn generateTest test`.

4.11. Pushing Stubs to SCM

If you use the SCM (Source Control Management) repository to keep the contracts and stubs, you might want to automate the step of pushing stubs to the repository. To do that, you can add the `pushStubsToScm` goal. The following example shows how to do so:

```

<plugin>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-contract-maven-plugin</artifactId>
  <version>${spring-cloud-contract.version}</version>
  <extensions>>true</extensions>
  <configuration>
    <!-- Base class mappings etc. -->

    <!-- We want to pick contracts from a Git repository -->
    <contractsRepositoryUrl>git://https://github.com/spring-cloud-
samples/spring-cloud-contract-nodejs-contracts-git.git</contractsRepositoryUrl>

    <!-- We reuse the contract dependency section to set up the path
to the folder that contains the contract definitions. In our case the
path will be /groupId/artifactId/version/contracts -->
    <contractDependency>
      <groupId>${project.groupId}</groupId>
      <artifactId>${project.artifactId}</artifactId>
      <version>${project.version}</version>
    </contractDependency>

    <!-- The contracts mode can't be classpath -->
    <contractsMode>REMOTE</contractsMode>
  </configuration>
  <executions>
    <execution>
      <phase>package</phase>
      <goals>
        <!-- By default we will not push the stubs back to SCM,
you have to explicitly add it as a goal -->
        <goal>pushStubsToScm</goal>
      </goals>
    </execution>
  </executions>
</plugin>

```

Under [Using the SCM Stub Downloader](#), you can find all possible configuration options that you can pass through the `<configuration><contractsProperties>` map, a system property, or an environment variable. For instance, you could specify a concrete branch to checkout, instead of the default one

```

<plugin>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-contract-maven-plugin</artifactId>
  <version>${spring-cloud-contract.version}</version>
  <extensions>>true</extensions>
  <configuration>
    <!-- Base class mappings etc. -->

    <!-- We want to pick contracts from a Git repository -->
    <contractsRepositoryUrl>git://https://github.com/spring-cloud-
samples/spring-cloud-contract-nodejs-contracts-git.git</contractsRepositoryUrl>
    <contractsProperties>
      <git.branch>another_branch</git.branch>
    </contractsProperties>

    <!-- We reuse the contract dependency section to set up the path
to the folder that contains the contract definitions. In our case the
path will be /groupId/artifactId/version/contracts -->
    <contractDependency>
      <groupId>${project.groupId}</groupId>
      <artifactId>${project.artifactId}</artifactId>
      <version>${project.version}</version>
    </contractDependency>

    <!-- The contracts mode can't be classpath -->
    <contractsMode>REMOTE</contractsMode>
  </configuration>
  <executions>
    <execution>
      <phase>package</phase>
      <goals>
        <!-- By default we will not push the stubs back to SCM,
you have to explicitly add it as a goal -->
        <goal>pushStubsToScm</goal>
      </goals>
    </execution>
  </executions>
</plugin>

```

4.12. Maven Plugin and STS

The following image shows an exception that you may see when you use STS:

[STS Exception] | <https://raw.githubusercontent.com/spring-cloud/spring-cloud->

contract/main/docs/src/main/asciidoc/images/sts_exception.png

When you click on the error marker, you should see something like the following:

```
plugin:1.1.0.M1:convert:default-convert:process-test-resources)
org.apache.maven.plugin.PluginExecutionException: Execution default-convert of
goal org.springframework.cloud:spring-
cloud-contract-maven-plugin:1.1.0.M1:convert failed. at
org.apache.maven.plugin.DefaultBuildPluginManager.executeMojo(DefaultBuildPluginMa
nager.java:145) at
org.eclipse.m2e.core.internal.embedder.MavenImpl.execute(MavenImpl.java:331) at
org.eclipse.m2e.core.internal.embedder.MavenImpl$11.call(MavenImpl.java:1362) at
...
org.eclipse.core.internal.jobs.Worker.run(Worker.java:55) Caused by:
java.lang.NullPointerException at
org.eclipse.m2e.core.internal.builder.plexusbuildapi.EclipseIncrementalBuildConte
xt.hasDelta(EclipseIncrementalBuildContext.java:53) at
org.sonatype.plexus.build.incremental.ThreadBuildContext.hasDelta(ThreadBuildCont
ext.java:59) at
```

To fix this issue, provide the following section in your `pom.xml`:

```

<build>
  <pluginManagement>
    <plugins>
      <!--This plugin's configuration is used to store Eclipse m2e settings
        only. It has no influence on the Maven build itself. -->
      <plugin>
        <groupId>org.eclipse.m2e</groupId>
        <artifactId>lifecycle-mapping</artifactId>
        <version>1.0.0</version>
        <configuration>
          <lifecycleMappingMetadata>
            <pluginExecutions>
              <pluginExecution>
                <pluginExecutionFilter>
                  <groupId>org.springframework.cloud</groupId>
                  <artifactId>spring-cloud-contract-maven-
plugin</artifactId>
                  <versionRange>[1.0,)</versionRange>
                  <goals>
                    <goal>convert</goal>
                  </goals>
                </pluginExecutionFilter>
                <action>
                  <execute />
                </action>
              </pluginExecution>
            </pluginExecutions>
          </lifecycleMappingMetadata>
        </configuration>
      </plugin>
    </plugins>
  </pluginManagement>
</build>

```

4.13. Maven Plugin with Spock Tests

You can select the [Spock Framework](#) for creating and running the auto-generated contract verification tests with both Maven and Gradle. However, while using Gradle is straightforward, in Maven, you need some additional setup in order to make the tests compile and execute properly.

First of all, you must use a plugin, such as the [GMavenPlus](#) plugin, to add Groovy to your project. In GMavenPlus plugin, you need to explicitly set test sources, including both the path where your base test classes are defined and the path where the generated contract tests are added. The following example shows how to do so:

```

<plugin>
  <groupId>org.codehaus.gmavenplus</groupId>
  <artifactId>gmavenplus-plugin</artifactId>
  <version>1.13.0</version>
  <executions>
    <execution>
      <goals>
        <goal>addSources</goal>
        <goal>addTestSources</goal>
        <goal>compile</goal>
        <goal>compileTests</goal>
      </goals>
    </execution>
  </executions>
  <configuration>
    <testSources>
      <testSource>
        <directory>${project.basedir}/src/test/groovy</directory>
        <includes>
          <include>**/*.groovy</include>
        </includes>
      </testSource>
      <testSource>
        <directory>
          ${project.basedir}/target/generated-test-
sources/contracts/com/example/beer
        </directory>
        <includes>
          <include>**/*.groovy</include>
          <include>**/*.gvy</include>
        </includes>
      </testSource>
    </testSources>
  </configuration>

```

If you uphold the Spock convention of ending the test class names with **Spec**, you also need to adjust your Maven Surefire plugin setup, as the following example shows:

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-surefire-plugin</artifactId>
  <configuration>
    <detail>true</detail>
    <includes>
      <include>**/*Test.*</include>
      <include>**/*Tests.*</include>
      <include>**/*Spec.*</include>
    </includes>
    <failIfNoTests>true</failIfNoTests>
  </configuration>
</plugin>
```

5. Gradle Project

To learn how to set up the Gradle project for Spring Cloud Contract Verifier, read the following sections:

- [Prerequisites](#)
- [Add Gradle Plugin with Dependencies](#)
- [Gradle and Rest Assured 2.0](#)
- [Snapshot Versions for Gradle](#)
- [Add stubs](#)
- [Default Setup](#)
- [Configuring the Plugin](#)
- [Configuration Options](#)
- [Single Base Class for All Tests](#)
- [Different Base Classes for Contracts](#)
- [Invoking Generated Tests](#)
- [Publishing Stubs to Artifact Repository](#)
- [Pushing Stubs to SCM](#)
- [Spring Cloud Contract Verifier on the Consumer Side](#)

5.1. Prerequisites

To use Spring Cloud Contract Verifier with WireMock, you must use either a Gradle or a Maven plugin.



If you want to use Spock in your projects, you must separately add the `spock-core` and `spock-spring` modules. See [Spock's documentation](#) for more information.

5.2. Add Gradle Plugin with Dependencies

To add a Gradle plugin with dependencies, you can use code similar to the following:

Plugin DSL GA versions

```
// build.gradle
plugins {
    id "groovy"
    // this will work only for GA versions of Spring Cloud Contract
    id "org.springframework.cloud.contract" version "${GAVerifierVersion}"
}

dependencyManagement {
    imports {
        mavenBom "org.springframework.cloud:spring-cloud-contract-
dependencies:${GAVerifierVersion}"
    }
}

dependencies {
    testImplementation "org.apache.groovy:groovy-all:${groovyVersion}"
    // example with adding Spock core and Spock Spring
    testImplementation "org.spockframework:spock-core:${spockVersion}"
    testImplementation "org.spockframework:spock-spring:${spockVersion}"
    testImplementation 'org.springframework.cloud:spring-cloud-starter-contract-
verifier'
}
```

Plugin DSL non GA versions

```
// settings.gradle
pluginManagement {
    plugins {
        id "org.springframework.cloud.contract" version "${verifierVersion}"
    }
    repositories {
        // to pick from local .m2
        mavenLocal()
        // for snapshots
        maven { url "https://repo.spring.io/snapshot" }
        // for milestones
        maven { url "https://repo.spring.io/milestone" }
        // for GA versions
        gradlePluginPortal()
    }
}

// build.gradle
plugins {
    id "groovy"
    id "org.springframework.cloud.contract"
}

dependencyManagement {
    imports {
        mavenBom "org.springframework.cloud:spring-cloud-contract-
dependencies:${verifierVersion}"
    }
}

dependencies {
    testImplementation "org.apache.groovy:groovy-all:${groovyVersion}"
    // example with adding Spock core and Spock Spring
    testImplementation "org.spockframework:spock-core:${spockVersion}"
    testImplementation "org.spockframework:spock-spring:${spockVersion}"
    testImplementation 'org.springframework.cloud:spring-cloud-starter-contract-
verifier'
}
```

Legacy Plugin Application

```
// build.gradle
buildscript {
    repositories {
        mavenCentral()
    }
    dependencies {
        classpath "org.springframework.boot:spring-boot-gradle-
plugin:${springboot_version}"
        classpath "org.springframework.cloud:spring-cloud-contract-gradle-
plugin:${verifier_version}"
        // here you can also pass additional dependencies such as Kotlin spec
e.g.:
        // classpath "org.springframework.cloud:spring-cloud-contract-spec-
kotlin:${verifier_version}"
    }
}

apply plugin: 'groovy'
apply plugin: 'org.springframework.cloud.contract'

dependencyManagement {
    imports {
        mavenBom "org.springframework.cloud:spring-cloud-contract-
dependencies:${verifier_version}"
    }
}

dependencies {
    testImplementation "org.apache.groovy:groovy-all:${groovyVersion}"
    // example with adding Spock core and Spock Spring
    testImplementation "org.spockframework:spock-core:${spockVersion}"
    testImplementation "org.spockframework:spock-spring:${spockVersion}"
    testImplementation 'org.springframework.cloud:spring-cloud-starter-contract-
verifier'
}
```

5.3. Gradle and Rest Assured 2.0

By default, Rest Assured 3.x is added to the classpath. However, to use Rest Assured 2.x, you can add it instead, as the following listing shows:

```

buildscript {
    repositories {
        mavenCentral()
    }
    dependencies {
        classpath "org.springframework.boot:spring-boot-gradle-
plugin:${springboot_version}"
        classpath "org.springframework.cloud:spring-cloud-contract-gradle-
plugin:${verifier_version}"
    }
}

dependencies {
    // all dependencies
    // you can exclude rest-assured from spring-cloud-contract-verifier
    testCompile "com.jayway.restassured:rest-assured:2.5.0"
    testCompile "com.jayway.restassured:spring-mock-mvc:2.5.0"
}

```

That way, the plugin automatically sees that Rest Assured 2.x is present on the classpath and modifies the imports accordingly.

5.4. Snapshot Versions for Gradle

You can add the additional snapshot repository to your `settings.gradle` to use snapshot versions, which are automatically uploaded after every successful build, as the following listing shows:

```

pluginManagement {
    repositories {
        mavenLocal()
        maven { url "https://repo.spring.io/snapshot" }
        maven { url "https://repo.spring.io/milestone" }
        maven { url "https://repo.spring.io/release" }
        gradlePluginPortal()
    }
}

```

5.5. Add stubs

By default, Spring Cloud Contract Verifier looks for stubs in the `src/contractTest/resources/contracts` directory. For transitional purposes the plugin will also look for contracts in `src/test/resources/contracts`, however, this directory is deprecated as of Spring Cloud Contract 3.0.0.

It should also be noted, that with this new Gradle source set, you should also migrate any base classes used within your contract tests to `src/contractTest/{language}` where `{language}` should be replaced with Java or Groovy as needed for your purposes.

The directory that contains stub definitions is treated as a class name, and each stub definition is treated as a single test. Spring Cloud Contract Verifier assumes that it contains at least one level of directories that are to be used as the test class name. If more than one level of nested directories is present, all except the last one is used as the package name. Consider the following structure:

```
src/contractTest/resources/contracts/myservice/shouldCreateUser.groovy
src/contractTest/resources/contracts/myservice/shouldReturnUser.groovy
```

Given the preceding structure, Spring Cloud Contract Verifier creates a test class named `defaultBasePackage.MyService` with two methods:

- `shouldCreateUser()`
- `shouldReturnUser()`

5.6. Running the Plugin

The plugin registers itself to be invoked before a `check` task. If you want it to be part of your build process, you need do nothing more. If you want only to generate tests, invoke the `generateContractTests` task.

5.7. Default Setup

The default Gradle Plugin setup creates the following Gradle part of the build (in pseudocode):

```

contracts {
    testFramework = 'JUNIT'
    testMode = 'MockMvc'
    generatedTestJavaSourcesDir = project.file("${project.buildDir}/generated-
test-sources/contractTest/java")
    generatedTestGroovySourcesDir = project.file("${project.buildDir}/generated-
test-sources/contractTest/groovy")
    generatedTestResourcesDir = project.file("${project.buildDir}/generated-test-
resources/contracts")
    contractsDslDir =
project.file("${project.projectDir}/src/contractTest/resources/contracts")
    basePackageForTests = 'org.springframework.cloud.verifier.tests'
    stubsOutputDir = project.file("${project.buildDir}/stubs")
    sourceSet = null
}

def verifierStubsJar = tasks.register(type: Jar, name: 'verifierStubsJar',
dependsOn: 'generateClientStubs') {
    baseName = project.name
    classifier = contracts.stubsSuffix
    from contractVerifier.stubsOutputDir
}

def copyContracts = tasks.register(type: Copy, name: 'copyContracts') {
    from contracts.contractsDslDir
    into contracts.stubsOutputDir
}

verifierStubsJar.dependsOn copyContracts

```

5.8. Configuring the Plugin

To change the default configuration, you can add a `contracts` snippet to your Gradle configuration, as the following listing shows:

```

contracts {
    testMode = 'MockMvc'
    baseClassForTests = 'org.mycompany.tests'
    generatedTestJavaSourcesDir = project.file('src/generatedContract')
}

```

To download contracts from a remote source, you can use the following snippets as needed:

```

contracts {
    // If your contracts exist in a JAR archive published to a Maven repository
    contractDependency {
        stringNotation = ''
        // OR
        groupId = ''
        artifactId = ''
        version = ''
        classifier = ''
    }

    // If your contracts exist in a Git SCM repository
    contractRepository {
        repositoryUrl = ''
        // username = ''
        // password = ''
    }

    // controls the nested location to find the contracts in either the JAR or Git
    SCM source
    contractsPath = ''
}

```

Since we are using Gradle's Jar packaging task, there are several options and capabilities that you may wish to utilize to further extend what is created by the `verifierStubsJar`. In order to do this, you would use the native mechanisms provided directly by Gradle for customizing an existing task like so:



for the sake of the example, we desire to add a `git.properties` file to the `verifierStubsJar`.

```

verifierStubsJar {
    from("${buildDir}/resources/main/") {
        include("git.properties")
    }
}

```

It should also be noted that as of 3.0.0, the default publication has been disabled. As a result this means, that you are able to create any named jar and publish it as you would normally have done via Gradle configuration options. This means that you can build a jar file customized just the way you would like and publish that for absolute full control over the jar's layout and contents.

5.9. Configuration Options

- **testMode**: Defines the mode for acceptance tests. By default, the mode is `MockMvc`, which is based on Spring's `MockMvc`. It can also be changed to `WebTestClient`, `JaxRsClient`, or `Explicit` (for real HTTP calls).
- **imports**: Creates an array with imports that should be included in the generated tests (for example, `['org.myorg.Matchers']`). By default, it creates an empty array.
- **staticImports**: Creates an array with static imports that should be included in generated tests (for example, `['org.myorg.Matchers.*']`). By default, it creates an empty array.
- **basePackageForTests**: Specifies the base package for all generated tests. If not set, the value is picked from the package of `baseClassForTests` and from `packageWithBaseClasses`. If neither of these values are set, the value is set to `org.springframework.cloud.contract.verifier.tests`.
- **baseClassForTests**: Creates a base class for all generated tests. By default, if you use Spock classes, the class is `spock.lang.Specification`.
- **packageWithBaseClasses**: Defines a package where all the base classes reside. This setting takes precedence over `baseClassForTests`.
- **baseClassMappings**: Explicitly maps a contract package to a FQN of a base class. This setting takes precedence over `packageWithBaseClasses` and `baseClassForTests`.
- **ignoredFiles**: Uses an `Antmatcher` to allow defining stub files for which processing should be skipped. By default, it is an empty array.
- **contractsDslDir**: Specifies the directory that contains contracts written by using the GroovyDSL. By default, its value is `$projectDir/src/contractTest/resources/contracts`.
- **generatedTestSourcesDir**: Specifies the test source directory where tests generated from the Groovy DSL should be placed. (Deprecated)
- **generatedTestJavaSourcesDir**: Specifies the test source directory where Java/JUnit tests generated from the Groovy DSL should be placed. By default, its value is `$buildDir/generated-test-sources/contractTest/java`.
- **generatedTestGroovySourcesDir**: Specifies the test source directory where Groovy/Spock tests generated from the Groovy DSL should be placed. By default, its value is `$buildDir/generated-test-sources/contractTest/groovy`.
- **generatedTestResourcesDir**: Specifies the test resource directory where resources used by the tests generated from the Groovy DSL should be placed. By default, its value is `$buildDir/generated-test-resources/contractTest`.
- **stubsOutputDir**: Specifies the directory where the generated WireMock stubs from the Groovy DSL should be placed.
- **testFramework**: Specifies the target test framework to be used. Currently, Spock, JUnit 4 (`TestFramework.JUNIT`) and JUnit 5 are supported, with JUnit 4 being the default framework.
- **contractsProperties**: A map that contains properties to be passed to Spring Cloud Contract components. Those properties might be used by (for example) built-in or custom Stub Downloaders.
- **sourceSet**: Source set where the contracts are stored. If not provided will assume `contractTest`

(for example, `project.sourceSets.contractTest.java` for JUnit or `project.sourceSets.contractTest.groovy` for Spock).

You can use the following properties when you want to specify the location of the JAR that contains the contracts:

- `contractDependency`: Specifies the Dependency that provides `groupid:artifactid:version:classifier` coordinates. You can use the `contractDependency` closure to set it up.
- `contractsPath`: Specifies the path to the jar. If contract dependencies are downloaded, the path defaults to `groupid/artifactid`, where `groupid` is slash separated. Otherwise, it scans contracts under the provided directory.
- `contractsMode`: Specifies the mode for downloading contracts (whether the JAR is available offline, remotely, and so on).
- `deleteStubsAfterTest`: If set to `false`, does not remove any downloaded contracts from temporary directories.
- `failOnNoContracts`: When enabled, will throw an exception when no contracts were found. Defaults to `true`.
- `failOnInProgress`: If set to `true`, then, if any contracts that are in progress are found, they break the build. On the producer side, you need to be explicit about the fact that you have contracts in progress and take into consideration that you might be causing false positive test results on the consumer side. Defaults to `true`.

There is also the `contractRepository { ... }` closure that contains the following properties

- `repositoryUrl`: The URL to the repository with contract definitions
- `username` : The repository username
- `password` : The repository password
- `proxyPort` : The port of the proxy
- `proxyHost` : The host of the proxy
- `cacheDownloadedContracts` : If set to `true`, caches the folder where non-snapshot contract artifacts got downloaded. Defaults to `true`.

You can also turn on the following experimental features in the plugin:

- `convertToYaml`: Converts all DSLs to the declarative YAML format. This can be extremely useful when you use external libraries in your Groovy DSLs. By turning this feature on (by setting it to `true`), you need not add the library dependency on the consumer side.
- `assertJsonSize`: You can check the size of JSON arrays in the generated tests. This feature is disabled by default.

5.10. Single Base Class for All Tests

When using Spring Cloud Contract Verifier in MockMvc (the default), you need to create a base specification for all generated acceptance tests. In this class, you need to point to an endpoint,

which should be verified. The following example shows how to do so:

```
abstract class BaseMockMvcSpec extends Specification {

    def setup() {
        RestAssuredMockMvc.standaloneSetup(new PairIdController())
    }

    void isProperCorrelationId(Integer correlationId) {
        assert correlationId == 123456
    }

    void isEmpty(String value) {
        assert value == null
    }

}
```

If you use **Explicit** mode, you can use a base class to initialize the whole tested application, as you might see in regular integration tests. If you use the **JAXRSCLIENT** mode, this base class should also contain a **protected WebTarget webTarget** field. Right now, the only option to test the JAX-RS API is to start a web server.

5.11. Different Base Classes for Contracts

If your base classes differ between contracts, you can tell the Spring Cloud Contract plugin which class should get extended by the autogenerated tests. You have two options:

- Follow a convention by providing the **packageWithBaseClasses**
- Provide explicit mapping by using **baseClassMappings**

5.11.1. By Convention

The convention is such that, if you have a contract in (for example) **src/contractTest/resources/contract/foo/bar/baz/** and set the value of the **packageWithBaseClasses** property to **com.example.base**, then Spring Cloud Contract Verifier assumes that there is a **BarBazBase** class under the **com.example.base** package. In other words, the system takes the last two parts of the package, if they exist, and forms a class with a **Base** suffix. This rule takes precedence over **baseClassForTests**.

5.11.2. By Mapping

You can manually map a regular expression of the contract's package to the fully qualified name of the base class for the matched contract. You have to provide a list called **baseClassMappings** that consists of **baseClassMapping** objects that take a **contractPackageRegex** to **baseClassFQN** mapping.

Assume that you have contracts in the following directories:

- `src/contractTest/resources/contract/com/`
- `src/contractTest/resources/contract/foo/`

By providing `baseClassForTests`, we have a fallback in case mapping did not succeed. (You could also provide the `packageWithBaseClasses` as a fallback.) That way, the tests generated from `src/contractTest/resources/contract/com/` contracts extend the `com.example.ComBase`, whereas the rest of the tests extend `com.example.FooBase`.

5.12. Invoking Generated Tests

To ensure that the provider side is compliant with your defined contracts, you need to run the following command:

```
./gradlew contractTest
```

5.13. Publishing Stubs to Artifact Repository

If you use an binary artifact repository to keep the stubs, you will need to configure the publishing section for Gradle to include the `verifierStubsJar`. To do that, you can use the example configuration below:

```
apply plugin: 'maven-publish'

publishing {
    publications {
        maven(MavenPublication) {
            // other configuration

            artifact verifierStubsJar
        }
    }
}
```

Since 3.0.0, the internal stubs publication has been deprecated and disabled by default. It is recommended to include the `verifierStubsJar` with one of your own publications.

5.14. Pushing Stubs to SCM

If you use the SCM repository to keep the contracts and stubs, you might want to automate the step of pushing stubs to the repository. To do that, you can call the `pushStubsToScm` task by running the

following command:

```
$ ./gradlew pushStubsToScm
```

Under [Using the SCM Stub Downloader](#), you can find all possible configuration options that you can pass either through the `contractsProperties` field (for example, `contracts { contractsProperties = [foo:"bar"] }`), through the `contractsProperties` method (for example, `contracts { contractsProperties([foo:"bar"]) }`), or through a system property or an environment variable.

5.15. Spring Cloud Contract Verifier on the Consumer Side

In a consuming service, you need to configure the Spring Cloud Contract Verifier plugin in exactly the same way as in the case of a provider. If you do not want to use Stub Runner, you need to copy the contracts stored in `src/contractTest/resources/contracts` and generate WireMock JSON stubs by using the following command:

```
./gradlew generateClientStubs
```



The `stubsOutputDir` option has to be set for stub generation to work.

When present, you can use JSON stubs in automated tests to consume a service. The following example shows how to do so:

```

@ContextConfiguration(loader == SpringApplicationContextLoader, classes ==
Application)
class LoanApplicationServiceSpec extends Specification {

    @ClassRule
    @Shared
    WireMockClassRule wireMockRule == new WireMockClassRule()

    @Autowired
    LoanApplicationService sut

    def 'should successfully apply for loan'() {
        given:
            LoanApplication application =
                new LoanApplication(client: new Client(clientPesel: '12345678901'),
amount: 123.123)
        when:
            LoanApplicationResult loanApplication == sut.loanApplication(application)
        then:
            loanApplication.loanApplicationStatus == LoanApplicationStatus.LOAN_APPLIED
            loanApplication.rejectionReason == null
    }
}

```

In the preceding example, `LoanApplication` makes a call to the `FraudDetection` service. This request is handled by a WireMock server configured with stubs that were generated by Spring Cloud Contract Verifier.

6. Docker Project

In this section, we publish a `springcloud/spring-cloud-contract` Docker image that contains a project that generates tests and runs them in `EXPLICIT` mode against a running application.



The `EXPLICIT` mode means that the tests generated from contracts send real requests and not mocked ones.

We also publish a `spring-cloud/spring-cloud-contract-stub-runner` Docker image that starts the standalone version of Stub Runner.

6.1. A Short Introduction to Maven, JARs, and Binary Storage

Since non-JVM projects can use the Docker image, it is good to explain the basic terms behind Spring Cloud Contract packaging defaults.

Parts of the following definitions were taken from the [Maven Glossary](#):

- **Project**: Maven thinks in terms of projects. Projects are all you build. Those projects follow a well defined “Project Object Model”. Projects can depend on other projects—in that case, the latter are called “dependencies”. A project may consist of several subprojects. However, these subprojects are still treated equally as projects.
- **Artifact**: An artifact is something that is either produced or used by a project. Examples of artifacts produced by Maven for a project include JAR files and source and binary distributions. Each artifact is uniquely identified by a group ID and an artifact ID that is unique within a group.
- **JAR**: JAR stands for Java ARchive. Its format is based on the ZIP file format. Spring Cloud Contract packages the contracts and generated stubs in a JAR file.
- **GroupId**: A group ID is a universally unique identifier for a project. While this is often just the project name (for example, `commons-collections`), it is helpful to use a fully-qualified package name to distinguish it from other projects with a similar name (for example, `org.apache.maven`). Typically, when published to the Artifact Manager, the `GroupId` gets slash separated and forms part of the URL. For example, for a group ID of `com.example` and an artifact ID of `application`, the result would be `/com/example/application/`.
- **Classifier**: The Maven dependency notation looks as follows: `groupId:artifactId:version:classifier`. The classifier is an additional suffix passed to the dependency—for example, `stubs` or `sources`. The same dependency (for example, `com.example:application`) can produce multiple artifacts that differ from each other with the classifier.
- **Artifact manager**: When you generate binaries, sources, or packages, you would like them to be available for others to download, reference, or reuse. In the case of the JVM world, those artifacts are generally JARs. For Ruby, those artifacts are gems. For Docker, those artifacts are Docker images. You can store those artifacts in a manager. Examples of such managers include [Artifactory](#) and [Nexus](#).

6.2. Generating Tests on the Producer Side

The image searches for contracts under the `/contracts` folder. The output from running the tests is available in the `/spring-cloud-contract/build` folder (useful for debugging purposes).

You can mount your contracts and pass the environment variables. The image then:

- Generates the contract tests
- Runs the tests against the provided URL
- Generates the [WireMock](#) stubs
- Publishes the stubs to a Artifact Manager (optional — turned on by default)

6.2.1. Environment Variables

The Docker image requires some environment variables to point to your running application, to the Artifact manager instance, and so on. The following list describes the environment variables:

The following environment variables are used when tests are run:

6.2.2. Customizing the gradle build

You can provide a customized `gradle.build` to be run in the container by mounting your customized build file as a volume when running the container:

```
$ docker run -v <absolute-path-of-your-custom-file>:/spring-cloud-  
contract/build.gradle springcloud/spring-cloud-contract:<version>
```

6.2.3. Example of Usage via HTTP

In this section, we explore a simple MVC application. To get started, clone the following git repository and cd to the resulting directory, by running the following commands:

```
$ git clone https://github.com/spring-cloud-samples/spring-cloud-contract-nodejs  
$ cd bookstore
```

The contracts are available in the `/contracts` folder.

Since we want to run tests, we can run the following command:

```
$ npm test
```

However, for learning purposes, we split it into pieces, as follows:


```

# Stop docker infra (nodejs, artifactory)
$ ./stop_infra.sh
# Start docker infra (nodejs, artifactory)
$ ./setup_infra.sh

# Kill & Run app
$ pkill -f "node app"
$ nohup node app &

# Prepare environment variables
$ SC_CONTRACT_DOCKER_VERSION="..."
$ APP_IP="192.168.0.100"
$ APP_PORT="3000"
$ ARTIFACTORY_PORT="8081"
$ APPLICATION_BASE_URL="http://${APP_IP}:${APP_PORT}"
$ ARTIFACTORY_URL="http://${APP_IP}:${ARTIFACTORY_PORT}/artifactory/libs-release-
local"
$ CURRENT_DIR="$( pwd )"
$ CURRENT_FOLDER_NAME=${PWD##*/}
$ PROJECT_VERSION="0.0.1.RELEASE"

# Run contract tests
$ docker run --rm -e "APPLICATION_BASE_URL=${APPLICATION_BASE_URL}" -e
"PUBLISH_ARTIFACTS=true" -e "PROJECT_NAME=${CURRENT_FOLDER_NAME}" -e
"REPO_WITH_BINARIES_URL=${ARTIFACTORY_URL}" -e
"PROJECT_VERSION=${PROJECT_VERSION}" -v "${CURRENT_DIR}/contracts:/contracts:ro"
-v "${CURRENT_DIR}/node_modules/spring-cloud-contract/output:/spring-cloud-
contract-output/" springcloud/spring-cloud-
contract:"${SC_CONTRACT_DOCKER_VERSION}"

# Kill app
$ pkill -f "node app"

```

Through bash scripts, the following happens:

- The infrastructure (MongoDb and Artifactory) is set up. In a real-life scenario, you would run the NodeJS application with a mocked database. In this example, we want to show how we can benefit from Spring Cloud Contract in very little time.
- Due to those constraints, the contracts also represent the stateful situation.
 - The first request is a **POST** that causes data to get inserted into the database.
 - The second request is a **GET** that returns a list of data with 1 previously inserted element.
- The NodeJS application is started (on port **3000**).
- The contract tests are generated through Docker, and tests are run against the running application.
 - The contracts are taken from **/contracts** folder.

- The output of the test is available under `node_modules/spring-cloud-contract/output`.
- The stubs are uploaded to Artifactory. You can find them in `localhost:8081/artifactory/libs-release-local/com/example/bookstore/0.0.1.RELEASE/`. The stubs are at `localhost:8081/artifactory/libs-release-local/com/example/bookstore/0.0.1.RELEASE/bookstore-0.0.1.RELEASE-stubs.jar`.

6.2.4. Example of Usage via Messaging

If you want to use Spring Cloud Contract with messaging via the Docker images (e.g. in case of polyglot applications) then you'll have to have the following prerequisites met:

- Middleware (e.g. RabbitMQ or Kafka) must be running before generating tests
- Your contract needs to call a method `triggerMessage(...)` with a `String` parameter that is equal to the contract's `label`.
- Your application needs to have a HTTP endpoint via which we can trigger a message
 - That endpoint should not be available on production (could be enabled via an environment variable)

Example of a Messaging Contract

The contract needs to call a `triggerMessage(...)` method. That method is already provided in the base class for all tests in the docker image and will send out a request to the HTTP endpoint on the producer side. Below you can find examples of such contracts.

Groovy

```
import org.springframework.cloud.contract.spec.Contract

Contract.make {
    description 'Send a pong message in response to a ping message'
    label 'ping_pong'
    input {
        // You have to provide the `triggerMessage` method with the `label`
        // as a String parameter of the method
        triggeredBy('triggerMessage("ping_pong")')
    }
    outputMessage {
        sentTo('output')
        body([
            message: 'pong'
        ])
    }
    metadata(
        [amqp:
            [
                outputMessage: [
                    connectToBroker: [
                        declareQueueWithName: "queue"
                    ],
                    messageProperties: [
                        receivedRoutingKey: '#'
                    ]
                ]
            ]
        ])
    ]
}
```

YAML

```
description: 'Send a pong message in response to a ping message'
label: 'ping_pong'
input:
  # You have to provide the `triggerMessage` method with the `label`
  # as a String parameter of the method
  triggeredBy: 'triggerMessage("ping_pong")'
outputMessage:
  sentTo: 'output'
  body:
    message: 'pong'
metadata:
  amqp:
    outputMessage:
      connectToBroker:
        declareQueueWithName: "queue"
      messageProperties:
        receivedRoutingKey: '#'
```

HTTP Endpoint to Trigger a Message

Why is there need to develop such an endpoint? Spring Cloud Contract would have to generate code in various languages (as it does in Java) to make it possible to trigger production code that sends a message to a broker. If such code is not generated then we need to be able to trigger the message anyways, and the way to do it is to provide an HTTP endpoint that the user will prepare in the language of their choosing.

The endpoint must have the following configuration:

- URL: `/springcloudcontract/{label}` where `label` can be any text
- Method: `POST`
- Basing on the `label` will generate a message that will be sent to a given destination according to the contract definition

Below you have an example of such an endpoint. If you're interested in providing an example in your language don't hesitate to file an issue in the [Spring Cloud Contract repository at Github](#).

Python

```
#!/usr/bin/env python

from flask import Flask
from flask import jsonify
import pika
import os

app = Flask(__name__)

# Production code that sends a message to RabbitMQ
def send_message(cmd):
    connection =
pika.BlockingConnection(pika.ConnectionParameters(host='localhost'))
    channel = connection.channel()
    channel.basic_publish(
        exchange='output',
        routing_key='#',
        body=cmd,
        properties=pika.BasicProperties(
            delivery_mode=2, # make message persistent
        ))
    connection.close()
    return " [x] Sent via Rabbit: %s" % cmd

# This should be ran in tests (shouldn't be publicly available)
if 'CONTRACT_TEST' in os.environ:
    @app.route('/springcloudcontract/<label>', methods=['POST'])
    def springcloudcontract(label):
        if label == "ping_pong":
            return send_message('{"message": "pong"}')
        else:
            raise ValueError('No such label expected.')
```

Running Message Tests on the Producer Side

Now, let's generate tests from contracts to test the producer side. We will run bash code to start the Docker image with attached contracts, however we will also add variables for the messaging code to work. In this case let's assume that the contracts are being stored in a Git repository.

```

#!/bin/bash
set -x

CURRENT_DIR="$( pwd )"

export SC_CONTRACT_DOCKER_VERSION="${SC_CONTRACT_DOCKER_VERSION:-4.0.1-SNAPSHOT}"
export APP_IP="$( ./whats_my_ip.sh )"
export APP_PORT="${APP_PORT:-8000}"
export APPLICATION_BASE_URL="http://${APP_IP}:${APP_PORT}"
export PROJECT_GROUP="${PROJECT_GROUP:-group}"
export PROJECT_NAME="${PROJECT_NAME:-application}"
export PROJECT_VERSION="${PROJECT_VERSION:-0.0.1-SNAPSHOT}"
export PRODUCER_STUBS_CLASSIFIER="${PRODUCER_STUBS_CLASSIFIER:-stubs}"
export FAIL_ON_NO_CONTRACTS="${FAIL_ON_NO_CONTRACTS:-false}"
# In our Python app we want to enable the HTTP endpoint
export CONTRACT_TEST="true"
# In the Verifier docker container we want to add support for RabbitMQ
export MESSAGING_TYPE="rabbit"

# Let's start the infrastructure (e.g. via Docker Compose)
yes | docker-compose kill || echo "Nothing running"
docker-compose up -d

echo "SC Contract Version [${SC_CONTRACT_DOCKER_VERSION}]"
echo "Application URL [${APPLICATION_BASE_URL}]"
echo "Project Version [${PROJECT_VERSION}]"

# Let's run python app
gunicorn -w 4 --bind 0.0.0.0 main:app &
APP_PID=$!

# Generate and run tests
docker run --rm \
    --name verifier \
    # For the image to find the RabbitMQ running in another container
    -e "SPRING_RABBITMQ_ADDRESSES=${APP_IP}:5672" \
    # We need to tell the container what messaging middleware we will
    use
    -e "MESSAGING_TYPE=${MESSAGING_TYPE}" \
    -e "PUBLISH_STUBS_TO_SCM=false" \
    -e "PUBLISH_ARTIFACTS=false" \
    -e "APPLICATION_BASE_URL=${APPLICATION_BASE_URL}" \
    -e "PROJECT_NAME=${PROJECT_NAME}" \
    -e "PROJECT_GROUP=${PROJECT_GROUP}" \
    -e "PROJECT_VERSION=${PROJECT_VERSION}" \
    -e
    "EXTERNAL_CONTRACTS_REPO_WITH_BINARIES_URL=git://https://github.com/marcingrzejszc
    zak/cdct_python_contracts.git" \
    -e "EXTERNAL_CONTRACTS_ARTIFACT_ID=${PROJECT_NAME}" \

```

```
-e "EXTERNAL_CONTRACTS_GROUP_ID=${PROJECT_GROUP}" \
-e "EXTERNAL_CONTRACTS_VERSION=${PROJECT_VERSION}" \
-v "${CURRENT_DIR}/build/spring-cloud-contract/output:/spring-
cloud-contract-output/" \
springcloud/spring-cloud-contract:"${SC_CONTRACT_DOCKER_VERSION}"

kill $APP_PID

yes | docker-compose kill
```

What will happen is:

- Tests will be generated from contracts taken from Git
- In the contract we've provided an entry in metadata called `declareQueueWithName` that will lead to creation of a queue in RabbitMQ with the given name **before** the request to trigger the message is sent
- Via the `triggerMessage("ping_pong")` method call a POST request to the Python application to the `/springcloudcontract/ping_pong` endpoint will be made
- The Python application will generate and send a `'{"message": "pong"}'` JSON via RabbitMQ to an exchange called `output`
- The generated test will poll for a message sent to the `output` exchange
- Once the message was received will assert its contents

After the tests have passed we know that the message was properly sent from the Python app to RabbitMQ.

6.3. Running Stubs on the Consumer Side

This section describes how to use Docker on the consumer side to fetch and run stubs.

We publish a `spring-cloud/spring-cloud-contract-stub-runner` Docker image that starts the standalone version of Stub Runner.

6.3.1. Security

Since the Spring Cloud Contract Stub Runner Docker Image uses the standalone version of Stub Runner the same security considerations need to be taken. You can read more about those [in this section of the documentation](#).

6.3.2. Environment Variables

You can run the docker image and pass any of the [common properties for JUnit and Spring](#) as environment variables. The convention is that all the letters should be upper case. The dot (.) should be replaced with underscore () characters. For example, the `stubrunner.repositoryRoot` property should be represented as a `STUBRUNNER_REPOSITORY_ROOT` environment variable.

In addition to those variables you can set the following ones:

- `MESSAGING_TYPE` - what type of messaging system are you using (currently supported are `rabbit`, `kafka`)
- `ADDITIONAL_OPTS` - any additional properties that you would like to pass to the application

6.3.3. Example of Usage

We want to use the stubs created in this [\[docker-server-side\]](#) step. Assume that we want to run the stubs on port `9876`. You can see the NodeJS code by cloning the repository and changing to the directory indicated in the following commands:

```
$ git clone https://github.com/spring-cloud-samples/spring-cloud-contract-nodejs
$ cd bookstore
```

Now we can run the Stub Runner Boot application with the stubs, by running the following commands:

```
# Provide the Spring Cloud Contract Docker version
$ SC_CONTRACT_DOCKER_VERSION="..."
# The IP at which the app is running and Docker container can reach it
$ APP_IP="192.168.0.100"
# Spring Cloud Contract Stub Runner properties
$ STUBRUNNER_PORT="8083"
# Stub coordinates 'groupId:artifactId:version:classifier:port'
$ STUBRUNNER_IDS="com.example:bookstore:0.0.1.RELEASE:stubs:9876"
$ STUBRUNNER_REPOSITORY_ROOT="http://${APP_IP}:8081/artifactory/libs-release-local"
# Run the docker with Stub Runner Boot
$ docker run --rm \
  -e "STUBRUNNER_IDS=${STUBRUNNER_IDS}" \
  -e "STUBRUNNER_REPOSITORY_ROOT=${STUBRUNNER_REPOSITORY_ROOT}" \
  -e "STUBRUNNER_STUBS_MODE=REMOTE" \
  -p "${STUBRUNNER_PORT}:${STUBRUNNER_PORT}" \
  -p "9876:9876" \
  springcloud/spring-cloud-contract-stub-runner:"${SC_CONTRACT_DOCKER_VERSION}"
```

When the preceding commands run,

- A standalone Stub Runner application gets started.
- It downloads the stub with coordinates `com.example:bookstore:0.0.1.RELEASE:stubs` on port `9876`.
- It gets downloads from Artifactory running at `192.168.0.100:8081/artifactory/libs-release-local`.

- After a while, Stub Runner is running on port **8083**.
- The stubs are running at port **9876**.

On the server side, we built a stateful stub. We can use curl to assert that the stubs are setup properly. To do so, run the following commands:

```
# let's run the first request (no response is returned)
$ curl -H "Content-Type:application/json" -X POST --data '{ "title" : "Title",
"genre" : "Genre", "description" : "Description", "author" : "Author", "publisher"
: "Publisher", "pages" : 100, "image_url" :
"https://d213dhlpdb53mu.cloudfront.net/assets/pivotal-square-logo-
41418bd391196c3022f3cd9f3959b3f6d7764c47873d858583384e759c7db435.svg", "buy_url" :
"https://pivotal.io" }' http://localhost:9876/api/books
# Now time for the second request
$ curl -X GET http://localhost:9876/api/books
# You will receive contents of the JSON
```



If you want use the stubs that you have built locally, on your host, you should set the `-e STUBRUNNER_STUBS_MODE=LOCAL` environment variable and mount the volume of your local m2 (`-v "${HOME}/.m2/:/home/scc/.m2:ro`).

6.3.4. Example of Usage with Messaging

In order to make messaging work it's enough to pass the `MESSAGING_TYPE` environment variable with `kafka` or `rabbit` values. This will lead to setting up the Stub Runner Boot Docker image with dependencies required to connect to the broker.

In order to set the connection properties you can check out Spring Cloud Stream properties page to set proper environment variables.

- [Spring Boot Integration properties](#)
 - You can search for `spring.rabbitmq.xxx` or `spring.kafka.xxx` properties
- [Stream specific RabbitMQ properties](#)
- [Stream specific Kafka properties](#)

The most common property you would set is the location of the running middlewara. If a property to set it is called `spring.rabbitmq.addresses` or `spring.kafka.bootstrap-servers` then you should name the environment variable `SPRING_RABBITMQ_ADDRESSES` and `SPRING_KAFKA_BOOTSTRAP_SERVERS` respectively.

6.4. Running Contract Tests against Existing Middleware

There is legitimate reason to run your contract tests against existing middleware. Some testing

frameworks might give you false positive results - the test within your build passes whereas on production the communication fails.

In Spring Cloud Contract docker images we give an option to connect to existing middleware. As presented in previous subsections we do support Kafka and RabbitMQ out of the box. However, via [Apache Camel Components](#) we can support other middleware too. Let's take a look at the following examples of usage.

6.4.1. Spring Cloud Contract Docker and running Middleware

In order to connect to arbitrary middleware, we'll leverage the `standalone` metadata entry in the contract section.

```
description: 'Send a pong message in response to a ping message'
label: 'standalone_ping_pong' ①
input:
  triggeredBy: 'triggerMessage("ping_pong")' ②
outputMessage:
  sentTo: 'rabbitmq:output' ③
  body: ④
  message: 'pong'
metadata:
  standalone: ⑤
  setup: ⑥
  options: rabbitmq:output?queue=output&routingKey=⑦
  outputMessage: ⑧
  additionalOptions: routingKey=#&queue=output ⑨
```

- ① Label by which we'll be able to trigger the message via Stub Runner
- ② As in the previous messaging examples we'll need to trigger the HTTP endpoint in the running application to make it send a message according to the provided protocol
- ③ `protocol:destination` as requested by Apache Camel
- ④ Output message body
- ⑤ Standalone metadata entry
- ⑥ Setup part will contain information about how to prepare for running contract tests before the actual call to HTTP endpoint of the running application is made
- ⑦ Apache Camel URI to be called in the setup phase. In this case we will try to poll for a message at the `output` exchange and due to to having the `queue=output` and `routingKey=` a queue with name `output` will be set and bound to the `output` exchange with routing key
- ⑧ Additional options (more technical ones) to be appended to the `protocol:destination` from point (3) - together will be combined in the following format `rabbitmq:output?routingKey=#&queue=output`.

For the contract tests to pass we will need as usual in case of messaging in polyglot environment a running application and running middleware. This time we will have different environment variables set for the Spring Cloud Contract Docker image.

```
#!/bin/bash
set -x

# Setup
# Run the middleware
docker-compose up -d rabbitmq ①

# Run the python application
unicorn -w 4 --bind 0.0.0.0 main:app & ②
APP_PID=$!

docker run --rm \
    --name verifier \
    -e "STANDALONE_PROTOCOL=rabbitmq" \ ③
    -e "CAMEL_COMPONENT_RABBITMQ_ADDRESSES=172.18.0.1:5672" \ ④
    -e "PUBLISH_STUBS_TO_SCM=false" \
    -e "PUBLISH_ARTIFACTS=false" \
    -e "APPLICATION_BASE_URL=172.18.0.1" \
    -e "PROJECT_NAME=application" \
    -e "PROJECT_GROUP=group" \
    -e "EXTERNAL_CONTRACTS_ARTIFACT_ID=application" \
    -e "EXTERNAL_CONTRACTS_GROUP_ID=group" \
    -e "EXTERNAL_CONTRACTS_VERSION=0.0.1-SNAPSHOT" \
    -v "${CURRENT_DIR}/build/spring-cloud-contract/output:/spring-cloud-
contract-output/" \
    springcloud/spring-cloud-contract:"${SC_CONTRACT_DOCKER_VERSION}"

# Teardown
kill $APP_PID
yes | docker-compose kill
```

- ① We need to have the middleware running first
- ② The application needs to be up and running
- ③ Via the `STANDALONE_PROTOCOL` environment variable we will fetch a [Apache Camel Component](#). The artifact that we will fetch is `org.apache.camel.springboot:camel-${STANDALONE_PROTOCOL}-starter`. In other words `STANDALONE_PROTOCOL` is matching Camel's component.
- ④ We're setting addresses (we could be setting credentials) via Camel's Spring Boot Starter mechanisms. Example for [Apache Camel's RabbitMQ Spring Boot Auto-Configuration](#)

6.4.2. Stub Runner Docker and running Middleware

In order to trigger a stub message against running middleware, we can run Stub Runner Docker image in the following manner.

Example of usage

```

$ docker run \
  -e "CAMEL_COMPONENT_RABBITMQ_ADDRESSES=172.18.0.1:5672" \ ①
  -e "STUBRUNNER_IDS=group:application:0.0.1-SNAPSHOT" \ ②
  -e
  "STUBRUNNER_REPOSITORY_ROOT=git://https://github.com/marcingrzejszczak/cdet_python_con
  tracts.git" \ ③
  -e ADDITIONAL_OPTS="--
  thin.properties.dependencies.rabbitmq=org.apache.camel.springboot:camel-rabbitmq-
  starter:3.4.0" \ ④
  -e "STUBRUNNER_STUBS_MODE=REMOTE" \ ⑤
  -v "${HOME}/.m2/:/home/scc/.m2:ro" \ ⑥
  -p 8750:8750 \ ⑦
  springcloud/spring-cloud-contract-stub-runner:3.0.4-SNAPSHOT ⑧

```

- ① We're injecting the address of RabbitMQ via [Apache Camel's Spring Boot Auto-Configuration](#)
- ② We're telling Stub Runner which stubs to download
- ③ We're providing an external location for our stubs (Git repository)
- ④ Via the `ADDITIONAL_OPTS="--thin.properties.dependencies.XXX=GROUP:ARTIFACT:VERSION` property we're telling Stub Runner which additional dependency to fetch at runtime. In this case we want to fetch `camel-rabbitmq-starter` so `XXX` is a random string and we want to fetch `org.apache.camel.springboot:camel-rabbitmq-starter` artifact in version `3.4.0`.
- ⑤ Since we're using Git, the remote option of fetching stubs needs to be set
- ⑥ So that we speed up launching of Stub Runner, we're attaching our local Maven repository `.m2` as a volume. If you don't have it populated you can consider setting the write permissions via `:rw` instead read only `:ro`.
- ⑦ We expose the port `8750` at which Stub Runner is running.
- ⑧ Coordinates of the Stub Runner Docker image.

After a while you'll notice the following text in your console, which means that Stub Runner is ready to accept requests.

```

o.a.c.impl.engine.AbstractCamelContext : Apache Camel 3.4.3 (camel-1) started in
0.007 seconds
o.s.c.c.s.server.StubRunnerBoot : Started StubRunnerBoot in 14.483 seconds
(JVM running for 18.666)
o.a.c.c.C.[Tomcat].[localhost].[/] : Initializing Spring DispatcherServlet
'dispatcherServlet'
o.s.web.servlet.DispatcherServlet : Initializing Servlet 'dispatcherServlet'
o.s.web.servlet.DispatcherServlet : Completed initialization in 2 ms

```

To get the list of triggers you can send an HTTP GET request to `localhost:8750/triggers` endpoint. To trigger a stub message, you can send a HTTP POST request to `localhost:8750/triggers/standalone_ping_pong`. In the console you'll see:

```
o.s.c.c.v.m.camel.CamelStubMessages      : Will send a message to URI
[rabbitmq:output?routingKey=#&queue=output]
```

If you check the RabbitMQ management console, you'll see that there's 1 message available in the `output` queue.

7. Spring Cloud Contract customization

In this section, we describe how to customize various parts of Spring Cloud Contract.

7.1. DSL Customization



This section is valid only for the Groovy DSL

You can customize the Spring Cloud Contract Verifier by extending the DSL, as shown in the remainder of this section.

7.1.1. Extending the DSL

You can provide your own functions to the DSL. The key requirement for this feature is to maintain the static compatibility. Later in this chapter, you can see examples of:

- Creating a JAR with reusable classes.
- Referencing of these classes in the DSLs.

You can find the full example [here](#).

7.1.2. Common JAR

The following examples show three classes that can be reused in the DSLs.

`PatternUtils` contains functions used by both the consumer and the producer. The following listing shows the `PatternUtils` class:

```
package com.example;

import java.util.regex.Pattern;

/**
 * If you want to use {@link Pattern} directly in your tests
 * then you can create a class resembling this one. It can
 * contain all the {@link Pattern} you want to use in the DSL.
 *
 * <pre>
 * {@code
 * request {
 *     body(
 *         [ age: $(c(PatternUtils.oldEnough()))]
 *     )
 * }
 * </pre>
 *
 * Notice that we're using both {@code $()} for dynamic values
 * and {@code c()} for the consumer side.
 *
 * @author Marcin Grzejszczak
 */
//tag::impl[]
public class PatternUtils {

    public static String tooYoung() {

        return "[0-1][0-9]";

    }

    public static Pattern oldEnough() {

        return Pattern.compile("[2-9][0-9]");

    }

    /**
     * Makes little sense but it's just an example ;)
     */
    public static Pattern ok() {

        return Pattern.compile("OK");

    }

}
//end::impl[]
```

`ConsumerUtils` contains functions used by the consumer. The following listing shows the `ConsumerUtils` class:

```

package com.example;

import org.springframework.cloud.contract.spec.internal.ClientDslProperty;

/**
 * DSL Properties passed to the DSL from the consumer's perspective.
 * That means that on the input side {@code Request} for HTTP
 * or {@code Input} for messaging you can have a regular expression.
 * On the {@code Response} for HTTP or {@code Output} for messaging
 * you have to have a concrete value.
 *
 * @author Marcin Grzejszczak
 */
//tag::impl[]
public class ConsumerUtils {
    /**
     * Consumer side property. By using the {@link ClientDslProperty}
     * you can omit most of boilerplate code from the perspective
     * of dynamic values. Example
     *
     * <pre>
     * {@code
     * request {
     *     body(
     *         [ age: $(ConsumerUtils.oldEnough())]
     *     )
     * }
     * </pre>
     *
     * That way it's in the implementation that we decide what value we will pass
     to the consumer
     * and which one to the producer.
     *
     * @author Marcin Grzejszczak
     */
    public static ClientDslProperty oldEnough() {

        // this example is not the best one and
        // theoretically you could just pass the regex instead of
`ServerDslProperty` but
        // it's just to show some new tricks :)
        return new ClientDslProperty(PatternUtils.oldEnough(), 40);

    }

}
//end::impl[]

```


`ProducerUtils` contains functions used by the producer. The following listing shows the `ProducerUtils` class:

```
package com.example;

import org.springframework.cloud.contract.spec.internal.ServerDslProperty;

/**
 * DSL Properties passed to the DSL from the producer's perspective.
 * That means that on the input side {@code Request} for HTTP
 * or {@code Input} for messaging you have to have a concrete value.
 * On the {@code Response} for HTTP or {@code Output} for messaging
 * you can have a regular expression.
 *
 * @author Marcin Grzejszczak
 */
//tag::impl[]
public class ProducerUtils {

    /**
     * Producer side property. By using the {@link ProducerUtils}
     * you can omit most of boilerplate code from the perspective
     * of dynamic values. Example
     *
     * <pre>
     * {@code
     * response {
     *     body(
     *         [ status: $(ProducerUtils.ok())]
     *     )
     * }
     * </pre>
     *
     * That way it's in the implementation that we decide what value we will pass
     to the consumer
     * and which one to the producer.
     */
    public static ServerDslProperty ok() {
        // this example is not the best one and
        // theoretically you could just pass the regex instead of
        `ServerDslProperty` but
        // it's just to show some new tricks :)
        return new ServerDslProperty( PatternUtils.ok(), "OK");
    }
}
//end::impl[]
```

7.1.3. Adding a Test Dependency in the Project's Dependencies

To add a test dependency in the project's dependencies, you must first add the common jar dependency as a test dependency. Because your contracts files are available on the test resources path, the common jar classes automatically become visible in your Groovy files. The following examples show how to test the dependency:

Maven

```
<dependency>
  <groupId>com.example</groupId>
  <artifactId>beer-common</artifactId>
  <version>${project.version}</version>
  <scope>test</scope>
</dependency>
```

Gradle

```
testImplementation("com.example:beer-common:0.0.1-SNAPSHOT")
```

7.1.4. Adding a Test Dependency in the Plugin's Dependencies

Now, you must add the dependency for the plugin to reuse at runtime, as the following example shows:

Maven

```
<plugin>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-contract-maven-plugin</artifactId>
  <version>${spring-cloud-contract.version}</version>
  <extensions>>true</extensions>
  <configuration>
    <testFramework>JUNIT5</testFramework>
    <packageWithBaseClasses>com.example</packageWithBaseClasses>
    <baseClassMappings>
      <baseClassMapping>
        <contractPackageRegex>.*intoxication.*</contractPackageRegex>

<baseClassFQN>com.example.intoxication.BeerIntoxicationBase</baseClassFQN>
      </baseClassMapping>
    </baseClassMappings>
  </configuration>
  <dependencies>
    <dependency>
      <groupId>com.example</groupId>
      <artifactId>beer-common</artifactId>
      <version>${project.version}</version>
      <scope>compile</scope>
    </dependency>
  </dependencies>
</plugin>
```

Gradle

```
classpath "com.example:beer-common:0.0.1-SNAPSHOT"
```

7.1.5. Referencing Classes in DSLs

You can now reference your classes in your DSL, as the following example shows:

```

package contracts.beer.rest

import com.example.ConsumerUtils
import com.example.ProducerUtils
import org.springframework.cloud.contract.spec.Contract

Contract.make {
    description("""
Represents a successful scenario of getting a beer
    """)
    given:
        client is old enough
    when:
        he applies for a beer
    then:
        we'll grant him the beer
    """)
    request {
        method 'POST'
        url '/check'
        body(
            age: $(ConsumerUtils.oldEnough())
        )
        headers {
            contentType(applicationJson())
        }
    }
    response {
        status 200
        body("""
            {
                "status": "${value(ProducerUtils.ok())}"
            }
        """)
        headers {
            contentType(applicationJson())
        }
    }
}

```



You can set the Spring Cloud Contract plugin up by setting `convertToYaml` to `true`. That way, you do NOT have to add the dependency with the extended functionality to the consumer side, since the consumer side uses YAML contracts instead of Groovy contracts.

7.2. WireMock Customization

In this section, we show how to customize the way you work with [WireMock](#).

7.2.1. Registering Your Own WireMock Extension

WireMock lets you register custom extensions. By default, Spring Cloud Contract registers the transformer, which lets you reference a request from a response. If you want to provide your own extensions, you can register an implementation of the `org.springframework.cloud.contract.verifier.dsl.wiremock.WireMockExtensions` interface. Since we use the `spring.factories` extension approach, you can create an entry similar to the following in the `META-INF/spring.factories` file:

```
org.springframework.cloud.contract.verifier.dsl.wiremock.WireMockExtensions=\
org.springframework.cloud.contract.stubrunner.provider.wiremock.TestWireMockExtens
ions
org.springframework.cloud.contract.spec.ContractConverter=\
org.springframework.cloud.contract.stubrunner.TestCustomYamlContractConverter
```

The following example shows a custom extension:

Example 3. TestWireMockExtensions.groovy

```
/*
 * Copyright 2013-2020 the original author or authors.
 *
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 *     https://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */

package org.springframework.cloud.contract.verifier.dsl.wiremock

import com.github.tomakehurst.wiremock.extension.Extension

/**
 * Extension that registers the default transformer and the custom one
 */
class TestWireMockExtensions implements WireMockExtensions {
    @Override
    List<Extension> extensions() {
        return [
            new DefaultResponseTransformer(),
            new CustomExtension()
        ]
    }
}

class CustomExtension implements Extension {

    @Override
    String getName() {
        return "foo-transformer"
    }
}
```



If you want the transformation to be applied only for a mapping that explicitly requires it, override the `applyGlobally()` method and set it to `false`.

7.2.2. Customization of WireMock Configuration

You can register a bean of type `org.springframework.cloud.contract.wiremock.WireMockConfigurationCustomizer` to customize the WireMock configuration (for example, to add custom transformers). The following example shows how to do so:

```
@Bean
WireMockConfigurationCustomizer optionsCustomizer() {
    return new WireMockConfigurationCustomizer() {
        @Override
        public void customize(WireMockConfiguration options) {
            // perform your customization here
        }
    };
}
```

7.2.3. Customization of WireMock via Metadata

With version 3.0.0 you're able to set `metadata` in your contracts. If you set an entry with key equal to `wiremock` and the value will be a valid WireMock's `StubMapping` JSON / map or an actual `StubMapping` object, Spring Cloud Contract will patch the generated stub with part of your customization. Let's look at the following example

```
name: "should count all frauds"
request:
  method: GET
  url: /yamlfrauds
response:
  status: 200
  body:
    count: 200
  headers:
    Content-Type: application/json
metadata:
  wiremock:
    stubMapping: >
      {
        "response" : {
          "fixedDelayMilliseconds": 2000
        }
      }
    }
```

In the `metadata` section we've set an entry with key `wiremock` and its value is a JSON `StubMapping` that sets a delay in the generated stub. Such code allowed us to get the following merged WireMock JSON stub.

```

{
  "id" : "ebae49e2-a2a3-490c-a57f-ba28e26b81ea",
  "request" : {
    "url" : "/yamlfrauds",
    "method" : "GET"
  },
  "response" : {
    "status" : 200,
    "body" : "{\"count\":200}",
    "headers" : {
      "Content-Type" : "application/json"
    },
    "fixedDelayMilliseconds" : 2000,
    "transformers" : [ "response-template" ]
  },
  "uuid" : "ebae49e2-a2a3-490c-a57f-ba28e26b81ea"
}

```

The current implementation allows to manipulate only the stub side (we don't change the generated test). Also, what does not get changed are the whole request and body and headers of the response.

Customization of WireMock via Metadata and a Custom Processor

If you want to apply a custom WireMock `StubMapping` post processing, you can under `META-INF/spring.factories` under the `org.springframework.cloud.contract.verifier.converter.StubProcessor` key register your own implementation of a stub processor. For your convenience we've created an interface called `org.springframework.cloud.contract.verifier.wiremock.WireMockStubPostProcessor` that is dedicated to WireMock.

You'll have to implement methods to inform Spring Cloud Contract whether the post processor is applicable for a given contract and how should the post processing look like.



On the consumer side, when using Stub Runner, remember to pass the custom `HttpServerStubConfigurer` implementation (e.g. the one that extends `WireMockHttpServerStubConfigurer`) where you'll register a custom extension of your choosing. If you don't do so, even you have a custom WireMock extension on the classpath, WireMock will not notice it, won't apply it and will print out a warning statement that the given extension was not found.

7.3. Using the Pluggable Architecture

You may encounter cases where your contracts have been defined in other formats, such as YAML, RAML, or PACT. In those cases, you still want to benefit from the automatic generation of tests and stubs. You can add your own implementation for generating both tests and stubs. Also, you can customize the way tests are generated (for example, you can generate tests for other languages) and the way stubs are generated (for example, you can generate stubs for other HTTP server

implementations).

7.3.1. Custom Contract Converter

The `ContractConverter` interface lets you register your own implementation of a contract structure converter. The following code listing shows the `ContractConverter` interface:

```
package org.springframework.cloud.contract.spec;

import java.io.File;
import java.util.Collection;

/**
 * Converter to be used to convert FROM {@link File} TO {@link Contract} and from
 * {@link Contract} to {@code T}.
 *
 * @param <T> - type to which we want to convert the contract
 * @author Marcin Grzejszczak
 * @since 1.1.0
 */
public interface ContractConverter<T> extends ContractStorer<T>, ContractReader<T>
{
    /**
     * Should this file be accepted by the converter. Can use the file extension
     to check
     * if the conversion is possible.
     * @param file - file to be considered for conversion
     * @return - {@code true} if the given implementation can convert the file
     */
    boolean isAccepted(File file);

    /**
     * Converts the given {@link File} to its {@link Contract} representation.
     * @param file - file to convert
     * @return - {@link Contract} representation of the file
     */
    Collection<Contract> convertFrom(File file);

    /**
     * Converts the given {@link Contract} to a {@link T} representation.
     * @param contract - the parsed contract
     * @return - {@link T} the type to which we do the conversion
     */
    T convertTo(Collection<Contract> contract);
}
```

Your implementation must define the condition on which it should start the conversion. Also, you must define how to perform that conversion in both directions.



Once you create your implementation, you must create a `/META-INF/spring.factories` file in which you provide the fully qualified name of your implementation.

The following example shows a typical `spring.factories` file:

```
org.springframework.cloud.contract.spec.ContractConverter=\
org.springframework.cloud.contract.verifier.converter.YamlContractConverter
```

7.3.2. Using the Custom Test Generator

If you want to generate tests for languages other than Java or you are not happy with the way the verifier builds Java tests, you can register your own implementation.

The `SingleTestGenerator` interface lets you register your own implementation. The following code listing shows the `SingleTestGenerator` interface:

```

package org.springframework.cloud.contract.verifier.builder;

import java.nio.file.Path;
import java.util.Collection;

import
org.springframework.cloud.contract.verifier.config.ContractVerifierConfigProperties;
import org.springframework.cloud.contract.verifier.file.ContractMetadata;

/**
 * Builds a single test.
 *
 * @since 1.1.0
 */
public interface SingleTestGenerator {

    /**
     * Creates contents of a single test class in which all test scenarios from
     the
     * contract metadata should be placed.
     * @param properties - properties passed to the plugin
     * @param listOfFiles - list of parsed contracts with additional metadata
     * @param generatedClassData - information about the generated class
     * @param includedDirectoryRelativePath - relative path to the included
     directory
     * @return contents of a single test class
     */
    String buildClass(ContractVerifierConfigProperties properties,
Collection<ContractMetadata> listOfFiles,
        String includedDirectoryRelativePath, GeneratedClassData
generatedClassData);

    class GeneratedClassData {

        public final String className;

        public final String classPackage;

        public final Path testClassPath;

        public GeneratedClassData(String className, String classPackage, Path
testClassPath) {
            this.className = className;
            this.classPackage = classPackage;
            this.testClassPath = testClassPath;
        }
    }
}

```

```
}
```

Again, you must provide a `spring.factories` file, such as the one shown in the following example:

```
org.springframework.cloud.contract.verifier.builder.SingleTestGenerator=/  
com.example.MyGenerator
```

7.3.3. Using the Custom Stub Generator

If you want to generate stubs for stub servers other than WireMock, you can plug in your own implementation of the `StubGenerator` interface. The following code listing shows the `StubGenerator` interface:

```

package org.springframework.cloud.contract.verifier.converter;

import java.io.File;
import java.util.List;
import java.util.Map;
import java.util.stream.Collectors;

import org.springframework.cloud.contract.spec.Contract;
import org.springframework.cloud.contract.verifier.file.ContractMetadata;

/**
 * Converts contracts into their stub representation.
 *
 * @param <T> - type of stub mapping
 * @since 1.1.0
 */
public interface StubGenerator<T> {

    /**
     * @param mapping - potential stub mapping mapping
     * @return {@code true} if this converter could have generated this mapping
     stub.
     */
    default boolean canReadStubMapping(File mapping) {
        return mapping.getName().endsWith(fileExtension());
    }

    /**
     * @param rootName - root name of the contract
     * @param content - metadata of the contract
     * @return the collection of converted contracts into stubs. One contract can
     result
     * in multiple stubs.
     */
    Map<Contract, String> convertContents(String rootName, ContractMetadata
content);

    /**
     * Post process a generated stub mapping.
     * @param stubMapping - mapping of a stub
     * @param contract - contract for which stub was generated
     * @return the converted stub mapping
     */
    default T postProcessStubMapping(T stubMapping, Contract contract) {
        List<StubPostProcessor> processors =
StubPostProcessor.PROCESSORS.stream().filter(p -> p.isApplicable(contract))
                .collect(Collectors.toList());
        if (processors.isEmpty()) {
            return defaultStubMappingPostProcessing(stubMapping, contract);
        }
    }
}

```

```

    }
    T stub = stubMapping;
    for (StubPostProcessor processor : processors) {
        stub = (T) processor.postProcess(stub, contract);
    }
    return stub;
}

/**
 * Stub mapping to chose when no post processors where found on the classpath.
 * @param stubMapping - mapping of a stub
 * @param contract - contract for which stub was generated
 * @return the converted stub mapping
 */
default T defaultStubMappingPostProcessing(T stubMapping, Contract contract) {
    return stubMapping;
}

/**
 * @param inputFileName - name of the input file
 * @return the name of the converted stub file. If you have multiple contracts
in a
 * single file then a prefix will be added to the generated file. If you
provide the
 * {@link Contract#getName} field then that field will override the generated
file
 * name.
 *
 * Example: name of file with 2 contracts is {@code foo.groovy}, it will be
converted
 * by the implementation to {@code foo.json}. The recursive file converter
will create
 * two files {@code 0_foo.json} and {@code 1_foo.json}
 */
String generateOutputFileNameForInput(String inputFileName);

/**
 * Describes the file extension of the generated mapping that this stub
generator can
 * handle.
 * @return string describing the file extension
 */
default String fileExtension() {
    return ".json";
}
}

```

Again, you must provide a `spring.factories` file, such as the one shown in the following example:

```
# Stub converters
org.springframework.cloud.contract.verifier.converter.StubGenerator=\
org.springframework.cloud.contract.verifier.wiremock.DslToWireMockClientConverter
```

The default implementation is the WireMock stub generation.



You can provide multiple stub generator implementations. For example, from a single DSL, you can produce both WireMock stubs and Pact files.

7.3.4. Using the Custom Stub Runner

If you decide to use custom stub generation, you also need a custom way of running stubs with your different stub provider.

Assume that you use [Moco](#) to build your stubs and that you have written a stub generator and placed your stubs in a JAR file.

In order for Stub Runner to know how to run your stubs, you have to define a custom HTTP Stub server implementation, which might resemble the following example:

```

package org.springframework.cloud.contract.stubrunner.provider.moco

import com.github.dreamhead.moco.bootstrap.arg.HttpArgs
import com.github.dreamhead.moco.runner.JsonRunner
import com.github.dreamhead.moco.runner.RunnerSetting
import groovy.transform.CompileStatic
import groovy.util.logging.Commons

import org.springframework.cloud.contract.stubrunner.HttpServerStub
import org.springframework.cloud.contract.stubrunner.HttpServerStubConfiguration

@Commons
@CompileStatic
class MocoHttpServerStub implements HttpServerStub {

    private boolean started
    private JsonRunner runner
    private int port

    @Override
    int port() {
        if (!isRunning()) {
            return -1
        }
        return port
    }

    @Override
    boolean isRunning() {
        return started
    }

    @Override
    HttpServerStub start(HttpServerStubConfiguration configuration) {
        this.port = configuration.port
        return this
    }

    @Override
    HttpServerStub stop() {
        if (!isRunning()) {
            return this
        }
        this.runner.stop()
        return this
    }

    @Override
    HttpServerStub registerMappings(Collection<File> stubFiles) {

```



```

        List<RunnerSetting> settings = stubFiles.findAll {
it.name.endsWith("json") }
        .collect {
            log.info("Trying to parse [${it.name}]")
            try {
                return
                RunnerSetting.aRunnerSetting().addStream(it.newInputStream()).
                    build()
            }
            catch (Exception e) {
                log.warn("Exception occurred while trying to parse file
[${it.name}]", e)
                return null
            }
        }.findAll { it }
        this.runner = JsonRunner.newJsonRunnerWithSetting(settings,
            HttpArgs.httpArgs().withPort(this.port).build())
        this.runner.run()
        this.started = true
        return this
    }

    @Override
    String registeredMappings() {
        return ""
    }

    @Override
    boolean isAccepted(File file) {
        return file.name.endsWith(".json")
    }
}

```

Then you can register it in your `spring.factories` file, as the following example shows:

```

org.springframework.cloud.contract.stubrunner.HttpServerStub=\
org.springframework.cloud.contract.stubrunner.provider.moco.MocoHttpServerStub

```

Now you can run stubs with Moco.



If you do not provide any implementation, the default (WireMock) implementation is used. If you provide more than one, the first one on the list is used.

7.3.5. Using the Custom Stub Downloader

You can customize the way your stubs are downloaded by creating an implementation of the

`StubDownloaderBuilder` interface, as the following example shows:

```
package com.example;

class CustomStubDownloaderBuilder implements StubDownloaderBuilder {

    @Override
    public StubDownloader build(final StubRunnerOptions stubRunnerOptions) {
        return new StubDownloader() {
            @Override
            public Map.Entry<StubConfiguration, File> downloadAndUnpackStubJar(
                StubConfiguration config) {
                File unpackedStubs = retrieveStubs();
                return new AbstractMap.SimpleEntry<>(
                    new StubConfiguration(config.getGroupId(),
config.getArtifactId(), version,
                                config.getClassifier()), unpackedStubs);
            }

            File retrieveStubs() {
                // here goes your custom logic to provide a folder where all the
stubs reside
            }
        }
    }
}
```

Then you can register it in your `spring.factories` file, as the following example shows:

```
# Example of a custom Stub Downloader Provider
org.springframework.cloud.contract.stubrunner.StubDownloaderBuilder=\
com.example.CustomStubDownloaderBuilder
```

Now you can pick a folder with the source of your stubs.



If you do not provide any implementation, the default (scanning the classpath) is used. If you provide the `stubsMode = StubRunnerProperties.StubsMode.LOCAL` or `stubsMode = StubRunnerProperties.StubsMode.REMOTE`, the Aether implementation is used. If you provide more than one, the first one on the list is used.

7.3.6. Using the SCM Stub Downloader

Whenever the `repositoryRoot` starts with a SCM protocol (currently, we support only `git://`), the stub downloader tries to clone the repository and use it as a source of contracts to generate tests or

stubs.

Through environment variables, system properties, or properties set inside the plugin or the contracts repository configuration, you can tweak the downloader's behavior. The following table describes the available properties:

Table 1. SCM Stub Downloader properties

Type of a property	Name of the property	Description
* <code>git.branch</code> (plugin prop) * <code>stubrunner.properties.git.branch</code> (system prop) * <code>STUBRUNNER_PROPERTIES_GIT_BRANCH</code> (env prop)	master	Which branch to checkout
* <code>git.username</code> (plugin prop) * <code>stubrunner.properties.git.username</code> (system prop) * <code>STUBRUNNER_PROPERTIES_GIT_USERNAME</code> (env prop)		Git clone username
* <code>git.password</code> (plugin prop) * <code>stubrunner.properties.git.password</code> (system prop) * <code>STUBRUNNER_PROPERTIES_GIT_PASSWORD</code> (env prop)		Git clone password
* <code>git.no-of-attempts</code> (plugin prop) * <code>stubrunner.properties.git.no-of-attempts</code> (system prop) * <code>STUBRUNNER_PROPERTIES_GIT_NO_OF_ATTEMPTS</code> (env prop)	10	Number of attempts to push the commits to <code>origin</code>

* <code>git.wait-between-attempts</code> (Plugin prop)	1000	Number of milliseconds to wait between attempts to push the commits to <code>origin</code>
* <code>stubrunner.properties.git.wait-between-attempts</code> (system prop)		
* <code>STUBRUNNER_PROPERTIES_GIT_WAIT_BETWEEN_ATTEMPTS</code> (env prop)		

8. “How-to” Guides

This section provides answers to some common “how do I do that...?” questions that often arise when using Spring Cloud Contract. Its coverage is not exhaustive, but it does cover quite a lot.

If you have a specific problem that we do not cover here, you might want to check out stackoverflow.com to see if someone has already provided an answer. Stack Overflow is also a great place to ask new questions (please use the `spring-cloud-contract` tag).

We are also more than happy to extend this section. If you want to add a “how-to”, send us a [pull request](#).

8.1. Why use Spring Cloud Contract?

Spring Cloud Contract works great in a polyglot environment. This project has a lot of really interesting features. Quite a few of these features definitely make Spring Cloud Contract Verifier stand out on the market of Consumer Driven Contract (CDC) tooling. The most interesting features include the following:

- Ability to do CDC with messaging.
- Clear and easy to use, statically typed DSL.
- Ability to copy-paste your current JSON file to the contract and edit only its elements.
- Automatic generation of tests from the defined contract.
- Stub Runner functionality: The stubs are automatically downloaded at runtime from Nexus or Artifactory.
- Spring Cloud integration: No discovery service is needed for integration tests.
- Ability to add support for any language & framework through Docker.

8.2. How Can I Write Contracts in a Language Other than Groovy?

You can write a contract in YAML. See [this section](#) for more information.

We are working on allowing more ways of describing the contracts. You can check the [github-issues](#) for more information.

8.3. How Can I Provide Dynamic Values to a Contract?

One of the biggest challenges related to stubs is their reusability. Only if they can be widely used can they serve their purpose. The hard-coded values (such as dates and IDs) of request and response elements generally make that difficult. Consider the following JSON request:

```
{
  "time" : "2016-10-10 20:10:15",
  "id" : "9febab1c-6f36-4a0b-88d6-3b6a6d81cd4a",
  "body" : "foo"
}
```

Now consider the following JSON response:

```
{
  "time" : "2016-10-10 21:10:15",
  "id" : "c4231e1f-3ca9-48d3-b7e7-567d55f0d051",
  "body" : "bar"
}
```

Imagine the pain required to set the proper value of the `time` field (assume that this content is generated by the database) by changing the clock in the system or by providing stub implementations of data providers. The same is related to the `id` field. You could create a stubbed implementation of UUID generator, but doing so makes little sense.

So, as a consumer, you want to send a request that matches any form of a time or any UUID. That way, your system works as usual, generating data without you having to stub out anything. Assume that, in case of the aforementioned JSON, the most important part is the `body` field. You can focus on that and provide matching for other fields. In other words, you would like the stub to work as follows:

```
{
  "time" : "SOMETHING THAT MATCHES TIME",
  "id" : "SOMETHING THAT MATCHES UUID",
  "body" : "foo"
}
```

As far as the response goes, as a consumer, you need a concrete value on which you can operate.

Consequently, the following JSON is valid:

```
{
  "time" : "2016-10-10 21:10:15",
  "id" : "c4231e1f-3ca9-48d3-b7e7-567d55f0d051",
  "body" : "bar"
}
```

In the previous sections, we generated tests from contracts. So, from the producer's side, the situation looks much different. We parse the provided contract, and, in the test, we want to send a real request to your endpoints. So, for the case of a producer for the request, we cannot have any sort of matching. We need concrete values on which the producer's backend can work. Consequently, the following JSON would be valid:

```
{
  "time" : "2016-10-10 20:10:15",
  "id" : "9febab1c-6f36-4a0b-88d6-3b6a6d81cd4a",
  "body" : "foo"
}
```

On the other hand, from the point of view of the validity of the contract, the response does not necessarily have to contain concrete values for `time` or `id`. Suppose you generate those on the producer side. Again, you have to do a lot of stubbing to ensure that you always return the same values. That is why, from the producer's side, you might want the following response:

```
{
  "time" : "SOMETHING THAT MATCHES TIME",
  "id" : "SOMETHING THAT MATCHES UUID",
  "body" : "bar"
}
```

How can you then provide a matcher for the consumer and a concrete value for the producer (and the opposite at some other time)? Spring Cloud Contract lets you provide a dynamic value. That means that it can differ for both sides of the communication.

You can read more about this in the [Contract DSL](#) section.



Read the [Groovy docs related to JSON](#) to understand how to properly structure the request and response bodies.

8.4. How to Do Stubs versioning?

This section covers versioning of the stubs, which you can handle in a number of different ways:

- [API Versioning](#)
- [JAR versioning](#)
- [Development or Production Stubs](#)

8.4.1. API Versioning

What does versioning really mean? If you refer to the API version, there are different approaches:

- Use hypermedia links and do not version your API by any means
- Pass the version through headers and URLs

We do not try to answer the question of which approach is better. You should pick whatever suits your needs and lets you generate business value.

Assume that you do version your API. In that case, you should provide as many contracts with as many versions as you support. You can create a subfolder for every version or append it to the contract name — whatever suits you best.

8.4.2. JAR versioning

If, by versioning, you mean the version of the JAR that contains the stubs, then there are essentially two main approaches.

Assume that you do continuous delivery and deployment, which means that you generate a new version of the jar each time you go through the pipeline and that the jar can go to production at any time. For example, your jar version looks like the following (because it got built on the 20.10.2016 at 20:15:21) :

```
1.0.0.20161020-201521-RELEASE
```

In that case, your generated stub jar should look like the following:

```
1.0.0.20161020-201521-RELEASE-stubs.jar
```

In this case, you should, inside your `application.yml` or `@AutoConfigureStubRunner` when referencing stubs, provide the latest version of the stubs. You can do that by passing the `+` sign. the following example shows how to do so:

```
@AutoConfigureStubRunner(ids = {"com.example:http-server-dsl:+:stubs:8080"})
```

If the versioning, however, is fixed (for example, `1.0.4.RELEASE` or `2.1.1`), you have to set the concrete value of the jar version. The following example shows how to do so for version 2.1.1:

```
@AutoConfigureStubRunner(ids = {"com.example:http-server-dsl:2.1.1:stubs:8080"})
```

8.4.3. Development or Production Stubs

You can manipulate the classifier to run the tests against the current development version of the stubs of other services or the ones that were deployed to production. If you alter your build to deploy the stubs with the `prod-stubs` classifier once you reach production deployment, you can run tests in one case with development stubs and in another case with production stubs.

The following example works for tests that use the development version of the stubs:

```
@AutoConfigureStubRunner(ids = {"com.example:http-server-dsl:+:stubs:8080"})
```

The following example works for tests that use the production version of stubs:

```
@AutoConfigureStubRunner(ids = {"com.example:http-server-dsl:+:prod-stubs:8080"})
```

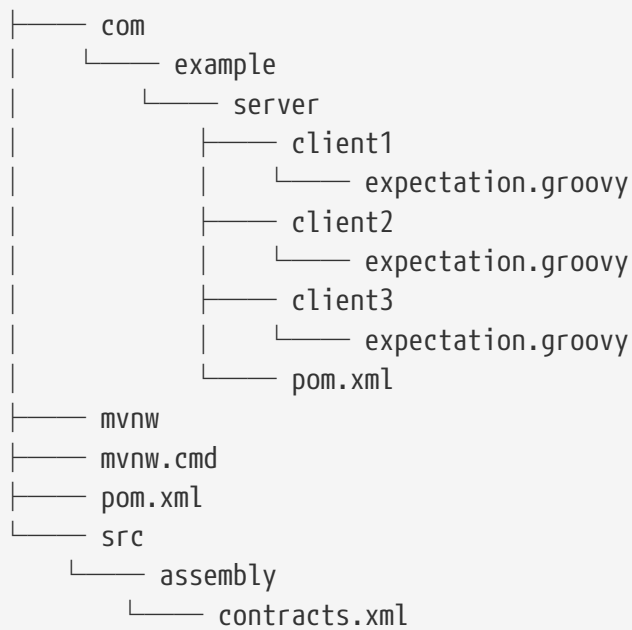
You can also pass those values also in properties from your deployment pipeline.

8.5. How Can I use a Common Repository with Contracts Instead of Storing Them with the Producer?

Another way of storing contracts, rather than having them with the producer, is to keep them in a common place. This situation can be related to security issues (where the consumers cannot clone the producer's code). Also, if you keep contracts in a single place, then you, as a producer, know how many consumers you have and which consumer you may break with your local changes.

8.5.1. Repo Structure

Assume that we have a producer with coordinates of `com.example:server` and three consumers: `client1`, `client2`, and `client3`. Then, in the repository with common contracts, you could have the following setup (which you can check out [here](#)). The following listing shows such a structure:



Under the slash-delimited `groupid/artifact id` folder (`com/example/server`), you have expectations of the three consumers (`client1`, `client2`, and `client3`). Expectations are the standard Groovy DSL contract files, as described throughout this documentation. This repository has to produce a JAR file that maps one-to-one to the contents of the repository.

The following example shows a `pom.xml` file inside the `server` folder:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns="http://maven.apache.org/POM/4.0.0"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
https://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <groupId>com.example</groupId>
  <artifactId>server</artifactId>
  <version>0.0.1</version>

  <name>Server Stubs</name>
  <description>POM used to install locally stubs for consumer side</description>

  <parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>3.2.0-SNAPSHOT</version>
    <relativePath/>
  </parent>

  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <java.version>17</java.version>
    <spring-cloud-contract.version>4.1.0-SNAPSHOT</spring-cloud-
contract.version>
    <spring-cloud-release.version>2023.0.0-SNAPSHOT</spring-cloud-
release.version>
    <excludeBuildFolders>true</excludeBuildFolders>
  </properties>

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

  <build>
    <plugins>
      <plugin>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-contract-maven-plugin</artifactId>
        <version>${spring-cloud-contract.version}</version>
```

```

        <extensions>true</extensions>
        <configuration>
            <!-- By default it would search under src/test/resources/ -->
            <contractsDirectory>${project.basedir}</contractsDirectory>
        </configuration>
    </plugin>
</plugins>
</build>

<repositories>
    <repository>
        <id>spring-snapshots</id>
        <name>Spring Snapshots</name>
        <url>https://repo.spring.io/snapshot</url>
        <snapshots>
            <enabled>true</enabled>
        </snapshots>
    </repository>
    <repository>
        <id>spring-milestones</id>
        <name>Spring Milestones</name>
        <url>https://repo.spring.io/milestone</url>
        <snapshots>
            <enabled>>false</enabled>
        </snapshots>
    </repository>
    <repository>
        <id>spring-releases</id>
        <name>Spring Releases</name>
        <url>https://repo.spring.io/release</url>
        <snapshots>
            <enabled>>false</enabled>
        </snapshots>
    </repository>
</repositories>
<pluginRepositories>
    <pluginRepository>
        <id>spring-snapshots</id>
        <name>Spring Snapshots</name>
        <url>https://repo.spring.io/snapshot</url>
        <snapshots>
            <enabled>true</enabled>
        </snapshots>
    </pluginRepository>
    <pluginRepository>
        <id>spring-milestones</id>
        <name>Spring Milestones</name>
        <url>https://repo.spring.io/milestone</url>
        <snapshots>
            <enabled>>false</enabled>
        </snapshots>
    </pluginRepository>

```

```
</pluginRepository>
<pluginRepository>
  <id>spring-releases</id>
  <name>Spring Releases</name>
  <url>https://repo.spring.io/release</url>
  <snapshots>
    <enabled>false</enabled>
  </snapshots>
</pluginRepository>
</pluginRepositories>

</project>
```

There are no dependencies other than the Spring Cloud Contract Maven Plugin. Those `pom.xml` files are necessary for the consumer side to run `mvn clean install -DskipTests` to locally install the stubs of the producer project.

The `pom.xml` file in the root folder can look like the following:

```

<?xml version="1.0" encoding="UTF-8"?>
<project xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns="http://maven.apache.org/POM/4.0.0"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
https://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <groupId>com.example.standalone</groupId>
  <artifactId>contracts</artifactId>
  <version>0.0.1</version>

  <name>Contracts</name>
  <description>Contains all the Spring Cloud Contracts, well, contracts. JAR
used by the
  producers to generate tests and stubs
  </description>

  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
  </properties>

  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-assembly-plugin</artifactId>
        <executions>
          <execution>
            <id>contracts</id>
            <phase>prepare-package</phase>
            <goals>
              <goal>single</goal>
            </goals>
            <configuration>
              <attach>true</attach>
            </configuration>
          </execution>
        </executions>
      </plugin>
    </plugins>

    <descriptor>${basedir}/src/assembly/contracts.xml</descriptor>
    <!-- If you want an explicit classifier remove the
following line -->
      <appendAssemblyId>false</appendAssemblyId>
    </configuration>
  </executions>
</build>

</project>

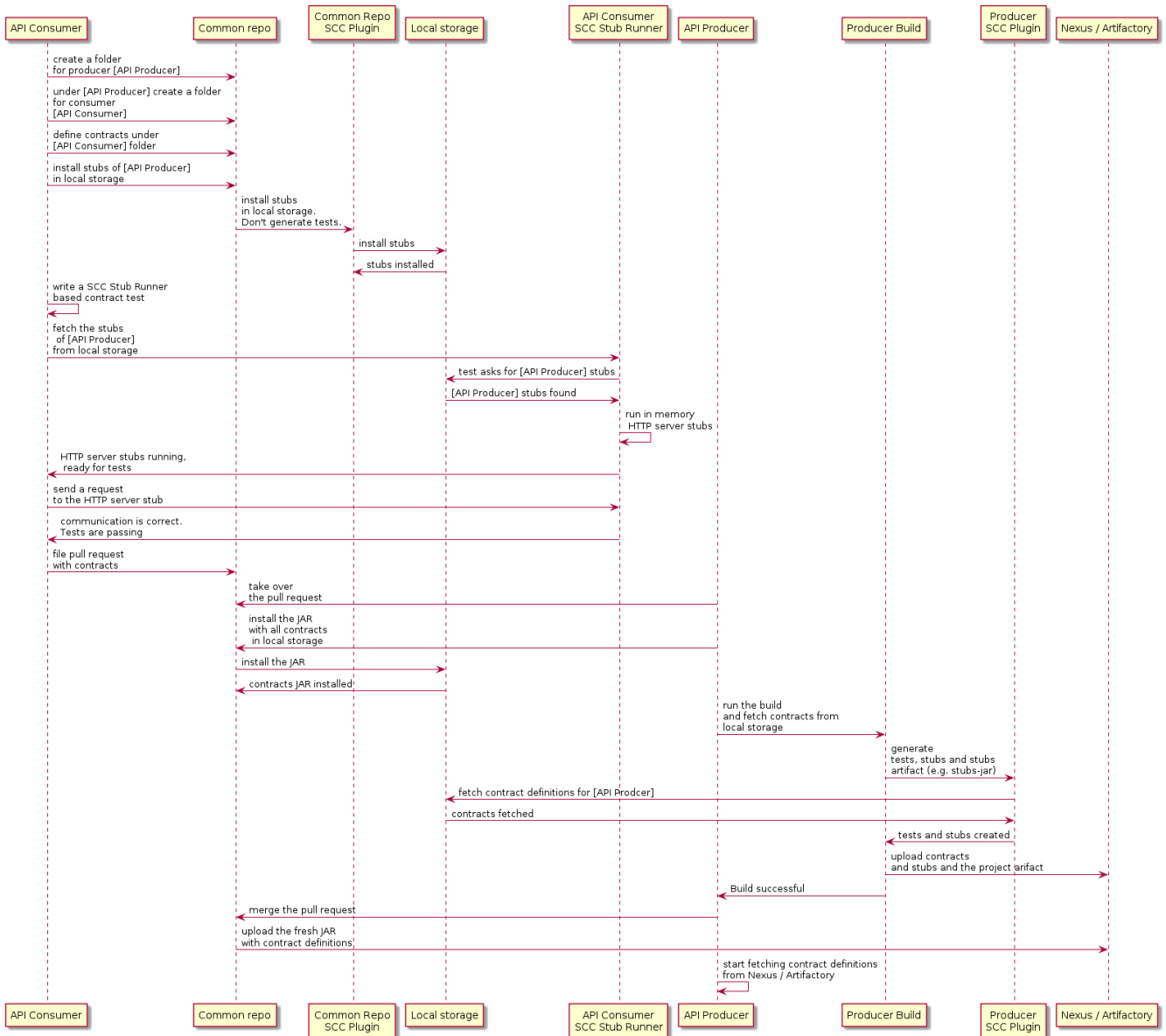
```

It uses the assembly plugin to build the JAR with all the contracts. The following example shows such a setup:

```
<assembly xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns="http://maven.apache.org/plugins/maven-assembly-
  plugin/assembly/1.1.3"
  xsi:schemaLocation="http://maven.apache.org/plugins/maven-assembly-
  plugin/assembly/1.1.3 https://maven.apache.org/xsd/assembly-1.1.3.xsd">
  <id>project</id>
  <formats>
    <format>jar</format>
  </formats>
  <includeBaseDirectory>>false</includeBaseDirectory>
  <fileSets>
    <fileSet>
      <directory>${project.basedir}</directory>
      <outputDirectory></outputDirectory>
      <useDefaultExcludes>>true</useDefaultExcludes>
      <excludes>
        <exclude>**/${project.build.directory}/**</exclude>
        <exclude>mvnw</exclude>
        <exclude>mvnw.cmd</exclude>
        <exclude>.mvn/**</exclude>
        <exclude>src/**</exclude>
      </excludes>
    </fileSet>
  </fileSets>
</assembly>
```

8.5.2. Workflow

The workflow assumes that Spring Cloud Contract is set up both on the consumer and on the producer side. There is also the proper plugin setup in the common repository with contracts. The CI jobs are set for a common repository to build an artifact of all contracts and upload it to Nexus or Artifactory. The following image shows the UML for this workflow:



8.5.3. Consumer

When the consumer wants to work on the contracts offline, instead of cloning the producer code, the consumer team clones the common repository, goes to the required producer's folder (for example, `com/example/server`) and runs `mvn clean install -DskipTests` to locally install the stubs converted from the contracts.



You need to have [Maven installed locally](#).

8.5.4. Producer

As a producer, you can alter the Spring Cloud Contract Verifier to provide the URL and the dependency of the JAR that contains the contracts, as follows:

```
<plugin>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-contract-maven-plugin</artifactId>
  <configuration>
    <contractsMode>REMOTE</contractsMode>
    <contractsRepositoryUrl>
      https://link/to/your/nexus/or/artifactory/or/sth
    </contractsRepositoryUrl>
    <contractDependency>
      <groupId>com.example.standalone</groupId>
      <artifactId>contracts</artifactId>
    </contractDependency>
  </configuration>
</plugin>
```

With this setup, the JAR with a `groupid` of `com.example.standalone` and an `artifactid` of `contracts` is downloaded from `link/to/your/nexus/or/artifactory/or/sth`. It is then unpacked in a local temporary folder, and the contracts present in `com/example/server` are picked as the ones used to generate the tests and the stubs. Due to this convention, the producer team can know which consumer teams are broken when some incompatible changes are made.

The rest of the flow looks the same.

8.5.5. How Can I Define Messaging Contracts per Topic Rather than per Producer?

To avoid messaging contracts duplication in the common repository, when a few producers write messages to one topic, we could create a structure in which the REST contracts are placed in a folder per producer and messaging contracts are placed in the folder per topic.

For Maven Projects

To make it possible to work on the producer side, we should specify an inclusion pattern for filtering common repository jar files by messaging topics we are interested in. The `includedFiles` property of the Maven Spring Cloud Contract plugin lets us do so. Also, `contractsPath` need to be specified, since the default path would be the common repository `groupid/artifactid`. The following example shows a Maven plugin for Spring Cloud Contract:


```

<plugin>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-contract-maven-plugin</artifactId>
  <version>${spring-cloud-contract.version}</version>
  <configuration>
    <contractsMode>REMOTE</contractsMode>

    <contractsRepositoryUrl>https://link/to/your/nexus/or/artifactory/or/sth</contractsRepositoryUrl>
    <contractDependency>
      <groupId>com.example</groupId>
      <artifactId>common-repo-with-contracts</artifactId>
      <version>+</version>
    </contractDependency>
    <contractsPath></contractsPath>
    <baseClassMappings>
      <baseClassMapping>
        <contractPackageRegex>.*messaging.*</contractPackageRegex>
        <baseClassFQN>com.example.services.MessagingBase</baseClassFQN>
      </baseClassMapping>
      <baseClassMapping>
        <contractPackageRegex>.*rest.*</contractPackageRegex>
        <baseClassFQN>com.example.services.TestBase</baseClassFQN>
      </baseClassMapping>
    </baseClassMappings>
    <includedFiles>
      <includedFile>**/${project.artifactId}/**</includedFile>
      <includedFile>**/${first-topic}/**</includedFile>
      <includedFile>**/${second-topic}/**</includedFile>
    </includedFiles>
  </configuration>
</plugin>

```



Many of the values in the preceding Maven plugin can be changed. We included it for illustration purposes rather than trying to provide a “typical” example.

For Gradle Projects

To work with a Gradle project:

1. Add a custom configuration for the common repository dependency, as follows:

```
ext {
    contractsGroupId = "com.example"
    contractsArtifactId = "common-repo"
    contractsVersion = "1.2.3"
}

configurations {
    contracts {
        transitive = false
    }
}
```

2. Add the common repository dependency to your classpath, as follows:

```
dependencies {
    contracts "${contractsGroupId}:${contractsArtifactId}:${contractsVersion}"
    testCompile
    "${contractsGroupId}:${contractsArtifactId}:${contractsVersion}"
}
```

3. Download the dependency to an appropriate folder, as follows:

```
task getContracts(type: Copy) {
    from configurations.contracts
    into new File(project.buildDir, "downloadedContracts")
}
```

4. Unzip the JAR, as follows:

```
task unzipContracts(type: Copy) {
    def zipFile = new File(project.buildDir,
        "downloadedContracts/${contractsArtifactId}-${contractsVersion}.jar")
    def outputDir = file("${buildDir}/unpackedContracts")

    from zipTree(zipFile)
    into outputDir
}
```

5. Cleanup unused contracts, as follows:

```
task deleteUnwantedContracts(type: Delete) {
    delete fileTree(dir: "${buildDir}/unpackedContracts",
        include: "**/*",
        excludes: [
            "**/${project.name}/**",
            "**/${first-topic}/**",
            "**/${second-topic}/**"])
}
```

6. Create task dependencies, as follows:

```
unzipContracts.dependsOn("getContracts")
deleteUnwantedContracts.dependsOn("unzipContracts")
build.dependsOn("deleteUnwantedContracts")
```

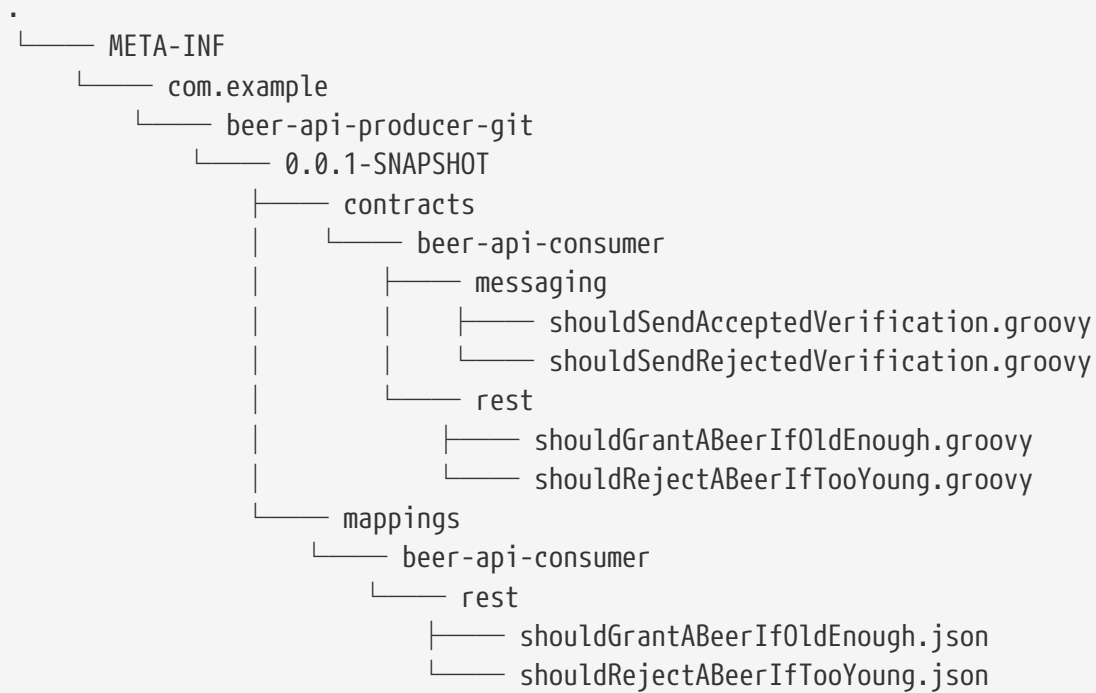
7. Configure the plugin by specifying the directory that contains the contracts, by setting the `contractsDslDir` property, as follows:

```
contracts {
    contractsDslDir = new File("${buildDir}/unpackedContracts")
}
```

8.6. How Can I Use Git as the Storage for Contracts and Stubs?

In the polyglot world, there are languages that do not use binary storage, as Artifactory and Nexus do. Starting from Spring Cloud Contract version 2.0.0, we provide mechanisms to store contracts and stubs in a SCM (Source Control Management) repository. Currently, the only supported SCM is Git.

The repository would have to have the following setup (which you can checkout from [here](#)):



Under the **META-INF** folder:

- We group applications by **groupId** (such as **com.example**).
- Each application is represented by its **artifactId** (for example, **beer-api-producer-git**).
- Next, each application is organized by its version (such as **0.0.1-SNAPSHOT**). Starting from Spring Cloud Contract version **2.1.0**, you can specify the versions as follows (assuming that your versions follow semantic versioning):
 - **+** or **latest**: To find the latest version of your stubs (assuming that the snapshots are always the latest artifact for a given revision number). That means:
 - If you have **1.0.0.RELEASE**, **2.0.0.BUILD-SNAPSHOT**, and **2.0.0.RELEASE**, we assume that the latest is **2.0.0.BUILD-SNAPSHOT**.
 - If you have **1.0.0.RELEASE** and **2.0.0.RELEASE**, we assume that the latest is **2.0.0.RELEASE**.
 - If you have a version called **latest** or **+**, we will pick that folder.
 - **release**: To find the latest release version of your stubs. That means:
 - If you have **1.0.0.RELEASE**, **2.0.0.BUILD-SNAPSHOT**, and **2.0.0.RELEASE** we assume that the latest is **2.0.0.RELEASE**.
 - If you have a version called **release**, we pick that folder.

Finally, there are two folders:

- **contracts**: The good practice is to store the contracts required by each consumer in the folder with the consumer name (such as **beer-api-consumer**). That way, you can use the **stubs-per-consumer** feature. Further directory structure is arbitrary.
- **mappings**: The Maven or Gradle Spring Cloud Contract plugins push the stub server mappings in

this folder. On the consumer side, Stub Runner scans this folder to start stub servers with stub definitions. The folder structure is a copy of the one created in the `contracts` subfolder.

8.6.1. Protocol Convention

To control the type and location of the source of contracts (whether binary storage or an SCM repository), you can use the protocol in the URL of the repository. Spring Cloud Contract iterates over registered protocol resolvers and tries to fetch the contracts (by using a plugin) or stubs (from Stub Runner).

For the SCM functionality, currently, we support the Git repository. To use it, in the property where the repository URL needs to be placed, you have to prefix the connection URL with `git://`. The following listing shows some examples:

```
git://file:///foo/bar
git://https://github.com/spring-cloud-samples/spring-cloud-contract-nodejs-
contracts-git.git
git://git@github.com:spring-cloud-samples/spring-cloud-contract-nodejs-contracts-
git.git
```

8.6.2. Producer

For the producer, to use the SCM (Source Control Management) approach, we can reuse the same mechanism we use for external contracts. We route Spring Cloud Contract to use the SCM implementation from the URL that starts with the `git://` protocol.



You have to manually add the `pushStubsToScm` goal in Maven or use (bind) the `pushStubsToScm` task in Gradle. We do not push stubs to the `origin` of your git repository.

The following listing includes the relevant parts both Maven and Gradle build files:

Maven

```
<plugin>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-contract-maven-plugin</artifactId>
  <version>${spring-cloud-contract.version}</version>
  <extensions>>true</extensions>
  <configuration>
    <!-- Base class mappings etc. -->

    <!-- We want to pick contracts from a Git repository -->
    <contractsRepositoryUrl>git://https://github.com/spring-cloud-
samples/spring-cloud-contract-nodejs-contracts-git.git</contractsRepositoryUrl>

    <!-- We reuse the contract dependency section to set up the path
to the folder that contains the contract definitions. In our case the
path will be /groupId/artifactId/version/contracts -->
    <contractDependency>
      <groupId>${project.groupId}</groupId>
      <artifactId>${project.artifactId}</artifactId>
      <version>${project.version}</version>
    </contractDependency>

    <!-- The contracts mode can't be classpath -->
    <contractsMode>REMOTE</contractsMode>
  </configuration>
  <executions>
    <execution>
      <phase>package</phase>
      <goals>
        <!-- By default we will not push the stubs back to SCM,
you have to explicitly add it as a goal -->
        <goal>pushStubsToScm</goal>
      </goals>
    </execution>
  </executions>
</plugin>
```

Gradle

```
contracts {
    // We want to pick contracts from a Git repository
    contractDependency {
        stringNotation = "${project.group}:${project.name}:${project.version}"
    }
    /*
    We reuse the contract dependency section to set up the path
    to the folder that contains the contract definitions. In our case the
    path will be /groupId/artifactId/version/contracts
    */
    contractRepository {
        repositoryUrl = "git://https://github.com/spring-cloud-samples/spring-
cloud-contract-nodejs-contracts-git.git"
    }
    // The mode can't be classpath
    contractsMode = "REMOTE"
    // Base class mappings etc.
}

/*
In this scenario we want to publish stubs to SCM whenever
the `publish` task is invoked
*/
publish.dependsOn("publishStubsToScm")
```

You can also further customize the `publishStubsToScm` gradle task. In the following example, the task is customized to pick contracts from a local git repository:

gradle

```
publishStubsToScm {
    // We want to modify the default set up of the plugin when publish stubs to
    scm is called
    // We want to pick contracts from a Git repository
    contractDependency {
        stringNotation = "${project.group}:${project.name}:${project.version}"
    }
    /*
    We reuse the contract dependency section to set up the path
    to the folder that contains the contract definitions. In our case the
    path will be /groupId/artifactId/version/contracts
    */
    contractRepository {
        repositoryUrl = "git://file://${new File(project.rootDir,
"../target")}/contract_empty_git/"
    }
    // We set the contracts mode to `LOCAL`
    contractsMode = "LOCAL"
}
```

IMPORTANT

Starting with the `2.3.0.RELEASE`, the `customize{}` closure previously used for the `publishStubsToScm` customization is no longer available. The settings should be applied directly within the `publishStubsToScm` closure, as in the preceding example.

With such a setup:

- A git project is cloned to a temporary directory
- The SCM stub downloader goes to the `META-INF/groupId/artifactId/version/contracts` folder to find contracts. For example, for `com.example:foo:1.0.0`, the path would be `META-INF/com.example/foo/1.0.0/contracts`.
- Tests are generated from the contracts.
- Stubs are created from the contracts.
- Once the tests pass, the stubs are committed in the cloned repository.
- Finally, a push is sent to that repo's `origin`.

8.6.3. Producer with Contracts Stored Locally

Another option to use the SCM as the destination for stubs and contracts is to store the contracts locally, with the producer, and only push the contracts and the stubs to SCM. The following listing shows the setup required to achieve this with Maven and Gradle:


```

<plugin>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-contract-maven-plugin</artifactId>
  <version>${spring-cloud-contract.version}</version>
  <extensions>>true</extensions>
  <!-- In the default configuration, we want to use the contracts stored locally -->
  <configuration>
    <baseClassMappings>
      <baseClassMapping>
        <contractPackageRegex>.*messaging.*</contractPackageRegex>
        <baseClassFQN>com.example.BeerMessagingBase</baseClassFQN>
      </baseClassMapping>
      <baseClassMapping>
        <contractPackageRegex>.*rest.*</contractPackageRegex>
        <baseClassFQN>com.example.BeerRestBase</baseClassFQN>
      </baseClassMapping>
    </baseClassMappings>
    <basePackageForTests>com.example</basePackageForTests>
  </configuration>
  <executions>
    <execution>
      <phase>package</phase>
      <goals>
        <!-- By default we will not push the stubs back to SCM,
              you have to explicitly add it as a goal -->
        <goal>pushStubsToScm</goal>
      </goals>
      <configuration>
        <!-- We want to pick contracts from a Git repository -->
        <contractsRepositoryUrl>git://file://${env.ROOT}/target/contract_empty_git/
          </contractsRepositoryUrl>
        <!-- Example of URL via git protocol -->
        <!--<contractsRepositoryUrl>git://git@github.com:spring-cloud-
samples/spring-cloud-contract-samples.git</contractsRepositoryUrl>-->
        <!-- Example of URL via http protocol -->
        <!--<contractsRepositoryUrl>git://https://github.com/spring-cloud-
samples/spring-cloud-contract-samples.git</contractsRepositoryUrl>-->
        <!-- We reuse the contract dependency section to set up the path
              to the folder that contains the contract definitions. In our case
              the
              path will be /groupId/artifactId/version/contracts -->
        <contractDependency>
          <groupId>${project.groupId}</groupId>
          <artifactId>${project.artifactId}</artifactId>
          <version>${project.version}</version>
        </contractDependency>

```

```

        <!-- The mode can't be classpath -->
        <contractsMode>LOCAL</contractsMode>
    </configuration>
</execution>
</executions>
</plugin>

```

Gradle

```

contracts {
    contractsDslDir = file("src/test/resources/contracts")
    testFramework = "JUNIT5"
    // Base package for generated tests
    basePackageForTests = "com.example"
    baseClassMappings {
        baseClassMapping(".*messaging.*", "com.example.BeerMessagingBase")
        baseClassMapping(".*rest.*", "com.example.BeerRestBase")
    }

    /*
    In this scenario we want to publish stubs to SCM whenever
    the `publish` task is executed
    */
    publishStubsToScm {

        // We want to pick contracts from a Git repository
        contractDependency {
            stringNotation = "${project.group}:${project.name}:${project.version}"
        }

        contractRepository {
            /*
            We reuse the contract dependency section to set up the path
            to the folder that contains the contract definitions. In our case the
            path will be /groupId/artifactId/version/contracts
            */
            repositoryUrl = "git://file://${new File(project.rootDir,
            "../target")}/contract_empty_git/"
        }
    }
}

```

With such a setup:

- Contracts from the default `src/test/resources/contracts` directory are picked.
- Tests are generated from the contracts.
- Stubs are created from the contracts.
- Once the tests pass:

- The git project is cloned to a temporary directory.
- The stubs and contracts are committed in the cloned repository.
- Finally, a push is done to that repository's **origin**.

8.6.4. Keeping Contracts with the Producer and Stubs in an External Repository

You can also keep the contracts in the producer repository but keep the stubs in an external git repository. This is most useful when you want to use the base consumer-producer collaboration flow but cannot use an artifact repository to store the stubs.

To do so, use the usual producer setup and then add the `pushStubsToScm` goal and set `contractsRepositoryUrl` to the repository where you want to keep the stubs.

8.6.5. Consumer

On the consumer side, when passing the `repositoryRoot` parameter, either from the `@AutoConfigureStubRunner` annotation, the JUnit 4 rule, JUnit 5 extension, or properties, you can pass the URL of the SCM repository, prefixed with the `git://` protocol. The following example shows how to do so:

```
@AutoConfigureStubRunner(  
    stubsMode="REMOTE",  
    repositoryRoot="git://https://github.com/spring-cloud-samples/spring-cloud-  
contract-nodejs-contracts-git.git",  
    ids="com.example:bookstore:0.0.1.RELEASE"  
)
```

With such a setup:

- The git project is cloned to a temporary directory.
- The SCM stub downloader goes to the `META-INF/groupId/artifactId/version/` folder to find stub definitions and contracts. For example, for `com.example:foo:1.0.0`, the path would be `META-INF/com.example/foo/1.0.0/`.
- Stub servers are started and fed with mappings.
- Messaging definitions are read and used in the messaging tests.

8.7. How Can I Debug the Request/Response Being Sent by the Generated Tests Client?

The generated tests all boil down to RestAssured in some form or fashion. RestAssured relies on the [Apache HttpClient](#). HttpClient has a facility called [wire logging](#), which logs the entire request and response to HttpClient. Spring Boot has a logging [common application property](#) for doing this sort

of thing. To use it, add it to your application properties, as follows:

```
logging.level.org.apache.http.wire=DEBUG
```

8.8. How Can I Debug the Mapping, Request, or Response Being Sent by WireMock?

Starting from version **1.2.0**, we set WireMock logging to **info** and set the WireMock notifier to being verbose. Now you can exactly know what request was received by the WireMock server and which matching response definition was picked.

To turn off this feature, set WireMock logging to **ERROR**, as follows:

```
logging.level.com.github.tomakehurst.wiremock=ERROR
```

8.9. How Can I See What Got Registered in the HTTP Server Stub?

You can use the `mappingsOutputFolder` property on `@AutoConfigureStubRunner`, `StubRunnerRule`, or `StubRunnerExtension` to dump all mappings for each artifact ID. Also, the port at which the given stub server was started is attached.

8.10. How Can I Reference Text from File?

In version 1.2.0, we added this ability. You can call a `file(...)` method in the DSL and provide a path relative to where the contract lies. If you use YAML, you can use the `bodyFromFile` property.

8.11. How Can I Generate Pact, YAML, or X files from Spring Cloud Contract Contracts?

Spring Cloud Contract comes with a `ToFileContractsTransformer` class that lets you dump contracts as files for the given `ContractConverter`. It contains a `static void main` method that lets you run the transformer as an executable. It takes the following arguments:

- argument 1 : **FQN**: Fully qualified name of the `ContractConverter` (for example, `PactContractConverter`). **REQUIRED**.
- argument 2 : **path**: Path where the dumped files should be stored. **OPTIONAL**—defaults to `target/converted-contracts`.
- argument 3 : **path**: Path where the contracts should be searched for. **OPTIONAL**—defaults to

`src/test/resources/contracts.`

After calling the transformer, the Spring Cloud Contract files are processed and, depending on the provided FQN of the `ContractTransformer`, the contracts are transformed to the required format and dumped to the provided folder.

The following example shows how to configure Pact integration for both Maven and Gradle:

Maven

```
<plugin>
  <groupId>org.codehaus.mojo</groupId>
  <artifactId>exec-maven-plugin</artifactId>
  <version>1.6.0</version>
  <executions>
    <execution>
      <id>convert-dsl-to-pact</id>
      <phase>process-test-classes</phase>
      <configuration>
        <classpathScope>test</classpathScope>
        <mainClass>
          org.springframework.cloud.contract.verifier.util.ToFileContractsTransformer
        </mainClass>
        <arguments>
          <argument>
            org.springframework.cloud.contract.verifier.spec.pact.PactContractConverter
          </argument>
          <argument>${project.basedir}/target/pacts</argument>
          <argument>
            ${project.basedir}/src/test/resources/contracts
          </argument>
        </arguments>
      </configuration>
      <goals>
        <goal>java</goal>
      </goals>
    </execution>
  </executions>
</plugin>
```

Gradle

```
task convertContracts(type: JavaExec) {
  main =
  "org.springframework.cloud.contract.verifier.util.ToFileContractsTransformer"
  classpath = sourceSets.test.compileClasspath

  args("org.springframework.cloud.contract.verifier.spec.pact.PactContractConverter"
    ,
    "${project.rootDir}/build/pacts",
    "${project.rootDir}/src/test/resources/contracts")
}

test.dependsOn("convertContracts")
```

8.12. How Can I Work with Transitive Dependencies?

The Spring Cloud Contract plugins add the tasks that create the stubs jar for you. One problem that arises is that, when reusing the stubs, you can mistakenly import all of that stub's dependencies. When building a Maven artifact, even though you have a couple of different jars, all of them share one `pom.xml` file, as the following listing shows:

```
|—— producer-0.0.1.BUILD-20160903.075506-1-stubs.jar
|—— producer-0.0.1.BUILD-20160903.075506-1-stubs.jar.sha1
|—— producer-0.0.1.BUILD-20160903.075655-2-stubs.jar
|—— producer-0.0.1.BUILD-20160903.075655-2-stubs.jar.sha1
|—— producer-0.0.1.BUILD-SNAPSHOT.jar
|—— producer-0.0.1.BUILD-SNAPSHOT.pom
|—— producer-0.0.1.BUILD-SNAPSHOT-stubs.jar
|—— ...
|—— ...
```

There are three possibilities of working with those dependencies so as not to have any issues with transitive dependencies:

- Mark all application dependencies as optional
- Create a separate `artifactId` for the stubs
- Exclude dependencies on the consumer side

8.12.1. How Can I Mark All Application Dependencies as Optional?

If, in the `producer` application, you mark all of your dependencies as optional, when you include the `producer` stubs in another application (or when that dependency gets downloaded by Stub Runner), then, since all of the dependencies are optional, they do not get downloaded.

8.12.2. How can I Create a Separate `artifactId` for the Stubs?

If you create a separate `artifactId`, you can set it up in whatever way you wish. For example, you might decide to have no dependencies at all.

8.12.3. How can I Exclude Dependencies on the Consumer Side?

As a consumer, if you add the stub dependency to your classpath, you can explicitly exclude the unwanted dependencies.

8.13. How Can I Generate Spring REST Docs Snippets from the Contracts?

When you want to include the requests and responses of your API by using Spring REST Docs, you only need to make some minor changes to your setup if you are using `MockMvc` and

RestAssuredMockMvc. To do so, include the following dependencies (if you have not already done so):

maven

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-contract-verifier</artifactId>
  <scope>test</scope>
</dependency>
<dependency>
  <groupId>org.springframework.restdocs</groupId>
  <artifactId>spring-restdocs-mockmvc</artifactId>
  <optional>true</optional>
</dependency>
```

gradle

```
testImplementation 'org.springframework.cloud:spring-cloud-starter-contract-
verifier'
testImplementation 'org.springframework.restdocs:spring-restdocs-mockmvc'
```

Next, you need to make some changes to your base class. The following examples use [WebApplicationContext](#) and the standalone option with RestAssured:

WebApplicationContext

```
package com.example.fraud;

import io.restassured.module.mockmvc.RestAssuredMockMvc;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.TestInfo;
import org.junit.jupiter.api.extension.ExtendWith;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.restdocs.RestDocumentationContextProvider;
import org.springframework.restdocs.RestDocumentationExtension;
import org.springframework.test.web.servlet.setup.MockMvcBuilders;
import org.springframework.web.context.WebApplicationContext;

import static
org.springframework.restdocs.mockmvc.MockMvcRestDocumentation.document;
import static
org.springframework.restdocs.mockmvc.MockMvcRestDocumentation.documentationConfigu
ration;

@ExtendWith(RestDocumentationExtension.class)
@SpringBootTest(classes = Application.class)
public abstract class FraudBaseWithWebAppSetup {

    @Autowired
    private WebApplicationContext context;

    @BeforeEach
    public void setup(TestInfo info, RestDocumentationContextProvider
restDocumentation) {

RestAssuredMockMvc.mockMvc(MockMvcBuilders.webAppContextSetup(this.context)
    .apply(documentationConfiguration(restDocumentation))
    .alwaysDo(document(
        getClass().getSimpleName() + "_" + info.getDisplayName()))
    .build());
    }

    protected void assertThatRejectionReasonIsNull(Object rejectionReason) {
        assert rejectionReason == null;
    }
}
```

Standalone

```
package com.example.fraud;

import io.restassured.module.mockmvc.RestAssuredMockMvc;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.TestInfo;
import org.junit.jupiter.api.extension.ExtendWith;

import org.springframework.restdocs.RestDocumentationContextProvider;
import org.springframework.restdocs.RestDocumentationExtension;
import org.springframework.test.web.servlet.setup.MockMvcBuilders;

import static
org.springframework.restdocs.mockmvc.MockMvcRestDocumentation.document;
import static
org.springframework.restdocs.mockmvc.MockMvcRestDocumentation.documentationConfigu
ration;

@ExtendWith(RestDocumentationExtension.class)
public abstract class FraudBaseWithStandaloneSetup {

    @BeforeEach
    public void setup(TestInfo info, RestDocumentationContextProvider
restDocumentation) {
        RestAssuredMockMvc.standaloneSetup(MockMvcBuilders
            .standaloneSetup(new FraudDetectionController())
            .apply(documentationConfiguration(restDocumentation))
            .alwaysDo(document(
                getClass().getSimpleName() + "_" +
info.getDisplayName())));
    }
}
```



You need not specify the output directory for the generated snippets (since version 1.2.0.RELEASE of Spring REST Docs).

8.14. How Can I Use Stubs from a Location

If you want to fetch contracts or stubs from a given location without cloning a repository or fetching a JAR, use the `stubs://` protocol when providing the repository root argument for Stub Runner or the Spring Cloud Contract plugin. You can read more about this in [this section](#) of the documentation.

8.15. How Can I Generate Stubs at Runtime

If you want to generate stubs at runtime for contracts, switch the `generateStubs` property in the `@AutoConfigureStubRunner` annotation, or call the `withGenerateStubs(true)` method on the JUnit Rule or Extension. You can read more about this in [this section](#) of the documentation.

8.16. How Can I Make The Build Pass if There Are No Contracts or Stubs

If you want Stub Runner not to fail if no stubs were found, switch the `generateStubs` property in the `@AutoConfigureStubRunner` annotation or call the `withFailOnNoStubs(false)` method on the JUnit Rule or Extension. You can read more about this in [this section](#) of the documentation.

If you want the plugins not to fail the build when no contracts were found, you can set the `failOnNoStubs` flag in Maven or call the `contractRepository { failOnNoStubs(false) }` closure in Gradle.

8.17. How Can I Mark that a Contract Is in Progress

If a contract is in progress, it means that the, on the producer side, tests are not generated, but the stub is generated. You can read more about this in [this section](#) of the documentation.

In a CI build, before going to production, you would like to ensure that no in-progress contracts are on the classpath, because they may lead to false positives. For this reason, by default, in the Spring Cloud Contract plugin, we set the value of `failOnInProgress` to `true`. If you want to allow such contracts when tests are to be generated, set the flag to `false`.

Appendix A: Common application properties

Various properties can be specified inside your `application.properties` file, inside your `application.yml` file, or as command line switches. This appendix provides a list of common Spring Cloud Contract properties and references to the underlying classes that consume them.



Property contributions can come from additional jar files on your classpath, so you should not consider this an exhaustive list. Also, you can define your own properties.

8.A.1. Default application properties

Name	Default	Description
<code>stubrunner.amqp.enabled</code>	<code>false</code>	Whether to enable support for Stub Runner and AMQP.
<code>stubrunner.amqp.mockConnection</code>	<code>true</code>	Whether to enable support for Stub Runner and AMQP mocked connection factory.

Name	Default	Description
stubrunner.classifier	stubs	The classifier to use by default in ivy co-ordinates for a stub.
stubrunner.cloud.consul.enabled	true	Whether to enable stubs registration in Consul.
stubrunner.cloud.delegate.enabled	true	Whether to enable DiscoveryClient's Stub Runner implementation.
stubrunner.cloud.enabled	true	Whether to enable Spring Cloud support for Stub Runner.
stubrunner.cloud.eureka.enabled	true	Whether to enable stubs registration in Eureka.
stubrunner.cloud.loadbalancer.enabled	true	Whether to enable Stub Runner's Spring Cloud Load Balancer integration.
stubrunner.cloud.stubbed.discovery.enabled	true	Whether Service Discovery should be stubbed for Stub Runner. If set to false, stubs will get registered in real service discovery.
stubrunner.cloud.zookeeper.enabled	true	Whether to enable stubs registration in Zookeeper.
stubrunner.consumer-name		You can override the default <code>spring.application.name</code> of this field by setting a value to this parameter.
stubrunner.delete-stubs-after-test	true	If set to <code>false</code> will NOT delete stubs from a temporary folder after running tests.
stubrunner.fail-on-no-stubs	true	When enabled, this flag will tell stub runner to throw an exception when no stubs / contracts were found.
stubrunner.generate-stubs	false	When enabled, this flag will tell stub runner to not load the generated stubs, but convert the found contracts at runtime to a stub format and run those stubs.
stubrunner.http-server-stub-configurer		Configuration for an HTTP server stub.

Name	Default	Description
stubrunner.ids	<code>[]</code>	The ids of the stubs to run in "ivy" notation ([groupId]:artifactId:[version]:[classifier]:[port]). {@code groupId}, {@code classifier}, {@code version} and {@code port} can be optional.
stubrunner.ids-to-service-ids		Mapping of Ivy notation based ids to serviceIds inside your application. Example "a:b" → "myService" "artifactId" → "myOtherService"
stubrunner.integration.enabled	<code>true</code>	Whether to enable Stub Runner integration with Spring Integration.
stubrunner.jms.enabled	<code>true</code>	Whether to enable Stub Runner integration with Spring JMS.
stubrunner.kafka.enabled	<code>true</code>	Whether to enable Stub Runner integration with Spring Kafka.
stubrunner.kafka.initializer.enabled	<code>true</code>	Whether to allow Stub Runner to take care of polling for messages instead of the KafkaStubMessages component. The latter should be used only on the producer side.
stubrunner.mappings-output-folder		Dumps the mappings of each HTTP server to the selected folder.
stubrunner.max-port	<code>15000</code>	Max value of a port for the automatically started WireMock server.
stubrunner.min-port	<code>10000</code>	Min value of a port for the automatically started WireMock server.
stubrunner.password		Repository password.
stubrunner.properties		Map of properties that can be passed to custom {@link org.springframework.cloud.contract.stubrunner.StubDownloaderBuilder}.
stubrunner.proxy-host		Repository proxy host.

Name	Default	Description
stubrunner.proxy-port		Repository proxy port.
stubrunner.server-id		
stubrunner.stream.enabled	true	Whether to enable Stub Runner integration with Spring Cloud Stream.
stubrunner.stubs-mode		Pick where the stubs should come from.
stubrunner.stubs-per-consumer	false	Should only stubs for this particular consumer get registered in HTTP server stub.
stubrunner.username		Repository username.
wiremock.placeholder.enabled	true	Flag to indicate that http URLs in generated wiremock stubs should be filtered to add or resolve a placeholder for a dynamic port.
wiremock.reset-mappings-after-each-test	false	
wiremock.rest-template-ssl-enabled	false	
wiremock.server.files	[]	
wiremock.server.https-port	-1	
wiremock.server.https-port-dynamic	false	
wiremock.server.port	8080	
wiremock.server.port-dynamic	false	
wiremock.server.stubs	[]	

8.A.2. Additional application properties



The following properties can be passed as a system property (for example, `stubrunner.properties.git.branch`), as an environment variable (for example, `STUBRUNNER_PROPERTIES_GIT_BRANCH`), or as a property inside stub runner's annotation or a JUnit Rule (JUnit 4) or Extension (JUnit 5). In the latter case, you can pass the `git.branch` property instead of `stubrunner.properties.git.branch`.

Table 2. Stubrunner Properties Options

Name	Default	Description
<code>stubrunner.properties.git.branch</code>		When using the SCM-based approach, you can customize the branch name to check out.
<code>stubrunner.properties.git.commit-message</code>	Updating project [\$project] with stubs	When using the SCM based approach, you can customize the commit message for created stubs. The <code>\$project</code> text is replaced with the project name.
<code>stubrunner.properties.git.no-of-attempts</code>	10	When using the-SCM based approach, you can customize the number of retries to push the stubs to Git.
<code>stubrunner.properties.git.username</code>		When using the SCM-based approach, you can pass the username to connect to the git repository.
<code>stubrunner.properties.git.password</code>		When using the SCM-based approach, you can pass the password to connect to the git repository.
<code>stubrunner.properties.git.wait-between-attempts</code>	1000	When using the SCM-based approach, you can customize waiting time in ms between attempts to push the stubs to git.
<code>stubrunner.properties.git.ensure-git-suffix</code>	true	When using the SCM based approach, you can prevent stubrunner from adding <code>.git</code> to the repository URL by setting this property to <code>false</code> . This adds compatibility with git repositories which do not support such URLs, for example Azure DevOps.
<code>stubrunner.properties.stubs.find-producer</code>	false	When using the <code>stubs</code> protocol, you can toggle this flag to search for contracts in the <code>group id / artifact id</code> instead of taking the stubs directly from the provided folder.

Spring Cloud Contract Reference Documentation

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

Legal

4.1.0-M1

Copyright © 2012-2020

Copies of this document may be made for your own use and for distribution to others, provided that you do not charge any fee for such copies and further provided that each copy contains this Copyright Notice, whether distributed in print or electronically.

1. Getting Started

If you are getting started with Spring Cloud Contract or Spring in general, start by reading this section. It answers the basic “what?”, “how?” and “why?” questions. It includes an introduction to Spring Cloud Contract, along with installation instructions. We then walk you through building your first Spring Cloud Contract application, discussing some core principles as we go.

1.1. Introducing Spring Cloud Contract

Spring Cloud Contract moves TDD to the level of software architecture. It lets you perform consumer-driven and producer-driven contract testing.

1.1.1. History

Before becoming Spring Cloud Contract, this project was called [Accurest](#). It was created by [Marcin Grzejszczak](#) and [Jakub Kubrynski](#) from ([Codearte](#)).

The **0.1.0** release took place on 26 Jan 2015, and it became stable with **1.0.0** release on 29 Feb 2016.

Why Do You Need It?

Assume that we have a system that consists of multiple microservices, as the following image shows:

[Microservices Architecture] | *Dep.s.png*

Testing Issues

If we want to test the application in the top left corner of the image in the preceding section to

determine whether it can communicate with other services, we could do one of two things:

- Deploy all microservices and perform end-to-end tests.
- Mock other microservices in unit and integration tests.

Both have their advantages but also a lot of disadvantages.

Deploy all microservices and perform end-to-end tests

Advantages:

- Simulates production.
- Tests real communication between services.

Disadvantages:

- To test one microservice, we have to deploy six microservices, a couple of databases, and other items.
- The environment where the tests run is locked for a single suite of tests (nobody else would be able to run the tests in the meantime).
- They take a long time to run.
- The feedback comes very late in the process.
- They are extremely hard to debug.

Mock other microservices in unit and integration tests

Advantages:

- They provide very fast feedback.
- They have no infrastructure requirements.

Disadvantages:

- The implementor of the service creates stubs that might have nothing to do with reality.
- You can go to production with passing tests and failing production.

To solve the aforementioned issues, Spring Cloud Contract was created. The main idea is to give you very fast feedback, without the need to set up the whole world of microservices. If you work on stubs, then the only applications you need are those that your application directly uses. The following image shows the relationship of stubs to an application:

[Stubbed Services] | *Stubs2.png*

Spring Cloud Contract gives you the certainty that the stubs that you use were created by the service that you call. Also, if you can use them, it means that they were tested against the producer's side. In short, you can trust those stubs.

1.1.2. Purposes

The main purposes of Spring Cloud Contract are:

- To ensure that HTTP and messaging stubs (used when developing the client) do exactly what the actual server-side implementation does.
- To promote the ATDD (acceptance test-driven development) method, and the microservices architectural style.
- To provide a way to publish changes in contracts that are immediately visible on both sides.
- To generate boilerplate test code to be used on the server side.

By default, Spring Cloud Contract integrates with [Wiremock](#) as the HTTP server stub.



Spring Cloud Contract's purpose is NOT to start writing business features in the contracts. Assume that we have a business use case of fraud check. If a user can be a fraud for 100 different reasons, we would assume that you would create two contracts, one for the positive case and one for the negative case. Contract tests are used to test contracts between applications, not to simulate full behavior.

1.1.3. What Is a Contract?

As consumers of services, we need to define what exactly we want to achieve. We need to formulate our expectations. That is why we write contracts. In other words, a contract is an agreement on how the API or message communication should look. Consider the following example:

Assume that you want to send a request that contains the ID of a client company and the amount it wants to borrow from us. You also want to send it to the `/fraudcheck` URL by using the `PUT` method. The following listing shows a contract to check whether a client should be marked as a fraud in both Groovy and YAML:

groovy

```
/*
 * Copyright 2013-2020 the original author or authors.
 *
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 *     https://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */
```

```
package contracts
```

```
org.springframework.cloud.contract.spec.Contract.make {
    request { // (1)
        method 'PUT' // (2)
        url '/fraudcheck' // (3)
        body([ // (4)
            "client.id": $(regex('[0-9]{10}')),
            loanAmount : 99999
        ])
        headers { // (5)
            contentType('application/json')
        }
    }
    response { // (6)
        status OK() // (7)
        body([ // (8)
            fraudCheckStatus : "FRAUD",
            "rejection.reason": "Amount too high"
        ])
        headers { // (9)
            contentType('application/json')
        }
    }
}
```

```
/*
```

From the Consumer perspective, when shooting a request in the integration test:

- (1) - If the consumer sends a request
- (2) - With the "PUT" method
- (3) - to the URL "/fraudcheck"

- (4) - with the JSON body that
 - * has a field `client.id` that matches a regular expression `[0-9]{10}`
 - * has a field `loanAmount` that is equal to `99999`
- (5) - with header `Content-Type` equal to `application/json`
- (6) - then the response will be sent with
- (7) - status equal `200`
- (8) - and JSON body equal to

```
{ "fraudCheckStatus": "FRAUD", "rejectionReason": "Amount too high" }
```
- (9) - with header `Content-Type` equal to `application/json`

From the Producer perspective, in the autogenerated producer-side test:

- (1) - A request will be sent to the producer
 - (2) - With the "PUT" method
 - (3) - to the URL "/fraudcheck"
 - (4) - with the JSON body that
 - * has a field `client.id` that will have a generated value that matches a regular expression `[0-9]{10}`
 - * has a field `loanAmount` that is equal to `99999`
 - (5) - with header `Content-Type` equal to `application/json`
 - (6) - then the test will assert if the response has been sent with
 - (7) - status equal `200`
 - (8) - and JSON body equal to

```
{ "fraudCheckStatus": "FRAUD", "rejectionReason": "Amount too high" }
```
 - (9) - with header `Content-Type` matching `application/json.*`
- */

yaml

```
request: # (1)
  method: PUT # (2)
  url: /yamlfraudcheck # (3)
  body: # (4)
    "client.id": 1234567890
    loanAmount: 99999
  headers: # (5)
    Content-Type: application/json
  matchers:
    body:
      - path: $.['client.id'] # (6)
        type: by_regex
        value: "[0-9]{10}"
response: # (7)
  status: 200 # (8)
  body: # (9)
    fraudCheckStatus: "FRAUD"
    "rejection.reason": "Amount too high"
  headers: # (10)
    Content-Type: application/json
```

```

#From the Consumer perspective, when shooting a request in the integration test:
#
#(1) - If the consumer sends a request
#(2) - With the "PUT" method
#(3) - to the URL "/yamlfraudcheck"
#(4) - with the JSON body that
# * has a field `client.id`
# * has a field `loanAmount` that is equal to `99999`
#(5) - with header `Content-Type` equal to `application/json`
#(6) - and a `client.id` json entry matches the regular expression `[0-9]{10}`
#(7) - then the response will be sent with
#(8) - status equal `200`
#(9) - and JSON body equal to
# { "fraudCheckStatus": "FRAUD", "rejectionReason": "Amount too high" }
#(10) - with header `Content-Type` equal to `application/json`
#
#From the Producer perspective, in the autogenerated producer-side test:
#
#(1) - A request will be sent to the producer
#(2) - With the "PUT" method
#(3) - to the URL "/yamlfraudcheck"
#(4) - with the JSON body that
# * has a field `client.id` `1234567890`
# * has a field `loanAmount` that is equal to `99999`
#(5) - with header `Content-Type` equal to `application/json`
#(7) - then the test will assert if the response has been sent with
#(8) - status equal `200`
#(9) - and JSON body equal to
# { "fraudCheckStatus": "FRAUD", "rejectionReason": "Amount too high" }
#(10) - with header `Content-Type` equal to `application/json`

```



It is expected that contracts are coming from a **trusted source**. You should never download nor interact with contracts coming from untrusted locations.

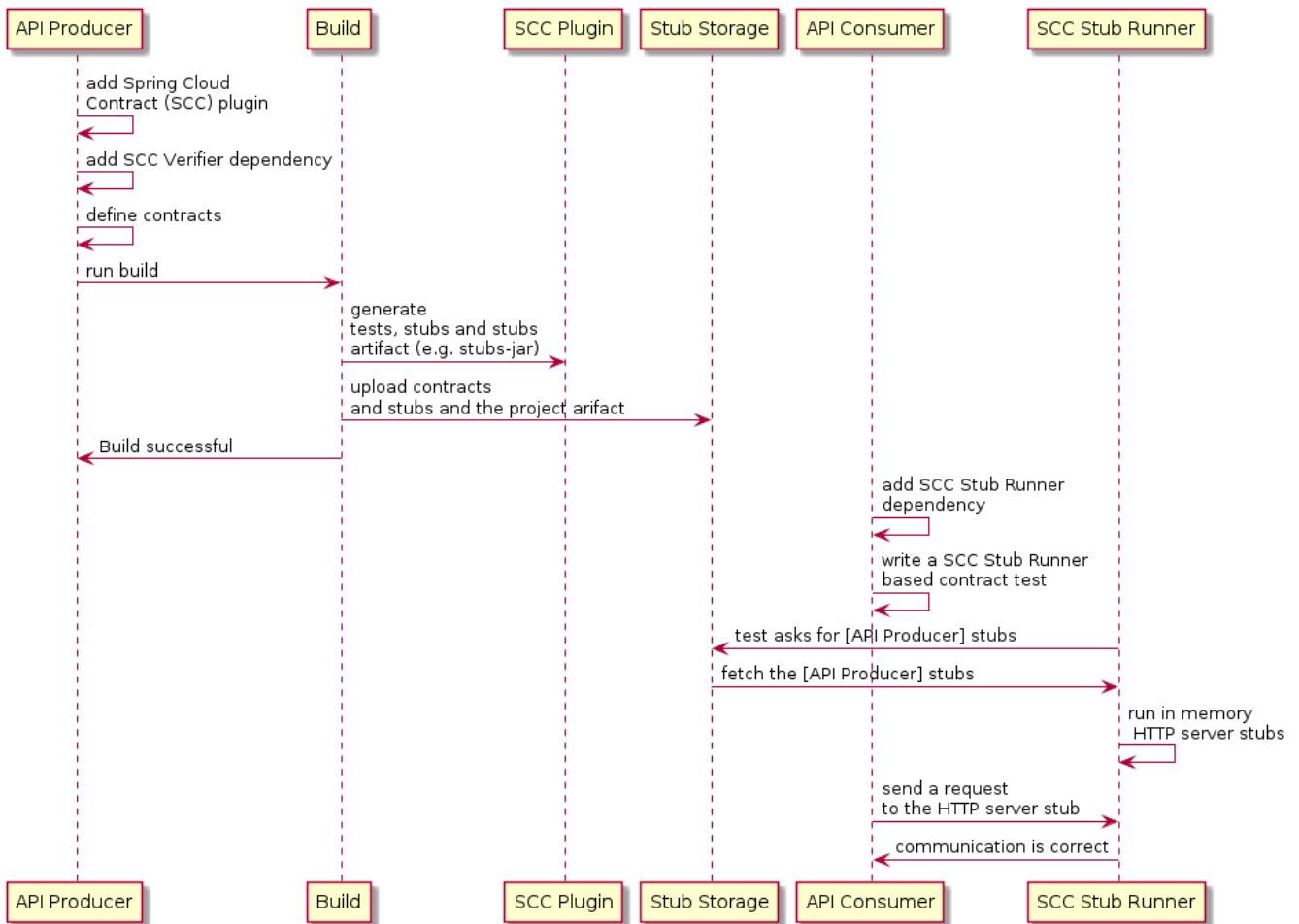
1.2. A Three-second Tour

This very brief tour walks through using Spring Cloud Contract. It consists of the following topics:

- [On the Producer Side](#)
- [On the Consumer Side](#)

You can find a somewhat longer tour [here](#).

The following UML diagram shows the relationship of the parts within Spring Cloud Contract:



1.2.1. On the Producer Side

To start working with Spring Cloud Contract, you can add files with REST or messaging contracts expressed in either Groovy DSL or YAML to the contracts directory, which is set by the `contractsDslDir` property. By default, it is `$rootDir/src/test/resources/contracts`.

Then you can add the Spring Cloud Contract Verifier dependency and plugin to your build file, as the following example shows:

```

<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-contract-verifier</artifactId>
  <scope>test</scope>
</dependency>
  
```

The following listing shows how to add the plugin, which should go in the build/plugins portion of the file:

```
<plugin>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-contract-maven-plugin</artifactId>
  <version>${spring-cloud-contract.version}</version>
  <extensions>>true</extensions>
</plugin>
```

Running `./mvnw clean install` automatically generates tests that verify the application compliance with the added contracts. By default, the tests get generated under `org.springframework.cloud.contract.verifier.tests..`

As the implementation of the functionalities described by the contracts is not yet present, the tests fail.

To make them pass, you must add the correct implementation of either handling HTTP requests or messages. Also, you must add a base test class for auto-generated tests to the project. This class is extended by all the auto-generated tests, and it should contain all the setup information necessary to run them (for example `RestAssuredMockMvc` controller setup or messaging test setup).

The following example, from `pom.xml`, shows how to specify the base test class:

```
<build>
  <plugins>
    <plugin>
      <groupId>org.springframework.cloud</groupId>
      <artifactId>spring-cloud-contract-maven-plugin</artifactId>
      <version>2.1.2.RELEASE</version>
      <extensions>>true</extensions>
      <configuration>
        <baseClassForTests>com.example.contractTest.BaseTestClass</baseClassForTests> ①
      </configuration>
    </plugin>
    <plugin>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-maven-plugin</artifactId>
    </plugin>
  </plugins>
</build>
```

① The `baseClassForTests` element lets you specify your base test class. It must be a child of a `configuration` element within `spring-cloud-contract-maven-plugin`.

Once the implementation and the test base class are in place, the tests pass, and both the application and the stub artifacts are built and installed in the local Maven repository. You can now

merge the changes, and you can publish both the application and the stub artifacts in an online repository.

1.2.2. On the Consumer Side

You can use **Spring Cloud Contract Stub Runner** in the integration tests to get a running WireMock instance or messaging route that simulates the actual service.

To do so, add the dependency to **Spring Cloud Contract Stub Runner**, as the following example shows:

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-contract-stub-runner</artifactId>
  <scope>test</scope>
</dependency>
```

You can get the Producer-side stubs installed in your Maven repository in either of two ways:

- By checking out the Producer side repository and adding contracts and generating the stubs by running the following commands:

```
$ cd local-http-server-repo
$ ./mvnw clean install -DskipTests
```



The tests are being skipped because the producer-side contract implementation is not in place yet, so the automatically-generated contract tests fail.

- By getting already-existing producer service stubs from a remote repository. To do so, pass the stub artifact IDs and artifact repository URL as **Spring Cloud Contract Stub Runner** properties, as the following example shows:

```
stubrunner:
  ids: 'com.example:http-server-dsl+:stubs:8080'
  repositoryRoot: https://repo.spring.io/libs-snapshot
```

Now you can annotate your test class with **@AutoConfigureStubRunner**. In the annotation, provide the **group-id** and **artifact-id** values for **Spring Cloud Contract Stub Runner** to run the collaborators' stubs for you, as the following example shows:


```
@RunWith(SpringRunner.class)
@SpringBootTest(webEnvironment=WebEnvironment.NONE)
@AutoConfigureStubRunner(ids = {"com.example:http-server-dsl:+:stubs:6565"},
    stubsMode = StubRunnerProperties.StubsMode.LOCAL)
public class LoanApplicationServiceTests {
    . . .
}
```



Use the **REMOTE** `stubsMode` when downloading stubs from an online repository and **LOCAL** for offline work.

Now, in your integration test, you can receive stubbed versions of HTTP responses or messages that are expected to be emitted by the collaborator service.

1.3. Developing Your First Spring Cloud Contract-based Application

This brief tour walks through using Spring Cloud Contract. It consists of the following topics:

- [On the Producer Side](#)
- [On the Consumer Side](#)

You can find an even more brief tour [here](#).

For the sake of this example, the **Stub Storage** is Nexus/Artifactory.

The following UML diagram shows the relationship of the parts of Spring Cloud Contract:

[Getting started first application] | *getting-started-three-second.png*

1.3.1. On the Producer Side

To start working with **Spring Cloud Contract**, you can add the Spring Cloud Contract Verifier dependency and plugin to your build file, as the following example shows:

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-contract-verifier</artifactId>
  <scope>test</scope>
</dependency>
```

The following listing shows how to add the plugin, which should go in the `build/plugins` portion of the file:

```
<plugin>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-contract-maven-plugin</artifactId>
  <version>${spring-cloud-contract.version}</version>
  <extensions>>true</extensions>
</plugin>
```

The easiest way to get started is to go to [the Spring Initializr](#) and add “Web” and “Contract Verifier” as dependencies. Doing so pulls in the previously mentioned dependencies and everything else you need in the `pom.xml` file (except for setting the base test class, which we cover later in this section). The following image shows the settings to use in [the Spring Initializr](#):

[Spring Initializr with Web and Contract Verifier] |



start_spring_io_dependencies.png

Now you can add files with **REST/** messaging contracts expressed in either Groovy DSL or YAML to the `contracts` directory, which is set by the `contractsDsldir` property. By default, it is `$rootDir/src/test/resources/contracts`. Note that the file name does not matter. You can organize your contracts within this directory with whatever naming scheme you like.

For the HTTP stubs, a contract defines what kind of response should be returned for a given request (taking into account the HTTP methods, URLs, headers, status codes, and so on). The following example shows an HTTP stub contract in both Groovy and YAML:

groovy

```
package contracts

org.springframework.cloud.contract.spec.Contract.make {
    request {
        method 'PUT'
        url '/fraudcheck'
        body([
            "client.id": $(regex('[0-9]{10}')),
            loanAmount: 99999
        ])
        headers {
            contentType('application/json')
        }
    }
    response {
        status OK()
        body([
            fraudCheckStatus: "FRAUD",
            "rejection.reason": "Amount too high"
        ])
        headers {
            contentType('application/json')
        }
    }
}
```

yaml

```
request:
  method: PUT
  url: /fraudcheck
  body:
    "client.id": 1234567890
    loanAmount: 99999
  headers:
    Content-Type: application/json
  matchers:
    body:
      - path: $.['client.id']
        type: by_regex
        value: "[0-9]{10}"
response:
  status: 200
  body:
    fraudCheckStatus: "FRAUD"
    "rejection.reason": "Amount too high"
  headers:
    Content-Type: application/json;charset=UTF-8
```

If you need to use messaging, you can define:

- The input and output messages (taking into account from where it was sent, the message body, and the header).
- The methods that should be called after the message is received.
- The methods that, when called, should trigger a message.

The following example shows a Camel messaging contract:

groovy

```
def contractDsl = Contract.make {
    name "foo"
    label 'some_label'
    input {
        triggeredBy('bookReturnedTriggered()')
    }
    outputMessage {
        sentTo('activemq:output')
        body(''{ "bookName" : "foo" }''')
        headers {
            header('BOOK-NAME', 'foo')
            messagingContentType(applicationJson())
        }
    }
}
```

yaml

```
label: some_label
input:
  triggeredBy: bookReturnedTriggered
outputMessage:
  sentTo: activemq:output
  body:
    bookName: foo
  headers:
    BOOK-NAME: foo
  contentType: application/json
```

Running `./mvnw clean install` automatically generates tests that verify the application compliance with the added contracts. By default, the generated tests are under `org.springframework.cloud.contract.verifier.tests..`

The generated tests may differ, depending on which framework and test type you have set up in your plugin.

In the next listing, you can find:

- The default test mode for HTTP contracts in `MockMvc`
- A JAX-RS client with the `JAXRS` test mode
- A `WebTestClient`-based test (this is particularly recommended while working with Reactive, `Web-Flux`-based applications) set with the `WEBTESTCLIENT` test mode



You need only one of these test frameworks. `MockMvc` is the default. To use one of the other frameworks, add its library to your classpath.

The following listing shows samples for all frameworks:

mockmvc

```
@Test
public void validate_shouldMarkClientAsFraud() throws Exception {
    // given:
    MockMvcRequestSpecification request = given()
        .header("Content-Type", "application/vnd.fraud.v1+json")
        .body("{\"client.id\":\"1234567890\",\"loanAmount\":99999}");

    // when:
    ResponseOptions response = given().spec(request)
        .put("/fraudcheck");

    // then:
    assertThat(response.statusCode()).isEqualTo(200);
    assertThat(response.header("Content-
Type")).matches("application/vnd.fraud.v1.json.*");
    // and:
    DocumentContext parsedJson =
    JsonPath.parse(response.getBody().asString());
    assertThatJson(parsedJson).field("'fraudCheckStatus']").matches("[A-
Z]{5}");
    assertThatJson(parsedJson).field("'rejection.reason']").isEqualTo("Amount
too high");
}
```



```
@SuppressWarnings("rawtypes")
public class FooTest {
    WebTarget webTarget;

    @Test
    public void validate_() throws Exception {

        // when:
        Response response = webTarget
            .path("/users")
            .queryParams("limit", "10")
            .queryParams("offset", "20")
            .queryParams("filter", "email")
            .queryParams("sort", "name")
            .queryParams("search", "55")
            .queryParams("age", "99")
            .queryParams("name", "Denis.Stepanov")
            .queryParams("email", "bob@email.com")
            .request()
            .build("GET")
            .invoke();
        String responseAsString = response.readEntity(String.class);

        // then:
        assertThat(response.getStatus()).isEqualTo(200);

        // and:
        DocumentContext parsedJson = JsonPath.parse(responseAsString);
        assertThatJson(parsedJson).field("[ 'property1'']").isEqualTo("a");
    }
}
```

webtestclient

```
@Test
public void validate_shouldRejectABeerIfTooYoung() throws Exception {
    // given:
    WebTestClientRequestSpecification request = given()
        .header("Content-Type", "application/json")
        .body("{\"age\":10}");

    // when:
    WebTestClientResponse response = given().spec(request)
        .post("/check");

    // then:
    assertThat(response.statusCode()).isEqualTo(200);
    assertThat(response.header("Content-
Type")).matches("application/json.*");
    // and:
    DocumentContext parsedJson =
    JsonPath.parse(response.getBody().asString());
    assertThatJson(parsedJson).field("[ 'status' ]").isEqualTo("NOT_OK");
}
```

As the implementation of the functionalities described by the contracts is not yet present, the tests fail.

To make them pass, you must add the correct implementation of handling either HTTP requests or messages. Also, you must add a base test class for auto-generated tests to the project. This class is extended by all the auto-generated tests and should contain all the setup necessary information needed to run them (for example, [RestAssuredMockMvc](#) controller setup or messaging test setup).

The following example, from [pom.xml](#), shows how to specify the base test class:

```

<build>
  <plugins>
    <plugin>
      <groupId>org.springframework.cloud</groupId>
      <artifactId>spring-cloud-contract-maven-plugin</artifactId>
      <version>2.1.2.RELEASE</version>
      <extensions>true</extensions>
      <configuration>

<baseClassForTests>com.example.contractTest.BaseTestClass</baseClassForTests> ①
      </configuration>
    </plugin>
    <plugin>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-maven-plugin</artifactId>
    </plugin>
  </plugins>
</build>

```

① The `baseClassForTests` element lets you specify your base test class. It must be a child of a `configuration` element within `spring-cloud-contract-maven-plugin`.

The following example shows a minimal (but functional) base test class:

```

package com.example.contractTest;

import org.junit.Before;

import io.restassured.module.mockmvc.RestAssuredMockMvc;

public class BaseTestClass {

    @Before
    public void setup() {
        RestAssuredMockMvc.standaloneSetup(new FraudController());
    }
}

```

This minimal class really is all you need to get your tests to work. It serves as a starting place to which the automatically generated tests attach.

Now we can move on to the implementation. For that, we first need a data class, which we then use in our controller. The following listing shows the data class:

```
package com.example.Test;

import com.fasterxml.jackson.annotation.JsonProperty;

public class LoanRequest {

    @JsonProperty("client.id")
    private String clientId;

    private Long loanAmount;

    public String getClientId() {
        return clientId;
    }

    public void setClientId(String clientId) {
        this.clientId = clientId;
    }

    public Long getLoanAmount() {
        return loanAmount;
    }

    public void setLoanRequestAmount(Long loanAmount) {
        this.loanAmount = loanAmount;
    }
}
```

The preceding class provides an object in which we can store the parameters. Because the client ID in the contract is called `client.id`, we need to use the `@JsonProperty("client.id")` parameter to map it to the `clientId` field.

Now we can move along to the controller, which the following listing shows:

```

package com.example.docTest;

import org.springframework.web.bind.annotation.PutMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class FraudController {

    @PutMapping(value = "/fraudcheck", consumes="application/json",
produces="application/json")
    public String check(@RequestBody LoanRequest loanRequest) { ①

        if (loanRequest.getLoanAmount() > 10000) { ②
            return "{fraudCheckStatus: FRAUD, rejection.reason: Amount too high}";
        } else {
            return "{fraudCheckStatus: OK, acceptance.reason: Amount OK}"; ④
        }
    }
}

```

- ① We map the incoming parameters to a `LoanRequest` object.
- ② We check the requested loan amount to see if it is too much.
- ③ If it is too much, we return the JSON (created with a simple string here) that the test expects.
- ④ If we had a test to catch when the amount is allowable, we could match it to this output.

The `FraudController` is about as simple as things get. You can do much more, including logging, validating the client ID, and so on.

Once the implementation and the test base class are in place, the tests pass, and both the application and the stub artifacts are built and installed in the local Maven repository. Information about installing the stubs jar to the local repository appears in the logs, as the following example shows:

```
[INFO] --- spring-cloud-contract-maven-plugin:1.0.0.BUILD-SNAPSHOT:generateStubs
(default-generateStubs) @ http-server ---
[INFO] Building jar: /some/path/http-server/target/http-server-0.0.1-SNAPSHOT-
stubs.jar
[INFO]
[INFO] --- maven-jar-plugin:2.6:jar (default-jar) @ http-server ---
[INFO] Building jar: /some/path/http-server/target/http-server-0.0.1-SNAPSHOT.jar
[INFO]
[INFO] --- spring-boot-maven-plugin:1.5.5.BUILD-SNAPSHOT:repackage (default) @
http-server ---
[INFO]
[INFO] --- maven-install-plugin:2.5.2:install (default-install) @ http-server ---
[INFO] Installing /some/path/http-server/target/http-server-0.0.1-SNAPSHOT.jar to
/path/to/your/.m2/repository/com/example/http-server/0.0.1-SNAPSHOT/http-server-
0.0.1-SNAPSHOT.jar
[INFO] Installing /some/path/http-server/pom.xml to
/path/to/your/.m2/repository/com/example/http-server/0.0.1-SNAPSHOT/http-server-
0.0.1-SNAPSHOT.pom
[INFO] Installing /some/path/http-server/target/http-server-0.0.1-SNAPSHOT-
stubs.jar to /path/to/your/.m2/repository/com/example/http-server/0.0.1-
SNAPSHOT/http-server-0.0.1-SNAPSHOT-stubs.jar
```

You can now merge the changes and publish both the application and the stub artifacts in an online repository.

1.3.2. On the Consumer Side

You can use Spring Cloud Contract Stub Runner in the integration tests to get a running WireMock instance or messaging route that simulates the actual service.

To get started, add the dependency to [Spring Cloud Contract Stub Runner](#), as follows:

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-contract-stub-runner</artifactId>
  <scope>test</scope>
</dependency>
```

You can get the Producer-side stubs installed in your Maven repository in either of two ways:

- By checking out the Producer side repository and adding contracts and generating the stubs by running the following commands:

```
$ cd local-http-server-repo
$ ./mvnw clean install -DskipTests
```



The tests are skipped because the Producer-side contract implementation is not yet in place, so the automatically-generated contract tests fail.

- By getting existing producer service stubs from a remote repository. To do so, pass the stub artifact IDs and artifact repository URL as **Spring Cloud Contract Stub Runner** properties, as the following example shows:

```
stubrunner:
  ids: 'com.example:http-server-dsl:+:stubs:8080'
  repositoryRoot: https://repo.spring.io/libs-snapshot
```

Now you can annotate your test class with **@AutoConfigureStubRunner**. In the annotation, provide the **group-id** and **artifact-id** for **Spring Cloud Contract Stub Runner** to run the collaborators' stubs for you, as the following example shows:

```
@RunWith(SpringRunner.class)
@SpringBootTest(webEnvironment=WebEnvironment.NONE)
@AutoConfigureStubRunner(ids = {"com.example:http-server-dsl:+:stubs:6565"},
    stubsMode = StubRunnerProperties.StubsMode.LOCAL)
public class LoanApplicationServiceTests {
    . . .
}
```



Use the **REMOTE stubsMode** when downloading stubs from an online repository and **LOCAL** for offline work.

In your integration test, you can receive stubbed versions of HTTP responses or messages that are expected to be emitted by the collaborator service. You can see entries similar to the following in the build logs:

```

2016-07-19 14:22:25.403 INFO 41050 --- [           main]
o.s.c.c.stubrunner.AetherStubDownloader : Desired version is + - will try to
resolve the latest version
2016-07-19 14:22:25.438 INFO 41050 --- [           main]
o.s.c.c.stubrunner.AetherStubDownloader : Resolved version is 0.0.1-SNAPSHOT
2016-07-19 14:22:25.439 INFO 41050 --- [           main]
o.s.c.c.stubrunner.AetherStubDownloader : Resolving artifact com.example:http-
server:jar:stubs:0.0.1-SNAPSHOT using remote repositories []
2016-07-19 14:22:25.451 INFO 41050 --- [           main]
o.s.c.c.stubrunner.AetherStubDownloader : Resolved artifact com.example:http-
server:jar:stubs:0.0.1-SNAPSHOT to /path/to/your/.m2/repository/com/example/http-
server/0.0.1-SNAPSHOT/http-server-0.0.1-SNAPSHOT-stubs.jar
2016-07-19 14:22:25.465 INFO 41050 --- [           main]
o.s.c.c.stubrunner.AetherStubDownloader : Unpacking stub from JAR [URI:
file:/path/to/your/.m2/repository/com/example/http-server/0.0.1-SNAPSHOT/http-
server-0.0.1-SNAPSHOT-stubs.jar]
2016-07-19 14:22:25.475 INFO 41050 --- [           main]
o.s.c.c.stubrunner.AetherStubDownloader : Unpacked file to
[/var/folders/0p/xwq47sq106x1_g3dtv6qfm940000gq/T/contracts100276532569594265]
2016-07-19 14:22:27.737 INFO 41050 --- [           main]
o.s.c.c.stubrunner.StubRunnerExecutor   : All stubs are now running RunningStubs
[NamesAndPorts={com.example:http-server:0.0.1-SNAPSHOT:stubs=8080}]

```

1.4. Step-by-step Guide to Consumer Driven Contracts (CDC) with Contracts on the Producer Side

Consider an example of fraud detection and the loan issuance process. The business scenario is such that we want to issue loans to people but do not want them to steal from us. The current implementation of our system grants loans to everybody.

Assume that **Loan Issuance** is a client to the **Fraud Detection** server. In the current sprint, we must develop a new feature: if a client wants to borrow too much money, we mark the client as a fraud.

Technical remarks

- Fraud Detection has an **artifact-id** of **http-server**.
- Loan Issuance has an **artifact-id** of **http-client**.
- Both have a **group-id** of **com.example**.
- For the sake of this example, the **Stub Storage** is Nexus/Artifactory.

Social remarks

- Both the client and the server development teams need to communicate directly and discuss changes while going through the process.
- CDC is all about communication.

The [server-side code is available here](#), and [the client code is available here](#).



In this case, the producer owns the contracts. Physically, all the contracts are in the producer's repository.

1.4.1. Technical Note

If you use the SNAPSHOT, Milestone, or Release Candidate versions, you need to add the following section to your build:

Maven

```
<repositories>
  <repository>
    <id>spring-snapshots</id>
    <name>Spring Snapshots</name>
    <url>https://repo.spring.io/snapshot</url>
    <snapshots>
      <enabled>true</enabled>
    </snapshots>
  </repository>
  <repository>
    <id>spring-milestones</id>
    <name>Spring Milestones</name>
    <url>https://repo.spring.io/milestone</url>
    <snapshots>
      <enabled>>false</enabled>
    </snapshots>
  </repository>
  <repository>
    <id>spring-releases</id>
    <name>Spring Releases</name>
    <url>https://repo.spring.io/release</url>
    <snapshots>
      <enabled>>false</enabled>
    </snapshots>
  </repository>
</repositories>
<pluginRepositories>
  <pluginRepository>
    <id>spring-snapshots</id>
    <name>Spring Snapshots</name>
    <url>https://repo.spring.io/snapshot</url>
    <snapshots>
      <enabled>true</enabled>
    </snapshots>
  </pluginRepository>
  <pluginRepository>
    <id>spring-milestones</id>
    <name>Spring Milestones</name>
    <url>https://repo.spring.io/milestone</url>
    <snapshots>
      <enabled>>false</enabled>
    </snapshots>
  </pluginRepository>
  <pluginRepository>
    <id>spring-releases</id>
    <name>Spring Releases</name>
    <url>https://repo.spring.io/release</url>
    <snapshots>
```

```
        <enabled>false</enabled>
    </snapshots>
</pluginRepository>
</pluginRepositories>
```

Gradle

```
repositories {
    mavenCentral()
    mavenLocal()
    maven { url "https://repo.spring.io/snapshot" }
    maven { url "https://repo.spring.io/milestone" }
    maven { url "https://repo.spring.io/release" }
}
```

For simplicity, we use the following acronyms:

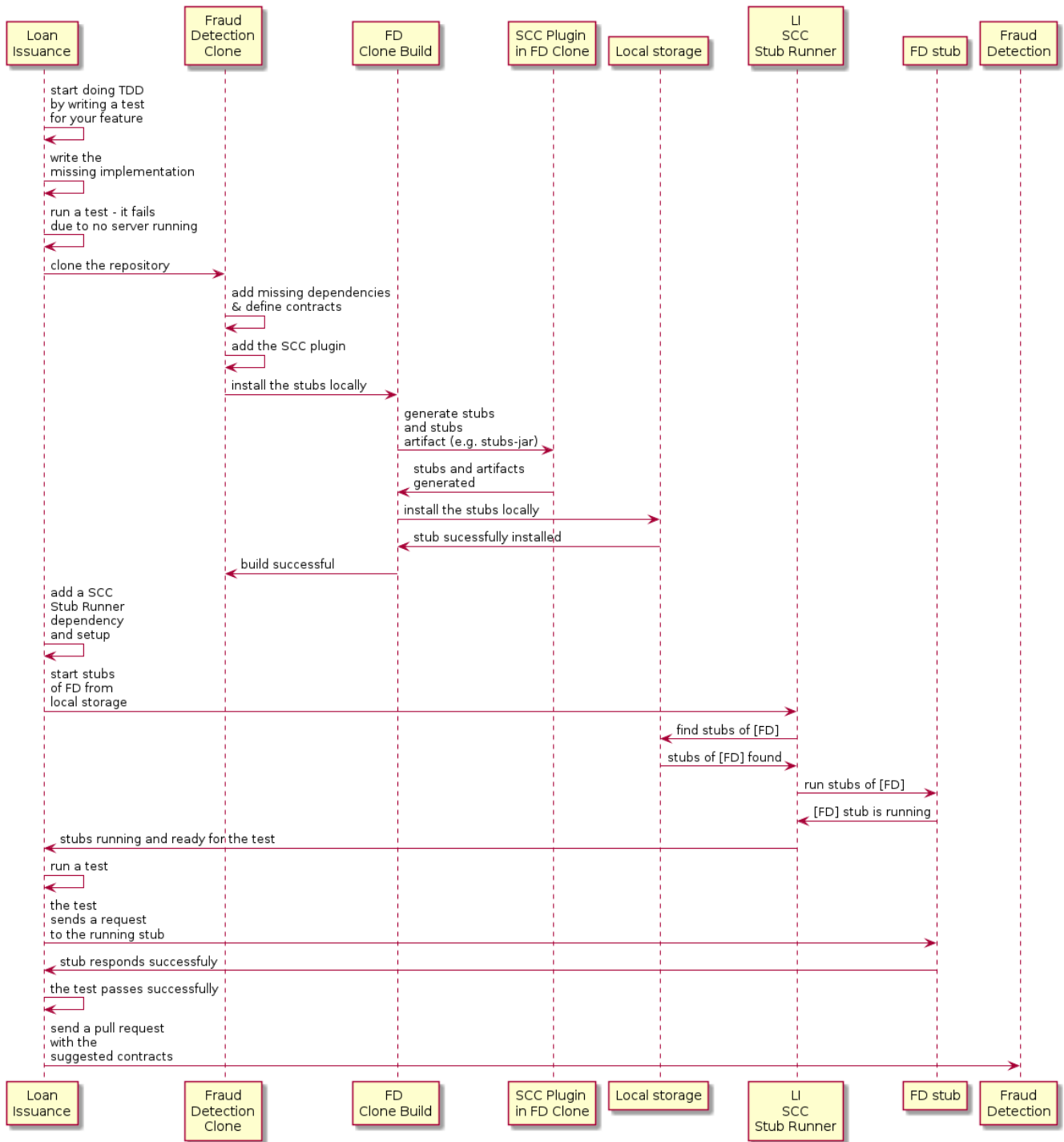
- Loan Issuance (LI): The HTTP client
- Fraud Detection (FD): The HTTP server
- SCC: Spring Cloud Contract

1.4.2. The Consumer Side (Loan Issuance)

As a developer of the Loan Issuance service (a consumer of the Fraud Detection server), you might do the following steps:

1. Start doing TDD by writing a test for your feature.
2. Write the missing implementation.
3. Clone the Fraud Detection service repository locally.
4. Define the contract locally in the repository of the fraud detection service.
5. Add the Spring Cloud Contract (SCC) plugin.
6. Run the integration tests.
7. File a pull request.
8. Create an initial implementation.
9. Take over the pull request.
10. Write the missing implementation.
11. Deploy your application.
12. Work online.

We start with the loan issuance flow, which the following UML diagram shows:



Start Doing TDD by Writing a Test for Your Feature

The following listing shows a test that we might use to check whether a loan amount is too large:

```

@Test
public void shouldBeRejectedDueToAbnormalLoanAmount() {
    // given:
    LoanApplication application = new LoanApplication(new Client("1234567890"),
        99999);
    // when:
    LoanApplicationResult loanApplication = service.loanApplication(application);
    // then:
    assertThat(loanApplication.getLoanApplicationStatus())
        .isEqualTo(LoanApplicationStatus.LOAN_APPLICATION_REJECTED);
    assertThat(loanApplication.getRejectionReason()).isEqualTo("Amount too high");
}

```

Assume that you have written a test of your new feature. If a loan application for a big amount is received, the system should reject that loan application with some description.

Write the Missing Implementation

At some point in time, you need to send a request to the Fraud Detection service. Assume that you need to send the request containing the ID of the client and the amount the client wants to borrow. You want to send it to the `/fraudcheck` URL by using the `PUT` method. To do so, you might use code similar to the following:

```

ResponseEntity<FraudServiceResponse> response = restTemplate.exchange(
    "http://localhost:" + port + fraudCheck(), HttpMethod.PUT,
    new HttpEntity<>(request, httpHeaders), FraudServiceResponse.class);

```

For simplicity, the port of the Fraud Detection service is set to `8080`, and the application runs on `8090`.



If you start the test at this point, it breaks, because no service currently runs on port `8080`.

Clone the Fraud Detection service repository locally

You can start by playing around with the server side contract. To do so, you must first clone it, by running the following command:

```

$ git clone https://your-git-server.com/server-side.git local-http-server-repo

```

Define the Contract Locally in the Repository of the Fraud Detection Service

As a consumer, you need to define what exactly you want to achieve. You need to formulate your expectations. To do so, write the following contract:



Place the contract in the `src/test/resources/contracts/fraud` folder. The `fraud` folder is important because the producer's test base class name references that folder.

The following example shows our contract, in both Groovy and YAML:

groovy

```
/*
 * Copyright 2013-2020 the original author or authors.
 *
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 *     https://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */
```

```
package contracts
```

```
org.springframework.cloud.contract.spec.Contract.make {
    request { // (1)
        method 'PUT' // (2)
        url '/fraudcheck' // (3)
        body([ // (4)
            "client.id": $(regex('[0-9]{10}')),
            loanAmount : 99999
        ])
        headers { // (5)
            contentType('application/json')
        }
    }
    response { // (6)
        status OK() // (7)
        body([ // (8)
            fraudCheckStatus : "FRAUD",
            "rejection.reason": "Amount too high"
        ])
        headers { // (9)
            contentType('application/json')
        }
    }
}
```

```
/*
```

From the Consumer perspective, when shooting a request in the integration test:

- (1) - If the consumer sends a request
- (2) - With the "PUT" method
- (3) - to the URL "/fraudcheck"

- (4) - with the JSON body that
 - * has a field `client.id` that matches a regular expression `[0-9]{10}`
 - * has a field `loanAmount` that is equal to `99999`
- (5) - with header `Content-Type` equal to `application/json`
- (6) - then the response will be sent with
- (7) - status equal `200`
- (8) - and JSON body equal to


```
{ "fraudCheckStatus": "FRAUD", "rejectionReason": "Amount too high" }
```
- (9) - with header `Content-Type` equal to `application/json`

From the Producer perspective, in the autogenerated producer-side test:

- (1) - A request will be sent to the producer
 - (2) - With the "PUT" method
 - (3) - to the URL "/fraudcheck"
 - (4) - with the JSON body that
 - * has a field `client.id` that will have a generated value that matches a regular expression `[0-9]{10}`
 - * has a field `loanAmount` that is equal to `99999`
 - (5) - with header `Content-Type` equal to `application/json`
 - (6) - then the test will assert if the response has been sent with
 - (7) - status equal `200`
 - (8) - and JSON body equal to


```
{ "fraudCheckStatus": "FRAUD", "rejectionReason": "Amount too high" }
```
 - (9) - with header `Content-Type` matching `application/json.*`
- */

yaml

```
request: # (1)
  method: PUT # (2)
  url: /yamlfraudcheck # (3)
  body: # (4)
    "client.id": 1234567890
    loanAmount: 99999
  headers: # (5)
    Content-Type: application/json
  matchers:
    body:
      - path: $.['client.id'] # (6)
        type: by_regex
        value: "[0-9]{10}"
response: # (7)
  status: 200 # (8)
  body: # (9)
    fraudCheckStatus: "FRAUD"
    "rejection.reason": "Amount too high"
  headers: # (10)
    Content-Type: application/json
```



```

#From the Consumer perspective, when shooting a request in the integration test:
#
#(1) - If the consumer sends a request
#(2) - With the "PUT" method
#(3) - to the URL "/yamlfraudcheck"
#(4) - with the JSON body that
# * has a field `client.id`
# * has a field `loanAmount` that is equal to `99999`
#(5) - with header `Content-Type` equal to `application/json`
#(6) - and a `client.id` json entry matches the regular expression `[0-9]{10}`
#(7) - then the response will be sent with
#(8) - status equal `200`
#(9) - and JSON body equal to
# { "fraudCheckStatus": "FRAUD", "rejectionReason": "Amount too high" }
#(10) - with header `Content-Type` equal to `application/json`
#
#From the Producer perspective, in the autogenerated producer-side test:
#
#(1) - A request will be sent to the producer
#(2) - With the "PUT" method
#(3) - to the URL "/yamlfraudcheck"
#(4) - with the JSON body that
# * has a field `client.id` `1234567890`
# * has a field `loanAmount` that is equal to `99999`
#(5) - with header `Content-Type` equal to `application/json`
#(7) - then the test will assert if the response has been sent with
#(8) - status equal `200`
#(9) - and JSON body equal to
# { "fraudCheckStatus": "FRAUD", "rejectionReason": "Amount too high" }
#(10) - with header `Content-Type` equal to `application/json`

```

The YML contract is quite straightforward. However, when you take a look at the contract written with a statically typed Groovy DSL, you might wonder what the `value(client(...), server(...))` parts are. By using this notation, Spring Cloud Contract lets you define parts of a JSON block, a URL, or other structure that is dynamic. In the case of an identifier or a timestamp, you need not hardcode a value. You want to allow some different ranges of values. To enable ranges of values, you can set regular expressions that match those values for the consumer side. You can provide the body by means of either a map notation or a String with interpolations. We highly recommend using the map notation.



To set up contracts, you must understand the map notation. See the [Groovy docs regarding JSON](#).

The previously shown contract is an agreement between two sides that:

- If an HTTP request is sent with all of:
 - A `PUT` method on the `/fraudcheck` endpoint
 - A JSON body with a `client.id` that matches the regular expression `[0-9]{10}` and `loanAmount`

equal to 99999

- A **Content-Type** header with a value of `application/vnd.fraud.v1+json`
- Then an HTTP response is sent to the consumer that
 - Has status `200`
 - Contains a JSON body with the `fraudCheckStatus` field containing a value of `FRAUD` and the `rejectionReason` field having a value of `Amount too high`
 - Has a **Content-Type** header with a value of `application/vnd.fraud.v1+json`

Once you are ready to check the API in practice in the integration tests, you need to install the stubs locally.

Add the Spring Cloud Contract Verifier Plugin

We can add either a Maven or a Gradle plugin. In this example, we show how to add Maven. First, we add the **Spring Cloud Contract** BOM, as the following example shows:

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

Next, add the **Spring Cloud Contract Verifier** Maven plugin, as the following example shows:

```
  <plugin>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-contract-maven-plugin</artifactId>
    <version>${spring-cloud-contract.version}</version>
    <extensions>true</extensions>
    <configuration>

<packageWithBaseClasses>com.example.fraud</packageWithBaseClasses>
<!--           <convertToYaml>true</convertToYaml>-->
    </configuration>
  </plugin>
```

Since the plugin was added, you get the **Spring Cloud Contract Verifier** features, which, from the provided contracts:

- Generate and run tests

- Produce and install stubs

You do not want to generate tests, since you, as the consumer, want only to play with the stubs. You need to skip the test generation and invocation. To do so, run the following commands:

```
$ cd local-http-server-repo
$ ./mvnw clean install -DskipTests
```

Once you run those commands, you should see something like the following content in the logs:

```
[INFO] --- spring-cloud-contract-maven-plugin:1.0.0.BUILD-SNAPSHOT:generateStubs
(default-generateStubs) @ http-server ---
[INFO] Building jar: /some/path/http-server/target/http-server-0.0.1-SNAPSHOT-
stubs.jar
[INFO]
[INFO] --- maven-jar-plugin:2.6:jar (default-jar) @ http-server ---
[INFO] Building jar: /some/path/http-server/target/http-server-0.0.1-SNAPSHOT.jar
[INFO]
[INFO] --- spring-boot-maven-plugin:1.5.5.BUILD-SNAPSHOT:repackage (default) @
http-server ---
[INFO]
[INFO] --- maven-install-plugin:2.5.2:install (default-install) @ http-server ---
[INFO] Installing /some/path/http-server/target/http-server-0.0.1-SNAPSHOT.jar to
/path/to/your/.m2/repository/com/example/http-server/0.0.1-SNAPSHOT/http-server-
0.0.1-SNAPSHOT.jar
[INFO] Installing /some/path/http-server/pom.xml to
/path/to/your/.m2/repository/com/example/http-server/0.0.1-SNAPSHOT/http-server-
0.0.1-SNAPSHOT.pom
[INFO] Installing /some/path/http-server/target/http-server-0.0.1-SNAPSHOT-
stubs.jar to /path/to/your/.m2/repository/com/example/http-server/0.0.1-
SNAPSHOT/http-server-0.0.1-SNAPSHOT-stubs.jar
```

The following line is extremely important:

```
[INFO] Installing /some/path/http-server/target/http-server-0.0.1-SNAPSHOT-
stubs.jar to /path/to/your/.m2/repository/com/example/http-server/0.0.1-
SNAPSHOT/http-server-0.0.1-SNAPSHOT-stubs.jar
```

It confirms that the stubs of the `http-server` have been installed in the local repository.

Running the Integration Tests

In order to profit from the Spring Cloud Contract Stub Runner functionality of automatic stub downloading, you must do the following in your consumer side project (**Loan Application service**):

1. Add the **Spring Cloud Contract** BOM, as follows:

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

2. Add the dependency to **Spring Cloud Contract Stub Runner**, as follows:

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-contract-stub-runner</artifactId>
  <scope>test</scope>
</dependency>
```

3. Annotate your test class with **@AutoConfigureStubRunner**. In the annotation, provide the **group-id** and **artifact-id** for the Stub Runner to download the stubs of your collaborators.

```
@SpringBootTest(webEnvironment = WebEnvironment.NONE)
@AutoConfigureStubRunner(ids = {
    "com.example:http-server-dsl:0.0.1:stubs"}, stubsMode =
StubRunnerProperties.StubsMode.LOCAL)
public class LoanApplicationServiceTests {
```

4. (Optional) Because you are playing with the collaborators offline, you can also provide the offline work switch (**StubRunnerProperties.StubsMode.LOCAL**).

Now, when you run your tests, you see something like the following output in the logs:

```

2016-07-19 14:22:25.403 INFO 41050 --- [          main]
o.s.c.c.stubrunner.AetherStubDownloader : Desired version is + - will try to
resolve the latest version
2016-07-19 14:22:25.438 INFO 41050 --- [          main]
o.s.c.c.stubrunner.AetherStubDownloader : Resolved version is 0.0.1-SNAPSHOT
2016-07-19 14:22:25.439 INFO 41050 --- [          main]
o.s.c.c.stubrunner.AetherStubDownloader : Resolving artifact com.example:http-
server:jar:stubs:0.0.1-SNAPSHOT using remote repositories []
2016-07-19 14:22:25.451 INFO 41050 --- [          main]
o.s.c.c.stubrunner.AetherStubDownloader : Resolved artifact com.example:http-
server:jar:stubs:0.0.1-SNAPSHOT to /path/to/your/.m2/repository/com/example/http-
server/0.0.1-SNAPSHOT/http-server-0.0.1-SNAPSHOT-stubs.jar
2016-07-19 14:22:25.465 INFO 41050 --- [          main]
o.s.c.c.stubrunner.AetherStubDownloader : Unpacking stub from JAR [URI:
file:/path/to/your/.m2/repository/com/example/http-server/0.0.1-SNAPSHOT/http-
server-0.0.1-SNAPSHOT-stubs.jar]
2016-07-19 14:22:25.475 INFO 41050 --- [          main]
o.s.c.c.stubrunner.AetherStubDownloader : Unpacked file to
[/var/folders/0p/xwq47sq106x1_g3dtv6qfm940000gq/T/contracts100276532569594265]
2016-07-19 14:22:27.737 INFO 41050 --- [          main]
o.s.c.c.stubrunner.StubRunnerExecutor   : All stubs are now running RunningStubs
[NamesAndPorts={com.example:http-server:0.0.1-SNAPSHOT:stubs=8080}]

```

This output means that Stub Runner has found your stubs and started a server for your application with a group ID of `com.example` and an artifact ID of `http-server` with version `0.0.1-SNAPSHOT` of the stubs and with the `stubs` classifier on port `8080`.

Filing a Pull Request

What you have done until now is an iterative process. You can play around with the contract, install it locally, and work on the consumer side until the contract works as you wish.

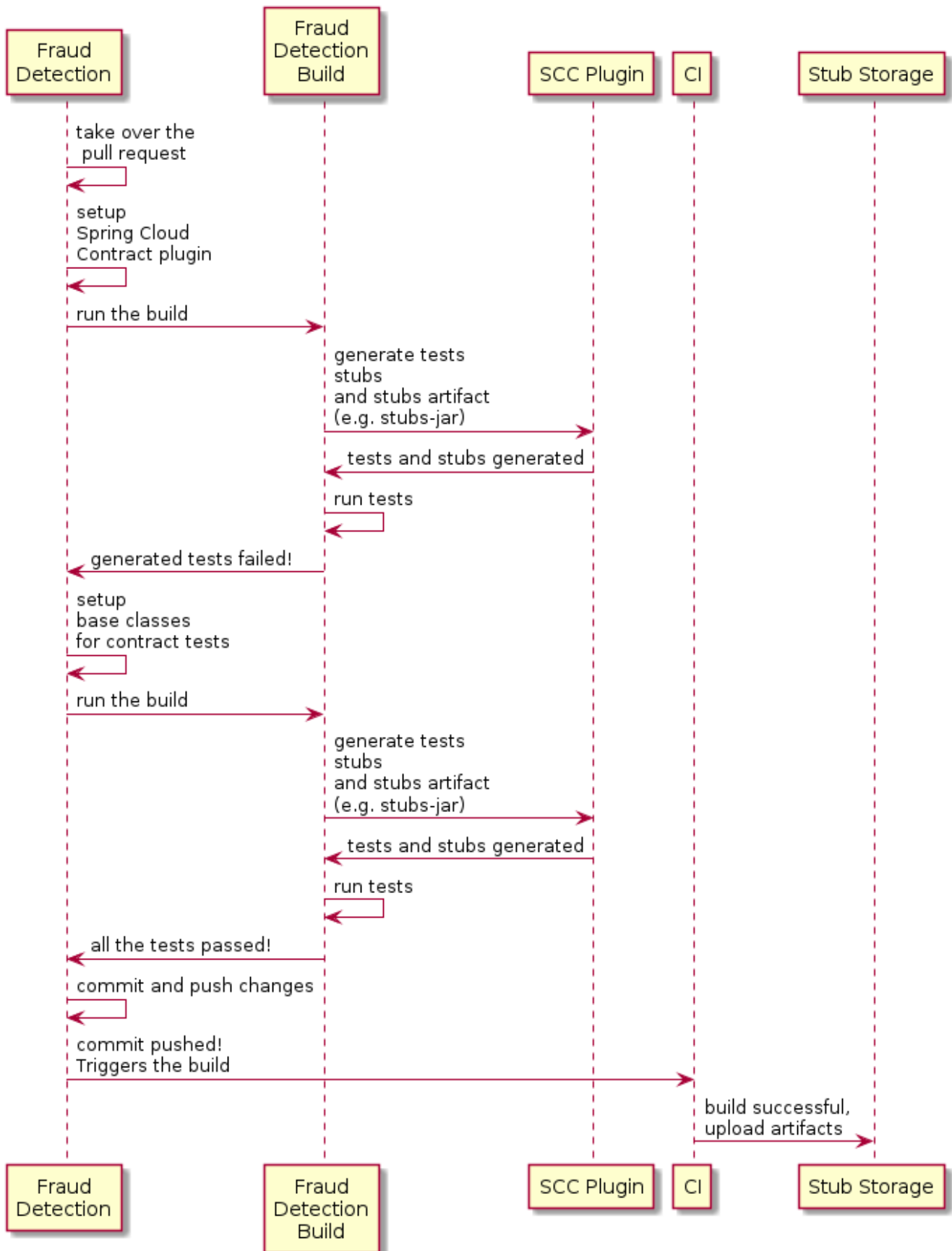
Once you are satisfied with the results and the test passes, you can publish a pull request to the server side. Currently, the consumer side work is done.

1.4.3. The Producer Side (Fraud Detection server)

As a developer of the Fraud Detection server (a server to the Loan Issuance service), you might want to:

- Take over the pull request
- Write the missing implementation
- Deploy the application

The following UML diagram shows the fraud detection flow:



Taking over the Pull Request

As a reminder, the following listing shows the initial implementation:

```
@RequestMapping(value = "/fraudcheck", method = PUT)
public FraudCheckResult fraudCheck(@RequestBody FraudCheck fraudCheck) {
    return new FraudCheckResult(FraudCheckStatus.OK, NO_REASON);
}
```

Then you can run the following commands:

```
$ git checkout -b contract-change-pr master
$ git pull https://your-git-server.com/server-side-fork.git contract-change-pr
```

You must add the dependencies needed by the autogenerated tests, as follows:

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-contract-verifier</artifactId>
  <scope>test</scope>
</dependency>
```

In the configuration of the Maven plugin, you must pass the `packageWithBaseClasses` property, as follows:

```
    <plugin>
      <groupId>org.springframework.cloud</groupId>
      <artifactId>spring-cloud-contract-maven-plugin</artifactId>
      <version>${spring-cloud-contract.version}</version>
      <extensions>true</extensions>
      <configuration>

<packageWithBaseClasses>com.example.fraud</packageWithBaseClasses>
<!--           <convertToYaml>true</convertToYaml>-->
      </configuration>
    </plugin>
```



This example uses “convention-based” naming by setting the `packageWithBaseClasses` property. Doing so means that the two last packages combine to make the name of the base test class. In our case, the contracts were placed under `src/test/resources/contracts/fraud`. Since you do not have two packages starting from the `contracts` folder, pick only one, which should be `fraud`. Add the `Base` suffix and capitalize `fraud`. That gives you the `FraudBase` test class name.

All the generated tests extend that class. Over there, you can set up your Spring Context or whatever is necessary. In this case, you should use [Rest Assured MVC](#) to start the server side `FraudDetectionController`. The following listing shows the `FraudBase` class:


```

/*
 * Copyright 2013-2020 the original author or authors.
 *
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 *     https://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */

package com.example.fraud;

import io.restassured.module.mockmvc.RestAssuredMockMvc;
import org.junit.jupiter.api.BeforeEach;

public class FraudBase {

    @BeforeEach
    public void setup() {
        RestAssuredMockMvc.standaloneSetup(new FraudDetectionController(),
            new FraudStatsController(stubbedStatsProvider()));
    }

    private StatsProvider stubbedStatsProvider() {
        return fraudType -> {
            switch (fraudType) {
                case DRUNKS:
                    return 100;
                case ALL:
                    return 200;
            }
        };
    }

    public void assertThatRejectionReasonIsNull(Object rejectionReason) {
        assert rejectionReason == null;
    }
}

```

Now, if you run the `./mvnw clean install`, you get something like the following output:

Results :

Tests in error:

ContractVerifierTest.validate_shouldMarkClientAsFraud:32 » IllegalState
Parsed...

This error occurs because you have a new contract from which a test was generated, and it failed since you have not implemented the feature. The auto-generated test would look like the following test method:

```
@Test
public void validate_shouldMarkClientAsFraud() throws Exception {
    // given:
    MockMvcRequestSpecification request = given()
        .header("Content-Type", "application/vnd.fraud.v1+json")
        .body("{\"client.id\":\"1234567890\",\"loanAmount\":99999}");

    // when:
    ResponseOptions response = given().spec(request)
        .put("/fraudcheck");

    // then:
    assertThat(response.statusCode()).isEqualTo(200);
    assertThat(response.header("Content-
Type")).matches("application/vnd.fraud.v1.json.*");
    // and:
    DocumentContext parsedJson =
    JsonPath.parse(response.getBody().asString());
    assertThatJson(parsedJson).field("[ 'fraudCheckStatus' ]").matches("[A-
Z]{5}");
    assertThatJson(parsedJson).field("[ 'rejection.reason' ]").isEqualTo("Amount
too high");
}
```

If you used the Groovy DSL, you can see that all the `producer()` parts of the Contract that were present in the `value(consumer(...), producer(...))` blocks got injected into the test. If you use YAML, the same applies for the `matchers` sections of the `response`.

Note that, on the producer side, you are also doing TDD. The expectations are expressed in the form of a test. This test sends a request to our own application with the URL, headers, and body defined in the contract. It also expects precisely defined values in the response. In other words, you have the `red` part of `red, green, and refactor`. It is time to convert the `red` into the `green`.

Write the Missing Implementation

Because you know the expected input and expected output, you can write the missing implementation as follows:

```
@RequestMapping(value = "/fraudcheck", method = PUT)
public FraudCheckResult fraudCheck(@RequestBody FraudCheck fraudCheck) {
    if (amountGreaterThanThreshold(fraudCheck)) {
        return new FraudCheckResult(FraudCheckStatus.FRAUD, AMOUNT_TOO_HIGH);
    }
    return new FraudCheckResult(FraudCheckStatus.OK, NO_REASON);
}
```

When you run `./mvnw clean install` again, the tests pass. Since the Spring Cloud Contract Verifier plugin adds the tests to the `generated-test-sources`, you can actually run those tests from your IDE.

Deploying Your Application

Once you finish your work, you can deploy your changes. To do so, you must first merge the branch by running the following commands:

```
$ git checkout master
$ git merge --no-ff contract-change-pr
$ git push origin master
```

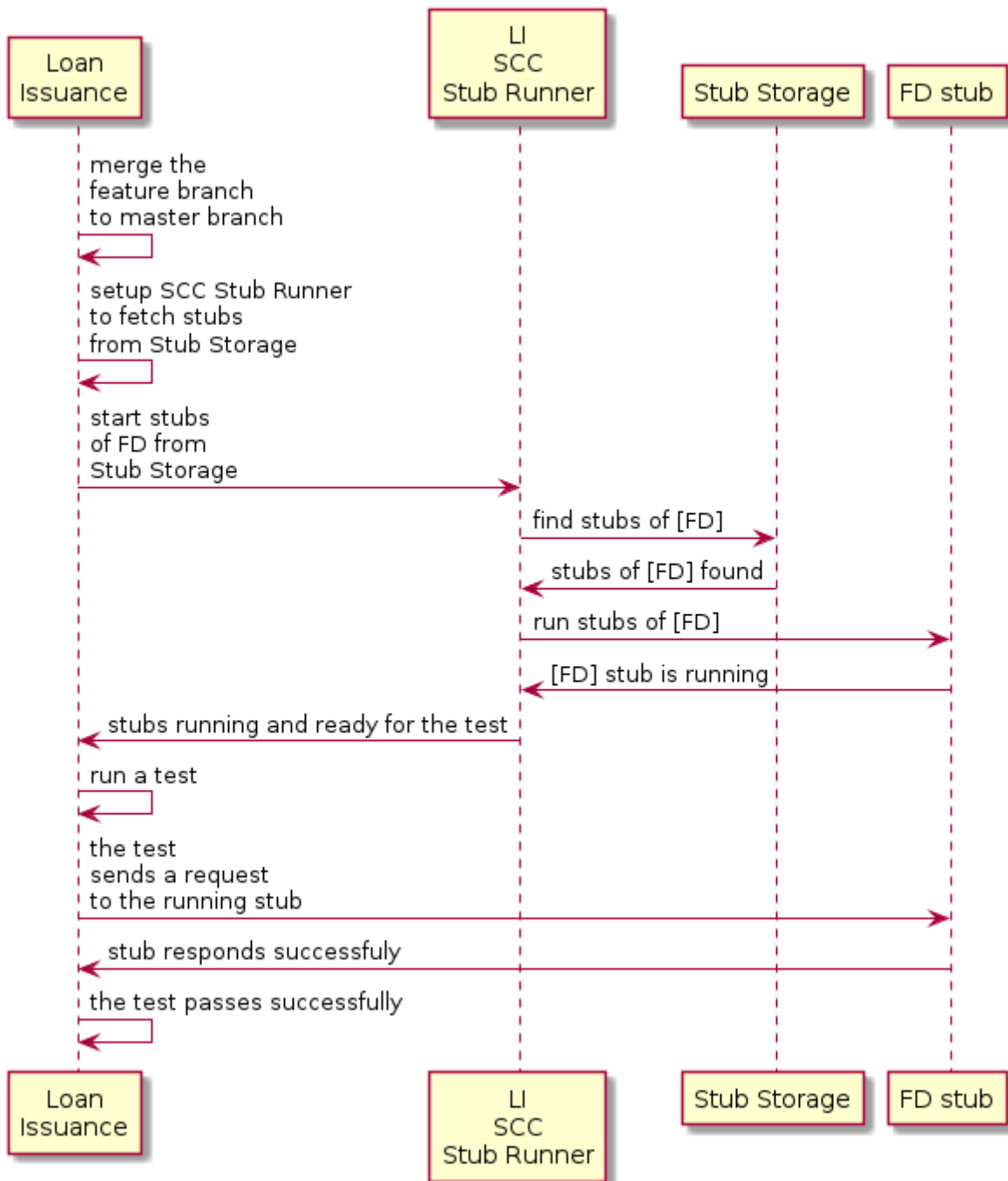
Your CI might run a command such as `./mvnw clean deploy`, which would publish both the application and the stub artifacts.

1.4.4. Consumer Side (Loan Issuance), Final Step

As a developer of the loan issuance service (a consumer of the Fraud Detection server), you need to:

- Merge our feature branch to `master`
- Switch to online mode of working

The following UML diagram shows the final state of the process:



Merging a Branch to Master

The following commands show one way to merge a branch into master with Git:

```

$ git checkout master
$ git merge --no-ff contract-change-pr
  
```

Working Online

Now you can disable the offline work for Spring Cloud Contract Stub Runner and indicate where the repository with your stubs is located. At this moment, the stubs of the server side are automatically downloaded from Nexus/Artifactory. You can set the value of `stubsMode` to `REMOTE`. The

following code shows an example of achieving the same thing by changing the properties:

```
stubrunner:  
  ids: 'com.example:http-server-dsl+:stubs:8080'  
  repositoryRoot: https://repo.spring.io/libs-snapshot
```

That's it. You have finished the tutorial.

1.5. Next Steps

Hopefully, this section provided some of the Spring Cloud Contract basics and got you on your way to writing your own applications. If you are a task-oriented type of developer, you might want to jump over to spring.io and check out some [getting started](#) guides that solve specific “How do I do that with Spring?” problems. We also have Spring Cloud Contract-specific “[how-to](#)” reference documentation.

Otherwise, the next logical step is to read [Using Spring Cloud Contract](#). If you are really impatient, you could also jump ahead and read about [Spring Cloud Contract features](#).

In addition, you can check out the following videos:

- "Consumer Driven Contracts and Your Microservice Architecture" by Olga Maciaszek-Sharma and Marcin Grzejszczak

The image shows a presentation slide titled "Who is who?" under a "Demo" header. It features two server icons. The left server is labeled "CONSUMER" and has "BLACK TERMINAL" and "BLACK IDE" listed below it. The right server is labeled "PRODUCER" and has "WHITE TERMINAL" and "WHITE IDE" listed below it. The slide is part of a presentation at BITCONF BYDGOSZCZ IT CONFERENCE on 14.09.2018, presented by Marcin Grzejszczak and Olga Maciaszek-Sharma. The slide number is 28 and the Spring logo is in the bottom right corner.

- "Contract Tests in the Enterprise" by Marcin Grzejszczak

Generating Stubs From Proxy

```
graph LR; Test[Test that calls Customer Rental History Service] --> Proxy[PROXY]; Proxy --> Service[Customer Rental History Service]; Service --> Payment[Payment processor]; Service --> Mainframe[Mainframe]; Proxy --> Record[Record traffic and dump stubs (e.g. once per day)]; Record --> Upload[Upload stubs for other teams to use];
```

SpringOne Platform by Pivotal

DEVOXX™
POLAND

- "Why Contract Tests Matter?" by Marcin Grzejszczak

IT talk ^{LUB} + LJUG



You can find the default project samples at [samples](#).

2. Using Spring Cloud Contract

This section goes into more detail about how you should use Spring Cloud Contract. It covers topics such as flows of how to work with Spring Cloud Contract. We also cover some Spring Cloud Contract best practices.

If you are starting out with Spring Cloud Contract, you should probably read the [Getting Started](#) guide before diving into this section.

2.1. Provider Contract Testing with Stubs in Nexus or Artifactory

You can check the [Developing Your First Spring Cloud Contract based application](#) link to see the provider contract testing with stubs in the Nexus or Artifactory flow.

2.2. Provider Contract Testing with Stubs in Git

In this flow, we perform the provider contract testing (the producer has no knowledge of how consumers use their API). The stubs are uploaded to a separate repository (they are not uploaded to Artifactory or Nexus).

2.2.1. Prerequisites

Before testing provider contracts with stubs in git, you must provide a git repository that contains all the stubs for each producer. For an example of such a project, see [this samples](#) or [this sample](#). As a result of pushing stubs there, the repository has the following structure:

```
$ tree .
├── META-INF
│   └── folder.with.group.id.as.its.name
│       ├── folder-with-artifact-id
│       │   └── folder-with-version
│       │       ├── contractA.groovy
│       │       ├── contractB.yml
│       │       └── contractC.groovy
```

You must also provide consumer code that has Spring Cloud Contract Stub Runner set up. For an example of such a project, see [this sample](#) and search for a `BeerControllerGitTest` test. You must also provide producer code that has Spring Cloud Contract set up, together with a plugin. For an example of such a project, see [this sample](#).

2.2.2. The Flow

The flow looks exactly as the one presented in [Developing Your First Spring Cloud Contract based application](#), but the `Stub Storage` implementation is a git repository.

You can read more about setting up a git repository and setting consumer and producer side in the [How To](#) page of the documentation.

2.2.3. Consumer setup

In order to fetch the stubs from a git repository instead of Nexus or Artifactory, you need to use the `git` protocol in the URL of the `repositoryRoot` property in Stub Runner. The following example shows how to set it up:

Annotation

```
@AutoConfigureStubRunner(  
    stubsMode = StubRunnerProperties.StubsMode.REMOTE,  
    repositoryRoot = "git://git@github.com:spring-cloud-samples/spring-cloud-  
contract-nodejs-contracts-git.git",  
    ids = "com.example:artifact-id:0.0.1")
```

JUnit 4 Rule

```
@Rule  
    public StubRunnerRule rule = new StubRunnerRule()  
        .downloadStub("com.example","artifact-id", "0.0.1")  
        .repoRoot("git://git@github.com:spring-cloud-samples/spring-cloud-  
contract-nodejs-contracts-git.git")  
        .stubsMode(StubRunnerProperties.StubsMode.REMOTE);
```

JUnit 5 Extension

```
@RegisterExtension  
    public StubRunnerExtension stubRunnerExtension = new StubRunnerExtension()  
        .downloadStub("com.example","artifact-id", "0.0.1")  
        .repoRoot("git://git@github.com:spring-cloud-samples/spring-cloud-  
contract-nodejs-contracts-git.git")  
        .stubsMode(StubRunnerProperties.StubsMode.REMOTE);
```

2.2.4. Setting up the Producer

To push the stubs to a git repository instead of Nexus or Artifactory, you need to use the **git** protocol in the URL of the plugin setup. Also you need to explicitly tell the plugin to push the stubs at the end of the build process. The following examples show how to do so in both Maven and Gradle:

Maven

```
<plugin>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-contract-maven-plugin</artifactId>
  <version>${spring-cloud-contract.version}</version>
  <extensions>>true</extensions>
  <configuration>
    <!-- Base class mappings etc. -->

    <!-- We want to pick contracts from a Git repository -->
    <contractsRepositoryUrl>git://git://git@github.com:spring-cloud-
samples/spring-cloud-contract-nodejs-contracts-git.git</contractsRepositoryUrl>

    <!-- We reuse the contract dependency section to set up the path
to the folder that contains the contract definitions. In our case the
path will be /groupId/artifactId/version/contracts -->
    <contractDependency>
      <groupId>${project.groupId}</groupId>
      <artifactId>${project.artifactId}</artifactId>
      <version>${project.version}</version>
    </contractDependency>

    <!-- The contracts mode can't be classpath -->
    <contractsMode>REMOTE</contractsMode>
  </configuration>
  <executions>
    <execution>
      <phase>package</phase>
      <goals>
        <!-- By default we will not push the stubs back to SCM,
you have to explicitly add it as a goal -->
        <goal>pushStubsToScm</goal>
      </goals>
    </execution>
  </executions>
</plugin>
```

Gradle

```
contracts {
    // We want to pick contracts from a Git repository
    contractDependency {
        stringNotation = "${project.group}:${project.name}:${project.version}"
    }
    /*
    We reuse the contract dependency section to set up the path
    to the folder that contains the contract definitions. In our case the
    path will be /groupId/artifactId/version/contracts
    */
    contractRepository {
        repositoryUrl = "git://git://git@github.com:spring-cloud-samples/spring-
cloud-contract-nodejs-contracts-git.git"
    }
    // The mode can't be classpath
    contractsMode = "REMOTE"
    // Base class mappings etc.
}

/*
In this scenario we want to publish stubs to SCM whenever
the `publish` task is run
*/
publish.dependsOn("publishStubsToScm")
```

You can read more about setting up a git repository in the [How To section](#) of the documentation.

2.3. Consumer Driven Contracts with Contracts on the Producer Side

See [Step-by-step Guide to Consumer Driven Contracts \(CDC\) with Contracts on the Producer Side](#) to see the Consumer Driven Contracts with contracts on the producer side flow.

2.4. Consumer Driven Contracts with Contracts in an External Repository

In this flow, we perform Consumer Driven Contract testing. The contract definitions are stored in a separate repository.

2.4.1. Prerequisites

To use consumer-driven contracts with the contracts held in an external repository, you need to set up a git repository that:

- Contains all the contract definitions for each producer.

- Can package the contract definitions in a JAR.
- For each contract producer, contains a way (for example, `pom.xml`) to install stubs locally through the Spring Cloud Contract Plugin (SCC Plugin).

For more information, see the [How To section](#), where we describe how to set up such a repository. For an example of such a project, see [this sample](#).

You also need consumer code that has Spring Cloud Contract Stub Runner set up. For an example of such a project, see [this sample](#). You also need producer code that has Spring Cloud Contract set up, together with a plugin. For an example of such a project, see [this sample](#). The stub storage is Nexus or Artifactory.

At a high level, the flow is as follows:

1. The consumer works with the contract definitions from the separate repository.
2. Once the consumer's work is done, a branch with working code is created on the consumer side, and a pull request is made to the separate repository that holds the contract definitions.
3. The producer takes over the pull request to the separate repository with contract definitions and installs the JAR with all contracts locally.
4. The producer generates tests from the locally stored JAR and writes the missing implementation to make the tests pass.
5. Once the producer's work is done, the pull request to the repository that holds the contract definitions is merged.
6. After the CI tool builds the repository with the contract definitions and the JAR with contract definitions gets uploaded to Nexus or Artifactory, the producer can merge its branch.
7. Finally, the consumer can switch to working online to fetch stubs of the producer from a remote location, and the branch can be merged to master.

2.4.2. Consumer Flow

The consumer:

1. Writes a test that would send a request to the producer.

The test fails due to no server being present.

2. Clones the repository that holds the contract definitions.
3. Sets up the requirements as contracts under the folder, with the consumer name as a subfolder of the producer.

For example, for a producer named `producer` and a consumer named `consumer`, the contracts would be stored under `src/main/resources/contracts/producer/consumer/`

4. Once the contracts are defined, installs the producer stubs to local storage, as the following example shows:

```

$ cd src/main/resource/contracts/producer
$ ./mvnw clean install

```

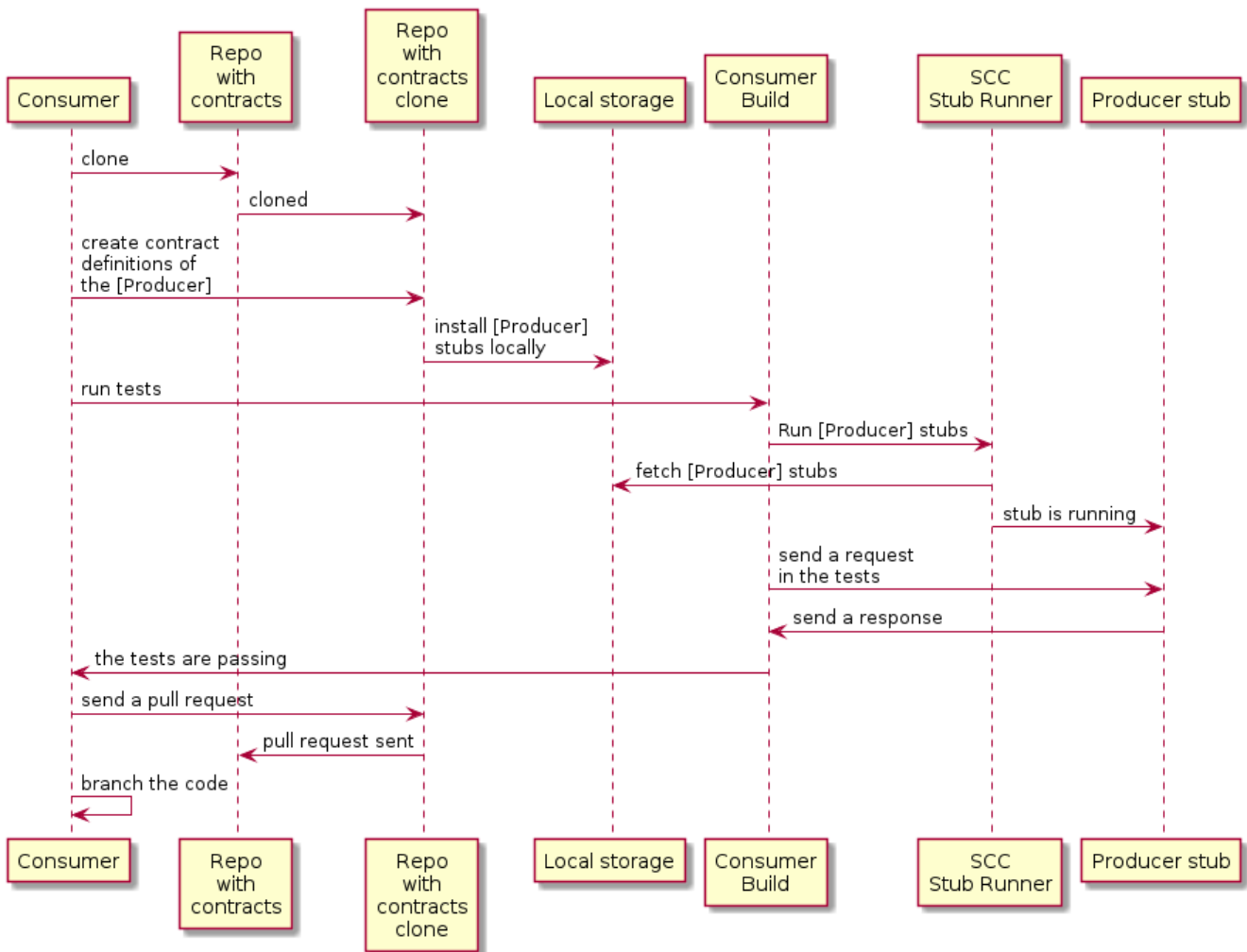
5. Sets up Spring Cloud Contract (SCC) Stub Runner in the consumer tests, to:

- Fetch the producer stubs from local storage.
- Work in the stubs-per-consumer mode (this enables consumer driven contracts mode).

The SCC Stub Runner:

- Fetches the producer stubs.
- Runs an in-memory HTTP server stub with the producer stubs. Now your test communicates with the HTTP server stub, and your tests pass.
- Creates a pull request to the repository with contract definitions, with the new contracts for the producer.
- Branches your consumer code, until the producer team has merged their code.

The following UML diagram shows the consumer flow:



2.4.3. Producer Flow

The producer:

1. Takes over the pull request to the repository with contract definitions. You can do it from the command line, as follows

```
$ git checkout -b the_branch_with_pull_request master
git pull https://github.com/user_id/project_name.git
the_branch_with_pull_request
```

2. Installs the contract definitions, as follows

```
$ ./mvnw clean install
```

3. Sets up the plugin to fetch the contract definitions from a JAR instead of from `src/test/resources/contracts`, as follows:

Maven

```
<plugin>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-contract-maven-plugin</artifactId>
  <version>${spring-cloud-contract.version}</version>
  <extensions>true</extensions>
  <configuration>
    <!-- We want to use the JAR with contracts with the following
coordinates -->
    <contractDependency>
      <groupId>com.example</groupId>
      <artifactId>beer-contracts</artifactId>
    </contractDependency>
    <!-- The JAR with contracts should be taken from Maven local -->
    <contractsMode>LOCAL</contractsMode>
    <!-- ... additional configuration -->
  </configuration>
</plugin>
```

Gradle

```
contracts {
  // We want to use the JAR with contracts with the following coordinates
  // group id `com.example`, artifact id `beer-contracts`, LATEST version and
  NO classifier
  contractDependency {
    stringNotation = 'com.example:beer-contracts:::'
  }
  // The JAR with contracts should be taken from Maven local
  contractsMode = "LOCAL"
  // Additional configuration
}
```

4. Runs the build to generate tests and stubs, as follows:

Maven

```
./mvnw clean install
```

Gradle

```
./gradlew clean build
```

5. Writes the missing implementation, to make the tests pass.

6. Merges the pull request to the repository with contract definitions, as follows:

```
$ git commit -am "Finished the implementation to make the contract tests pass"  
$ git checkout master  
$ git merge --no-ff the_branch_with_pull_request  
$ git push origin master
```

The CI system builds the project with the contract definitions and uploads the JAR with the contract definitions to Nexus or Artifactory.

7. Switches to working remotely.

8. Sets up the plugin so that the contract definitions are no longer taken from the local storage but from a remote location, as follows:

Maven

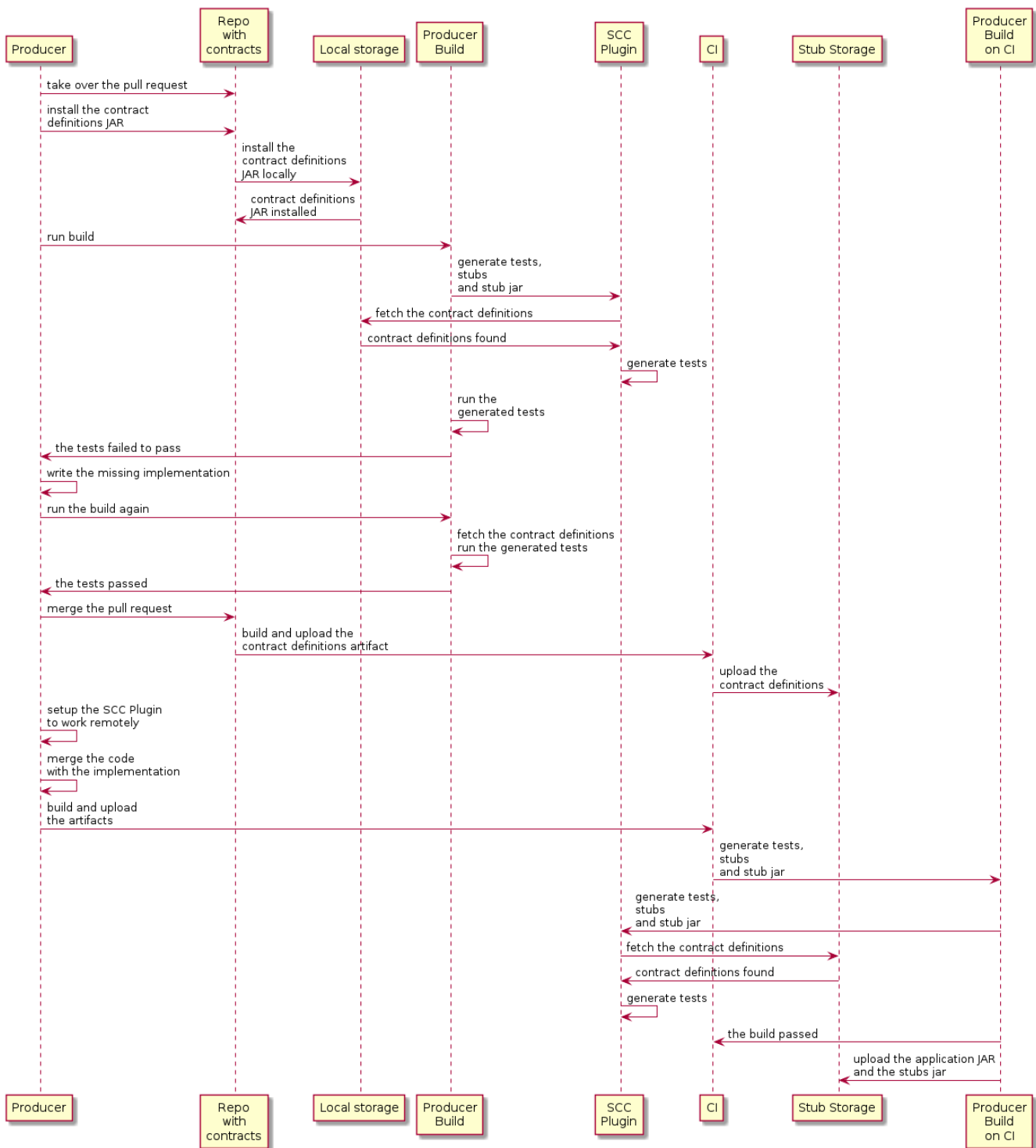
```
<plugin>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-contract-maven-plugin</artifactId>
  <version>${spring-cloud-contract.version}</version>
  <extensions>true</extensions>
  <configuration>
    <!-- We want to use the JAR with contracts with the following
coordinates -->
    <contractDependency>
      <groupId>com.example</groupId>
      <artifactId>beer-contracts</artifactId>
    </contractDependency>
    <!-- The JAR with contracts should be taken from a remote location -->
    <contractsMode>REMOTE</contractsMode>
    <!-- ... additional configuration -->
  </configuration>
</plugin>
```

Gradle

```
contracts {
  // We want to use the JAR with contracts with the following coordinates
  // group id `com.example`, artifact id `beer-contracts`, LATEST version and
  NO classifier
  contractDependency {
    stringNotation = 'com.example:beer-contracts:::'
  }
  // The JAR with contracts should be taken from a remote location
  contractsMode = "REMOTE"
  // Additional configuration
}
```

9. Merges the producer code with the new implementation.
10. The CI system:
 - Builds the project.
 - Generates tests, stubs, and the stub JAR.
 - Uploads the artifact with the application and the stubs to Nexus or Artifactory.

The following UML diagram shows the producer process:



2.5. Consumer-driven Contracts with Contracts on the Producer Side, Pushed to Git

You can read the [Step-by-step Guide to Consumer Driven Contracts \(CDC\) with contracts laying on the producer side](#) to see the consumer driven contracts with contracts on the producer side flow.

The stub storage implementation is a git repository. We describe its setup in the [Provider Contract Testing with Stubs in Git](#) section.

You can read more about setting up a git repository for the consumer and producer sides in the [How To](#) section of the documentation.

2.6. Provider Contract Testing with Stubs in Artifactory for a non-Spring Application

2.6.1. The Flow

You can read [Developing Your First Spring Cloud Contract-based Application](#) to see the flow for provider contract testing with stubs in Nexus or Artifactory.

2.6.2. Setting up the Consumer

For the consumer side, you can use a JUnit rule. That way, you need not start a Spring context. The following listing shows such a rule (in JUnit4 and JUnit 5);

JUnit 4 Rule

```
@Rule
public StubRunnerRule rule = new StubRunnerRule()
    .downloadStub("com.example","artifact-id", "0.0.1")
    .repoRoot("git://git@github.com:spring-cloud-samples/spring-cloud-
contract-nodejs-contracts-git.git")
    .stubsMode(StubRunnerProperties.StubsMode.REMOTE);
```

JUnit 5 Extension

```
@RegisterExtension
public StubRunnerExtension stubRunnerExtension = new StubRunnerExtension()
    .downloadStub("com.example","artifact-id", "0.0.1")
    .repoRoot("git://git@github.com:spring-cloud-samples/spring-cloud-
contract-nodejs-contracts-git.git")
    .stubsMode(StubRunnerProperties.StubsMode.REMOTE);
```

2.6.3. Setting up the Producer

By default, the Spring Cloud Contract Plugin uses Rest Assured's `MockMvc` setup for the generated tests. Since non-Spring applications do not use `MockMvc`, you can change the `testMode` to `EXPLICIT` to send a real request to an application bound at a specific port.

In this example, we use a framework called [Javalin](#) to start a non-Spring HTTP server.

Assume that we have the following application:

```
package com.example.demo;

import io.javalin.Javalin;

public class DemoApplication {

    public static void main(String[] args) {
        new DemoApplication().run(7000);
    }

    public Javalin start(int port) {
        return Javalin.create().start(port);
    }

    public Javalin registerGet(Javalin app) {
        return app.get("/", ctx -> ctx.result("Hello World"));
    }

    public Javalin run(int port) {
        return registerGet(start(port));
    }

}
```

Given that application, we can set up the plugin to use the **EXPLICIT** mode (that is, to send out requests to a real port), as follows:

Maven

```
<plugin>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-contract-maven-plugin</artifactId>
  <version>${spring-cloud-contract.version}</version>
  <extensions>>true</extensions>
  <configuration>
    <baseClassForTests>com.example.demo.BaseClass</baseClassForTests>
    <!-- This will setup the EXPLICIT mode for the tests -->
    <testMode>EXPLICIT</testMode>
  </configuration>
</plugin>
```

Gradle

```
contracts {
  // This will setup the EXPLICIT mode for the tests
  testMode = "EXPLICIT"
  baseClassForTests = "com.example.demo.BaseClass"
}
```

The base class might resemble the following:

```

import io.javalin.Javalin;
import io.restassured.RestAssured;
import org.junit.After;
import org.junit.Before;
import org.springframework.cloud.test.TestSocketUtils;

public class BaseClass {

    Javalin app;

    @Before
    public void setup() {
        // pick a random port
        int port = TestSocketUtils.findAvailableTcpPort();
        // start the application at a random port
        this.app = start(port);
        // tell Rest Assured where the started application is
        RestAssured.baseURI = "http://localhost:" + port;
    }

    @After
    public void close() {
        // stop the server after each test
        this.app.stop();
    }

    private Javalin start(int port) {
        // reuse the production logic to start a server
        return new DemoApplication().run(port);
    }
}

```

With such a setup:

- We have set up the Spring Cloud Contract plugin to use the **EXPLICIT** mode to send real requests instead of mocked ones.
- We have defined a base class that:
 - Starts the HTTP server on a random port for each test.
 - Sets Rest Assured to send requests to that port.
 - Closes the HTTP server after each test.

2.7. Provider Contract Testing with Stubs in Artifactory in a Non-JVM World

In this flow, we assume that:

- The API Producer and API Consumer are non-JVM applications.
- The contract definitions are written in YAML.
- The Stub Storage is Artifactory or Nexus.
- Spring Cloud Contract Docker (SCC Docker) and Spring Cloud Contract Stub Runner Docker (SCC Stub Runner Docker) images are used.

You can read more about how to use Spring Cloud Contract with Docker [here](#).

[Here](#), you can read a blog post about how to use Spring Cloud Contract in a polyglot world.

[Here](#), you can find a sample of a NodeJS application that uses Spring Cloud Contract both as a producer and a consumer.

2.7.1. Producer Flow

At a high level, the producer:

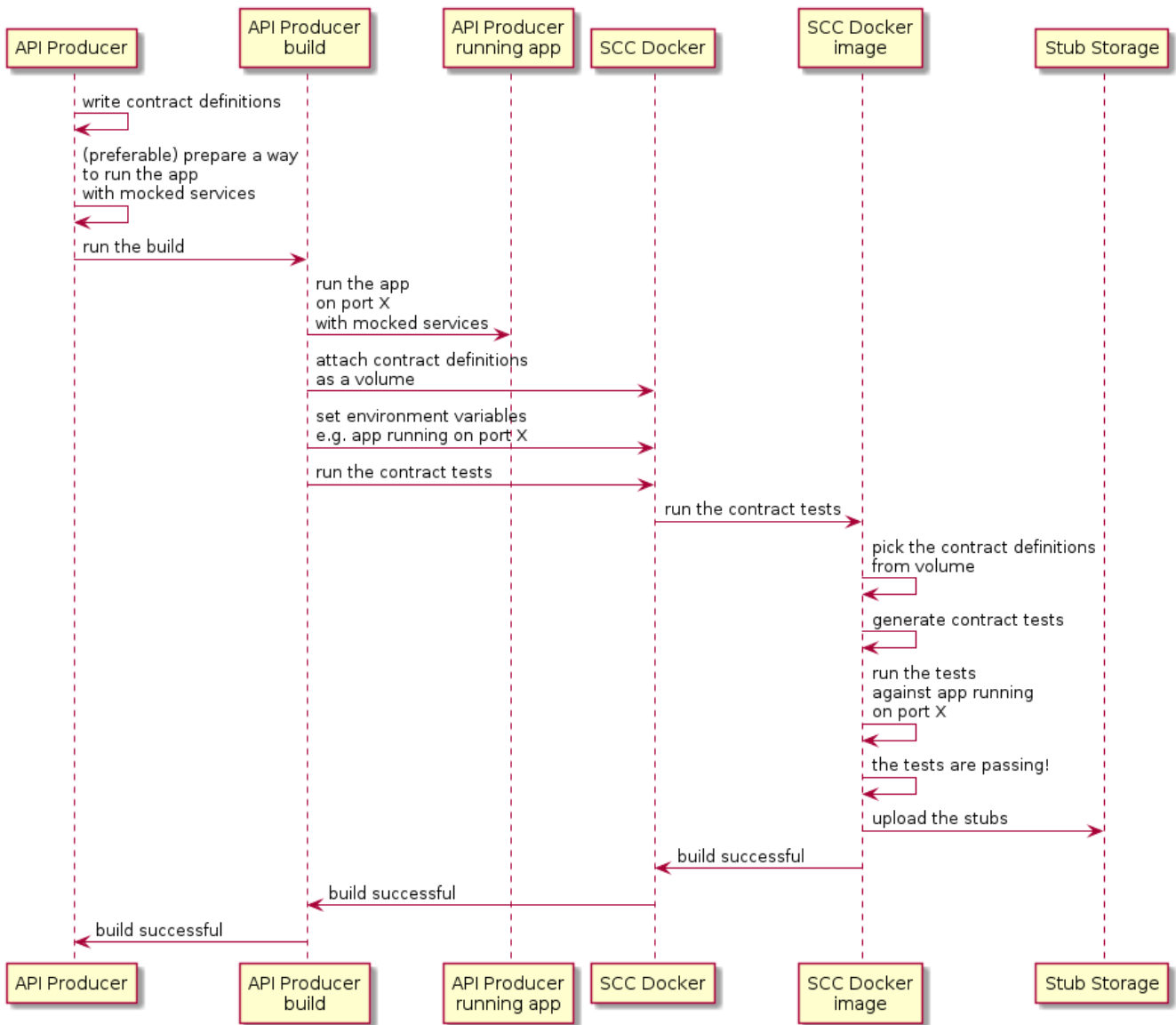
1. Writes contract definitions (for example, in YAML).
2. Sets up the build tool to:
 - a. Start the application with mocked services on a given port.

If mocking is not possible, you can set up the infrastructure and define tests in a stateful way.
 - b. Run the Spring Cloud Contract Docker image and pass the port of a running application as an environment variable.

The SCC Docker image: * Generates the tests from the attached volume. * Runs the tests against the running application.

Upon test completion, stubs get uploaded to a stub storage site (such as Artifactory or Git).

The following UML diagram shows the producer flow:



2.7.2. Consumer Flow

At a high level, the consumer:

1. Sets up the build tool to:

- Start the Spring Cloud Contract Stub Runner Docker image and start the stubs.

The environment variables configure:

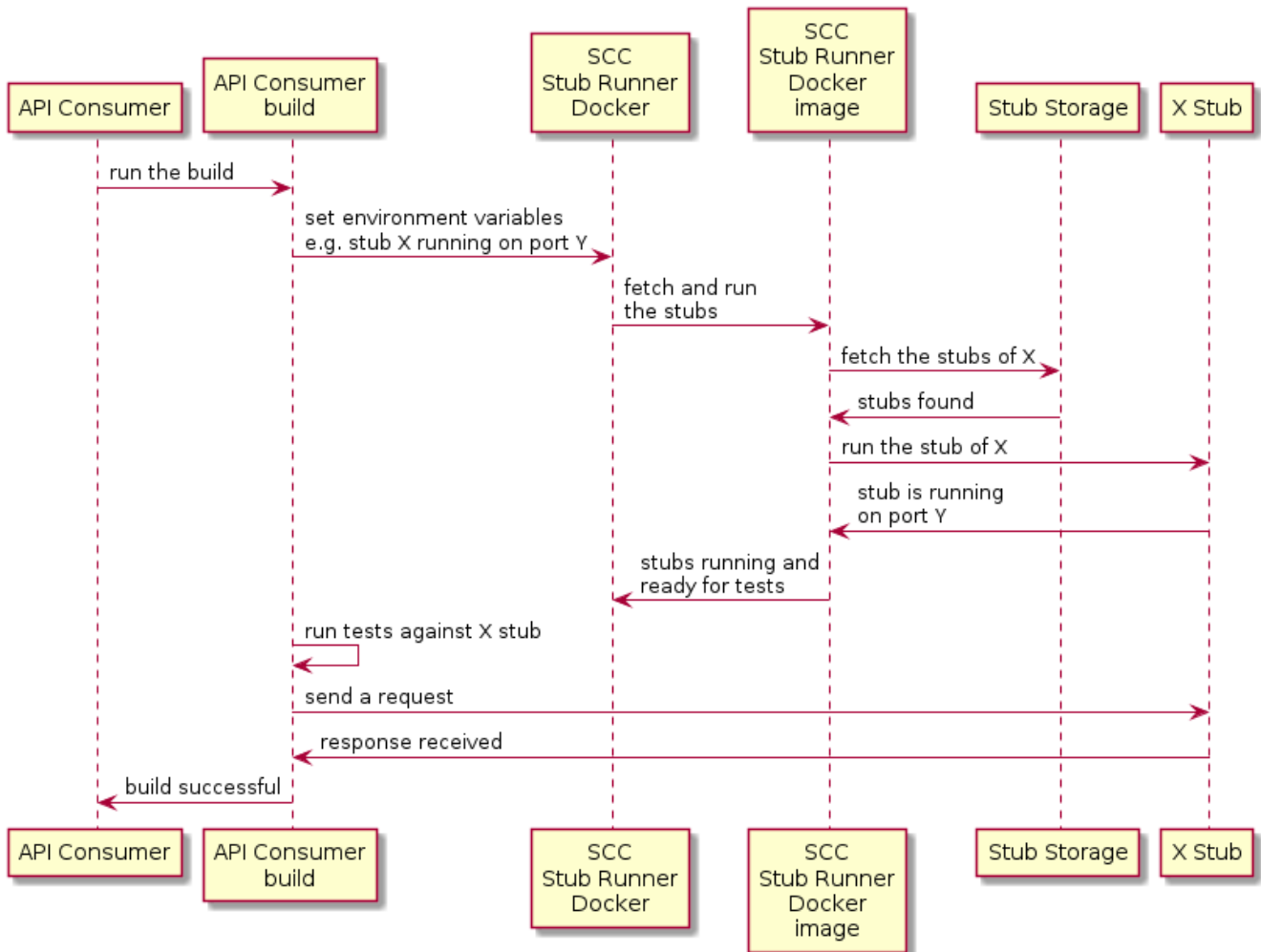
- The stubs to fetch.
- The location of the repositories.

Note that:

- To use the local storage, you can also attach it as a volume.
- The ports at which the stubs are running need to be exposed.

2. Run the application tests against the running stubs.

The following UML diagram shows the consumer flow:



2.8. Provider Contract Testing with REST Docs and Stubs in Nexus or Artifactory

In this flow, we do not use a Spring Cloud Contract plugin to generate tests and stubs. We write [Spring RESTDocs](#), and, from them, we automatically generate stubs. Finally, we set up our builds to package the stubs and upload them to the stub storage site — in our case, Nexus or Artifactory.

2.8.1. Producer Flow

As a producer, we:

1. Write RESTDocs tests of our API.
2. Add Spring Cloud Contract Stub Runner starter to our build (`spring-cloud-starter-contract-stub-runner`), as follows:

Maven

```
<dependencies>
  <dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-contract-stub-runner</artifactId>
    <scope>test</scope>
  </dependency>
</dependencies>

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

Gradle

```
dependencies {
    testImplementation 'org.springframework.cloud:spring-cloud-starter-
contract-stub-runner'
}

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

3. We set up the build tool to package our stubs, as follows:

Maven

```
<!-- pom.xml -->
<plugins>
  <plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-assembly-plugin</artifactId>
    <executions>
      <execution>
        <id>stub</id>
        <phase>prepare-package</phase>
        <goals>
          <goal>single</goal>
        </goals>
        <inherited>>false</inherited>
        <configuration>
          <attach>>true</attach>
          <descriptors>
            ${basedir}/src/assembly/stub.xml
          </descriptors>
        </configuration>
      </execution>
    </executions>
  </plugin>
</plugins>

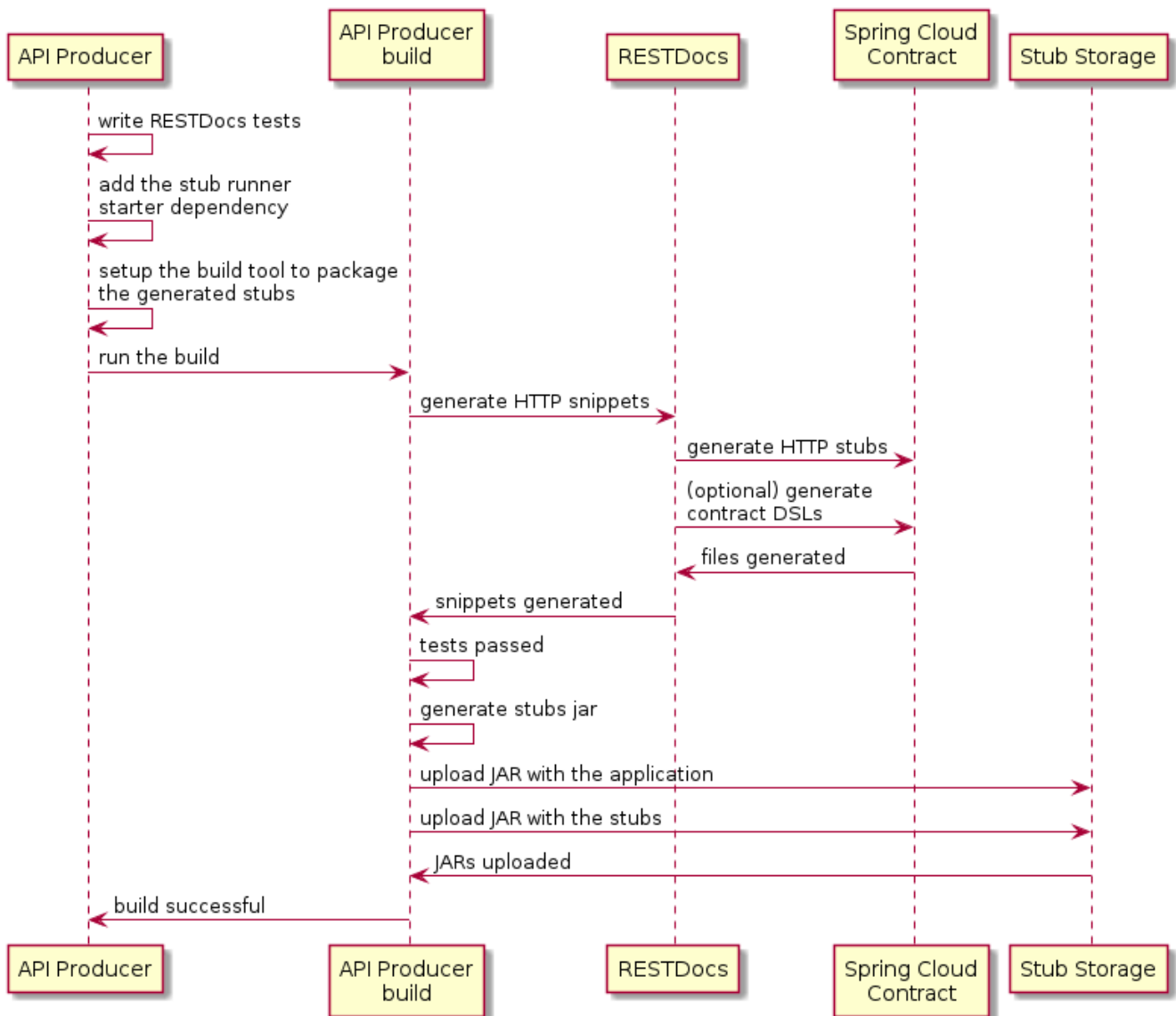
<!-- src/assembly/stub.xml -->
<assembly
  xmlns="http://maven.apache.org/plugins/maven-assembly-
plugin/assembly/1.1.3"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/plugins/maven-assembly-
plugin/assembly/1.1.3 http://maven.apache.org/xsd/assembly-1.1.3.xsd">
  <id>stubs</id>
  <formats>
    <format>jar</format>
  </formats>
  <includeBaseDirectory>>false</includeBaseDirectory>
  <fileSets>
    <fileSet>
      <directory>${project.build.directory}/generated-
snippets/stubs</directory>
      <outputDirectory>META-
INF/${project.groupId}/${project.artifactId}/${project.version}/mappings</output
Directory>
      <includes>
        <include>**/*</include>
      </includes>
    </fileSet>
  </fileSets>
</assembly>
```

Gradle

```
task stubsJar(type: Jar) {
    classifier = "stubs"
    into("META-INF/${project.group}/${project.name}/${project.version}/mappings") {
        include('**/*.*)
        from("${project.buildDir}/generated-snippets/stubs")
    }
}
// we need the tests to pass to build the stub jar
stubsJar.dependsOn(test)
bootJar.dependsOn(stubsJar)
```

Now, when we run the tests, stubs are automatically published and packaged.

The following UML diagram shows the producer flow:



2.8.2. Consumer Flow

Since the consumer flow is not affected by the tool used to generate the stubs, you can read [Developing Your First Spring Cloud Contract-based Application](#) to see the flow for consumer side of the provider contract testing with stubs in Nexus or Artifactory.

2.9. What to Read Next

You should now understand how you can use Spring Cloud Contract and some best practices that you should follow. You can now go on to learn about specific [Spring Cloud Contract features](#), or you could skip ahead and read about the [advanced features of Spring Cloud Contract](#).

3. Spring Cloud Contract Features

This section dives into the details of Spring Cloud Contract. Here you can learn about the key features that you may want to use and customize. If you have not already done so, you might want to read the "[Getting Started](#)" and "[Using Spring Cloud Contract](#)" sections, so that you have a good grounding in the basics.

3.1. Contract DSL

Spring Cloud Contract supports DSLs written in the following languages:

- Groovy
- YAML
- Java
- Kotlin



Spring Cloud Contract supports defining multiple contracts in a single file (In Groovy return a list instead of a single contract).

The following example shows a contract definition:

Groovy

```
org.springframework.cloud.contract.spec.Contract.make {
    request {
        method 'PUT'
        url '/api/12'
        headers {
            header 'Content-Type':
'application/vnd.org.springframework.cloud.contract.verifier.twitter-places-
analyzer.v1+json'
        }
        body '''\
    [{
        "created_at": "Sat Jul 26 09:38:57 +0000 2014",
        "id": 492967299297845248,
        "id_str": "492967299297845248",
        "text": "Gonna see you at Warsaw",
        "place":
        {
            "attributes": {},
            "bounding_box":
            {
                "coordinates":
                [[
                    [-77.119759,38.791645],
                    [-76.909393,38.791645],
                    [-76.909393,38.995548],
                    [-77.119759,38.995548]
                ]],
                "type": "Polygon"
            },
            "country": "United States",
            "country_code": "US",
            "full_name": "Washington, DC",
            "id": "01fbe706f872cb32",
            "name": "Washington",
            "place_type": "city",
            "url": "https://api.twitter.com/1/geo/id/01fbe706f872cb32.json"
        }
    }
    ...
}
response {
    status OK()
}
}
```

YAML

```
description: Some description
name: some name
priority: 8
ignored: true
request:
  url: /foo
  queryParameters:
    a: b
    b: c
  method: PUT
  headers:
    foo: bar
    fooReq: baz
  body:
    foo: bar
  matchers:
    body:
      - path: $.foo
        type: by_regex
        value: bar
    headers:
      - key: foo
        regex: bar
response:
  status: 200
  headers:
    foo2: bar
    foo3: foo33
    fooRes: baz
  body:
    foo2: bar
    foo3: baz
    nullValue: null
  matchers:
    body:
      - path: $.foo2
        type: by_regex
        value: bar
      - path: $.foo3
        type: by_command
        value: executeMe($it)
      - path: $.nullValue
        type: by_null
        value: null
    headers:
      - key: foo2
        regex: bar
      - key: foo3
        command: andMeToo($it)
```

Java

```
import java.util.Collection;
import java.util.Collections;
import java.util.function.Supplier;

import org.springframework.cloud.contract.spec.Contract;
import org.springframework.cloud.contract.verifier.util.ContractVerifierUtil;

class contract_rest implements Supplier<Collection<Contract>> {

    @Override
    public Collection<Contract> get() {
        return Collections.singletonList(Contract.make(c -> {
            c.description("Some description");
            c.name("some name");
            c.priority(8);
            c.ignored();
            c.request(r -> {
                r.url("/foo", u -> {
                    u.queryParameters(q -> {
                        q.parameter("a", "b");
                        q.parameter("b", "c");
                    });
                });
                r.method(r.PUT());
                r.headers(h -> {
                    h.header("foo", r.value(r.client(r.regex("bar")),
r.server("bar")));
                    h.header("fooReq", "baz");
                });
                r.body(ContractVerifierUtil.map().entry("foo", "bar"));
                r.bodyMatchers(m -> {
                    m.jsonPath("$.foo", m.byRegex("bar"));
                });
            });
            c.response(r -> {
                r.fixedDelayMilliseconds(1000);
                r.status(r.OK());
                r.headers(h -> {
                    h.header("foo2", r.value(r.server(r.regex("bar")),
r.client("bar")));
                    h.header("foo3", r.value(r.server(r.execute("andMeToo($it)")),
r.client("foo33")));
                    h.header("fooRes", "baz");
                });
                r.body(ContractVerifierUtil.map().entry("foo2",
"bar").entry("foo3", "baz").entry("nullValue", null));
                r.bodyMatchers(m -> {
                    m.jsonPath("$.foo2", m.byRegex("bar"));
                    m.jsonPath("$.foo3", m.byCommand("executeMe($it)"));
                });
            });
        });
    }
}
```



```
        m.jsonPath("$.nullValue", m.byNull());
    });
});
}
}
```

```
import org.springframework.cloud.contract.spec.ContractDsl.Companion.contract
import org.springframework.cloud.contract.spec.withQueryParameters

contract {
    name = "some name"
    description = "Some description"
    priority = 8
    ignored = true
    request {
        url = url("/foo") withQueryParameters {
            parameter("a", "b")
            parameter("b", "c")
        }
        method = PUT
        headers {
            header("foo", value(client(regex("bar")), server("bar")))
            header("fooReq", "baz")
        }
        body = body(mapOf("foo" to "bar"))
        bodyMatchers {
            jsonPath("$.foo", byRegex("bar"))
        }
    }
    response {
        delay = fixedMilliseconds(1000)
        status = OK
        headers {
            header("foo2", value(server(regex("bar")), client("bar")))
            header("foo3", value(server(execute("andMeToo(\${it}")),
client("foo33"))))
            header("fooRes", "baz")
        }
        body = body(mapOf(
            "foo" to "bar",
            "foo3" to "baz",
            "nullValue" to null
        ))
        bodyMatchers {
            jsonPath("$.foo2", byRegex("bar"))
            jsonPath("$.foo3", byCommand("executeMe(\${it}"))
            jsonPath("$.nullValue", byNull)
        }
    }
}
```



You can compile contracts to stubs mapping by using the following standalone Maven command:

```
mvn org.springframework.cloud:spring-cloud-contract-maven-plugin:convert
```

3.1.1. Contract DSL in Groovy

If you are not familiar with Groovy, do not worry. You can use Java syntax in the Groovy DSL files as well.

If you decide to write the contract in Groovy, do not be alarmed if you have not used Groovy before. Knowledge of the language is not really needed, as the Contract DSL uses only a tiny subset of it (only literals, method calls, and closures). Also, the DSL is statically typed, to make it programmer-readable without any knowledge of the DSL itself.



Remember that, inside the Groovy contract file, you have to provide the fully qualified name to the `Contract` class and `make` static imports, such as `org.springframework.cloud.spec.Contract.make { ... }`. You can also provide an import to the `Contract` class (`import org.springframework.cloud.spec.Contract`) and then call `Contract.make { ... }`.

3.1.2. Contract DSL in Java

To write a contract definition in Java, you need to create a class that implements either the `Supplier<Contract>` interface (for a single contract) or `Supplier<Collection<Contract>>` (for multiple contracts).

You can also write the contract definitions under `src/test/java` (for example, `src/test/java/contracts`) so that you do not have to modify the classpath of your project. In this case, you have to provide a new location of contract definitions to your Spring Cloud Contract plugin.

The following example (in both Maven and Gradle) has the contract definitions under `src/test/java`:

Maven

```
<plugin>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-contract-maven-plugin</artifactId>
  <version>${spring-cloud-contract.version}</version>
  <extensions>true</extensions>
  <configuration>
    <contractsDirectory>src/test/java/contracts</contractsDirectory>
  </configuration>
</plugin>
```

Gradle

```
contracts {
    contractsDslDir = new File(project.rootDir, "src/test/java/contracts")
}
```

3.1.3. Contract DSL in Kotlin

To get started with writing contracts in Kotlin, you need to start with a (newly created) Kotlin Script file (*.kts*). As with the Java DSL, you can put your contracts in any directory of your choice. By default, the Maven plugin will look at the `src/test/resources/contracts` directory and Gradle plugin will look at the `src/contractTest/resources/contracts` directory.



Since 3.0.0, the Gradle plugin will also look at the legacy directory `src/test/resources/contracts` for migration purposes. When contracts are found in this directory, a warning will be logged during your build.

You need to explicitly pass the `spring-cloud-contract-spec-kotlin` dependency to your project plugin setup. The following example (in both Maven and Gradle) shows how to do so:

Maven

```
<plugin>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-contract-maven-plugin</artifactId>
  <version>${spring-cloud-contract.version}</version>
  <extensions>>true</extensions>
  <configuration>
    <!-- some config -->
  </configuration>
  <dependencies>
    <dependency>
      <groupId>org.springframework.cloud</groupId>
      <artifactId>spring-cloud-contract-spec-kotlin</artifactId>
      <version>${spring-cloud-contract.version}</version>
    </dependency>
  </dependencies>
</plugin>

<dependencies>
  <!-- Remember to add this for the DSL support in the IDE and on the
consumer side -->
  <dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-contract-spec-kotlin</artifactId>
    <scope>test</scope>
  </dependency>
</dependencies>
```

Gradle

```
buildscript {
    repositories {
        // ...
    }
    dependencies {
        classpath "org.springframework.cloud:spring-cloud-contract-gradle-
plugin:${scContractVersion}"
    }
}

dependencies {
    // ...

    // Remember to add this for the DSL support in the IDE and on the consumer
side
    testImplementation "org.springframework.cloud:spring-cloud-contract-spec-
kotlin"
    // Kotlin versions are very particular down to the patch version. The
<kotlin_version> needs to be the same as you have imported for your project.
    testImplementation "org.jetbrains.kotlin:kotlin-scripting-compiler-
embeddable:<kotlin_version>"
}
```



Remember that, inside the Kotlin Script file, you have to provide the fully qualified name to the `ContractDSL` class. Generally you would use its contract function as follows: `org.springframework.cloud.contract.spec.ContractDSL.contract { ... }`. You can also provide an import to the `contract` function (`import org.springframework.cloud.contract.spec.ContractDSL.Companion.contract`) and then call `contract { ... }`.

3.1.4. Contract DSL in YAML

To see a schema of a YAML contract, visit the [YML Schema](#) page.

3.1.5. Limitations



The support for verifying the size of JSON arrays is experimental. If you want to turn it on, set the value of the following system property to `true`: `spring.cloud.contract.verifier.assert.size`. By default, this feature is set to `false`. You can also set the `assertJsonSize` property in the plugin configuration.



Because JSON structure can have any form, it can be impossible to parse it properly when using the Groovy DSL and the `value(consumer(...), producer(...))` notation in `GString`. That is why you should use the Groovy Map notation.

3.1.6. Common Top-Level Elements

The following sections describe the most common top-level elements:

- [Description](#)
- [Name](#)
- [Ignoring Contracts](#)
- [Contracts in Progress](#)
- [Passing Values from Files](#)
- [Metadata](#)

Description

You can add a `description` to your contract. The description is arbitrary text. The following code shows an example:

Groovy

```
org.springframework.cloud.contract.spec.Contract.make {  
    description(''  
given:  
    An input  
when:  
    Sth happens  
then:  
    Output  
''')  
}
```


YAML

```
description: Some description
name: some name
priority: 8
ignored: true
request:
  url: /foo
  queryParameters:
    a: b
    b: c
  method: PUT
  headers:
    foo: bar
    fooReq: baz
  body:
    foo: bar
  matchers:
    body:
      - path: $.foo
        type: by_regex
        value: bar
    headers:
      - key: foo
        regex: bar
response:
  status: 200
  headers:
    foo2: bar
    foo3: foo33
    fooRes: baz
  body:
    foo2: bar
    foo3: baz
    nullValue: null
  matchers:
    body:
      - path: $.foo2
        type: by_regex
        value: bar
      - path: $.foo3
        type: by_command
        value: executeMe($it)
      - path: $.nullValue
        type: by_null
        value: null
    headers:
      - key: foo2
        regex: bar
      - key: foo3
        command: andMeToo($it)
```

Java

```
Contract.make(c -> {
    c.description("Some description");
}));
```

Kotlin

```
contract {
    description = ""
given:
    An input
when:
    Sth happens
then:
    Output
""
}
```

Name

You can provide a name for your contract. Assume that you provide the following name: **should register a user**. If you do so, the name of the autogenerated test is **validate_should_register_a_user**. Also, the name of the stub in a WireMock stub is **should_register_a_user.json**.



You must ensure that the name does not contain any characters that make the generated test not compile. Also, remember that, if you provide the same name for multiple contracts, your autogenerated tests fail to compile and your generated stubs override each other.

The following example shows how to add a name to a contract:

Groovy

```
org.springframework.cloud.contract.spec.Contract.make {  
    name("some_special_name")  
}
```

YAML

```
name: some name
```

Java

```
Contract.make(c -> {  
    c.name("some name");  
}));
```

Kotlin

```
contract {  
    name = "some_special_name"  
}
```

Ignoring Contracts

If you want to ignore a contract, you can either set a value for ignored contracts in the plugin configuration or set the **ignored** property on the contract itself. The following example shows how to do so:

Groovy

```
org.springframework.cloud.contract.spec.Contract.make {  
    ignored()  
}
```

YAML

```
ignored: true
```

Java

```
Contract.make(c -> {  
    c.ignored();  
}));
```

Kotlin

```
contract {  
    ignored = true  
}
```

Contracts in Progress

A contract in progress does not generate tests on the producer side but does allow generation of stubs.



Use this feature with caution as it may lead to false positives, because you generate stubs for your consumers to use without actually having the implementation in place.

If you want to set a contract in progress, the following example shows how to do so:

Groovy

```
org.springframework.cloud.contract.spec.Contract.make {
    inProgress()
}
```

YAML

```
inProgress: true
```

Java

```
Contract.make(c -> {
    c.inProgress();
});
```

Kotlin

```
contract {
    inProgress = true
}
```

You can set the value of the `failOnInProgress` Spring Cloud Contract plugin property to ensure that your build breaks when at least one contract in progress remains in your sources.

Passing Values from Files

Starting with version `1.2.0`, you can pass values from files. Assume that you have the following resources in your project:

```
├── src
│   ├── test
│   │   ├── resources
│   │   │   ├── contracts
│   │   │   │   ├── readFromFile.groovy
│   │   │   │   ├── request.json
│   │   │   │   └── response.json
```

Further assume that your contract is as follows:

Groovy

```
/*
 * Copyright 2013-2020 the original author or authors.
 *
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 *     https://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */

import org.springframework.cloud.contract.spec.Contract

Contract.make {
    request {
        method('PUT')
        headers {
            contentType(applicationJson())
        }
        body(file("request.json"))
        url("/1")
    }
    response {
        status OK()
        body(file("response.json"))
        headers {
            contentType(applicationJson())
        }
    }
}
```

YAML

```
request:
  method: GET
  url: /foo
  bodyFromFile: request.json
response:
  status: 200
  bodyFromFile: response.json
```

Java

```
import java.util.Collection;
import java.util.Collections;
import java.util.function.Supplier;

import org.springframework.cloud.contract.spec.Contract;

class contract_rest_from_file implements Supplier<Collection<Contract>> {

    @Override
    public Collection<Contract> get() {
        return Collections.singletonList(Contract.make(c -> {
            c.request(r -> {
                r.url("/foo");
                r.method(r.GET());
                r.body(r.file("request.json"));
            });
            c.response(r -> {
                r.status(r.OK());
                r.body(r.file("response.json"));
            });
        }));
    }
}
```

Kotlin

```
import org.springframework.cloud.contract.spec.ContractDsl.Companion.contract

contract {
    request {
        url = url("/1")
        method = PUT
        headers {
            contentType = APPLICATION_JSON
        }
        body = bodyFromFile("request.json")
    }
    response {
        status = OK
        body = bodyFromFile("response.json")
        headers {
            contentType = APPLICATION_JSON
        }
    }
}
```

Further assume that the JSON files are as follows:

request.json

```
{
  "status": "REQUEST"
}
```

response.json

```
{
  "status": "RESPONSE"
}
```

When test or stub generation takes place, the contents of the `request.json` and `response.json` files are passed to the body of a request or a response. The name of the file needs to be a file in a location relative to the folder in which the contract resides.

If you need to pass the contents of a file in binary form, you can use the `fileAsBytes` method in the coded DSL or a `bodyFromFileAsBytes` field in YAML.

The following example shows how to pass the contents of binary files:

Groovy

```
import org.springframework.cloud.contract.spec.Contract

Contract.make {
    request {
        url("/1")
        method(PUT())
        headers {
            contentType(applicationOctetStream())
        }
        body(fileAsBytes("request.pdf"))
    }
    response {
        status 200
        body(fileAsBytes("response.pdf"))
        headers {
            contentType(applicationOctetStream())
        }
    }
}
```

YAML

```
request:
  url: /1
  method: PUT
  headers:
    Content-Type: application/octet-stream
  bodyFromFileAsBytes: request.pdf
response:
  status: 200
  bodyFromFileAsBytes: response.pdf
  headers:
    Content-Type: application/octet-stream
```

Java

```
import java.util.Collection;
import java.util.Collections;
import java.util.function.Supplier;

import org.springframework.cloud.contract.spec.Contract;

class contract_rest_from_pdf implements Supplier<Collection<Contract>> {

    @Override
    public Collection<Contract> get() {
        return Collections.singletonList(Contract.make(c -> {
            c.request(r -> {
                r.url("/1");
                r.method(r.PUT());
                r.body(r.fileAsBytes("request.pdf"));
                r.headers(h -> {
                    h.contentType(h.applicationOctetStream());
                });
            });
            c.response(r -> {
                r.status(r.OK());
                r.body(r.fileAsBytes("response.pdf"));
                r.headers(h -> {
                    h.contentType(h.applicationOctetStream());
                });
            });
        }));
    }
}
```

Kotlin

```
import org.springframework.cloud.contract.spec.ContractDsl.Companion.contract

contract {
    request {
        url = url("/1")
        method = PUT
        headers {
            contentType = APPLICATION_OCTET_STREAM
        }
        body = bodyFromFileAsBytes("contracts/request.pdf")
    }
    response {
        status = OK
        body = bodyFromFileAsBytes("contracts/response.pdf")
        headers {
            contentType = APPLICATION_OCTET_STREAM
        }
    }
}
```



You should use this approach whenever you want to work with binary payloads, both for HTTP and messaging.

Metadata

You can add `metadata` to your contract. Via the metadata you can pass in configuration to extensions. Below you can find an example of using the `wiremock` key. Its value is a map whose key is `stubMapping` and value being WireMock's `StubMapping` object. Spring Cloud Contract is able to patch parts of your generated stub mapping with your custom code. You may want to do that in order to add webhooks, custom delays or integrate with third party WireMock extensions.

groovy

```
Contract.make {
  request {
    method GET()
    url '/drunks'
  }
  response {
    status OK()
    body([
      count: 100
    ])
    headers {
      contentType("application/json")
    }
  }
  metadata([
    wiremock: [
      stubMapping: '''\
        {
          "response" : {
            "fixedDelayMilliseconds": 2000
          }
        }
      ...
    ]
  ])
}
```

yaml

```
name: "should count all frauds"
request:
  method: GET
  url: /yamlfrauds
response:
  status: 200
  body:
    count: 200
  headers:
    Content-Type: application/json
metadata:
  wiremock:
    stubMapping: >
      {
        "response" : {
          "fixedDelayMilliseconds": 2000
        }
      }
}
```

java

```
Contract.make(c -> {
    c.metadata(MetadataUtil.map().entry("wiremock",
    ContractVerifierUtil.map().entry("stubMapping",
        "{ \"response\" : { \"fixedDelayMilliseconds\" : 2000 } }"));
}));
```

kotlin

```
contract {
    metadata("wiremock" to ("stubmapping" to ""
{
    "response" : {
        "fixedDelayMilliseconds": 2000
    }
}""))
}
```

In the following sections you can find examples of the supported metadata entries.

3.2. Contracts for HTTP

Spring Cloud Contract lets you verify applications that use REST or HTTP as a means of communication. Spring Cloud Contract verifies that, for a request that matches the criteria from the **request** part of the contract, the server provides a response that is in keeping with the **response** part of the contract. Subsequently, the contracts are used to generate WireMock stubs that, for any request matching the provided criteria, provide a suitable response.

3.2.1. HTTP Top-Level Elements

You can call the following methods in the top-level closure of a contract definition:

- **request**: Mandatory
- **response** : Mandatory
- **priority**: Optional

The following example shows how to define an HTTP request contract:

Groovy

```
org.springframework.cloud.contract.spec.Contract.make {
    // Definition of HTTP request part of the contract
    // (this can be a valid request or invalid depending
    // on type of contract being specified).
    request {
        method GET()
        url "/foo"
        //...
    }

    // Definition of HTTP response part of the contract
    // (a service implementing this contract should respond
    // with following response after receiving request
    // specified in "request" part above).
    response {
        status 200
        //...
    }

    // Contract priority, which can be used for overriding
    // contracts (1 is highest). Priority is optional.
    priority 1
}
```

YAML

```
priority: 8
request:
...
response:
...
```

Java

```
org.springframework.cloud.contract.spec.Contract.make(c -> {
    // Definition of HTTP request part of the contract
    // (this can be a valid request or invalid depending
    // on type of contract being specified).
    c.request(r -> {
        r.method(r.GET());
        r.url("/foo");
        // ...
    });

    // Definition of HTTP response part of the contract
    // (a service implementing this contract should respond
    // with following response after receiving request
    // specified in "request" part above).
    c.response(r -> {
        r.status(200);
        // ...
    });

    // Contract priority, which can be used for overriding
    // contracts (1 is highest). Priority is optional.
    c.priority(1);
});
```

Kotlin

```
contract {
    // Definition of HTTP request part of the contract
    // (this can be a valid request or invalid depending
    // on type of contract being specified).
    request {
        method = GET
        url = url("/foo")
        // ...
    }

    // Definition of HTTP response part of the contract
    // (a service implementing this contract should respond
    // with following response after receiving request
    // specified in "request" part above).
    response {
        status = OK
        // ...
    }

    // Contract priority, which can be used for overriding
    // contracts (1 is highest). Priority is optional.
    priority = 1
}
```



If you want to make your contract have a higher priority, you need to pass a lower number to the `priority` tag or method. For example, a `priority` with a value of `5` has higher priority than a `priority` with a value of `10`.

3.2.2. HTTP Request

The HTTP protocol requires only the method and the URL to be specified in a request. The same information is mandatory in request definition of the contract.

The following example shows a contract for a request:

Groovy

```
org.springframework.cloud.contract.spec.Contract.make {
    request {
        // HTTP request method (GET/POST/PUT/DELETE).
        method 'GET'

        // Path component of request URL is specified as follows.
        urlPath('/users')
    }

    response {
        //...
        status 200
    }
}
```

YAML

```
method: PUT
url: /foo
```

Java

```
org.springframework.cloud.contract.spec.Contract.make(c -> {
    c.request(r -> {
        // HTTP request method (GET/POST/PUT/DELETE).
        r.method("GET");

        // Path component of request URL is specified as follows.
        r.urlPath("/users");
    });

    c.response(r -> {
        // ...
        r.status(200);
    });
});
```

Kotlin

```
contract {
    request {
        // HTTP request method (GET/POST/PUT/DELETE).
        method = method("GET")

        // Path component of request URL is specified as follows.
        urlPath = path("/users")
    }
    response {
        // ...
        status = code(200)
    }
}
```

You can specify an absolute rather than a relative `url`, but using `urlPath` is the recommended way, as doing so makes the tests be host-independent.

The following example uses `url`:

Groovy

```
org.springframework.cloud.contract.spec.Contract.make {
    request {
        method 'GET'

        // Specifying `url` and `urlPath` in one contract is illegal.
        url('http://localhost:8888/users')
    }

    response {
        //...
        status 200
    }
}
```

YAML

```
request:
  method: PUT
  urlPath: /foo
```

Java

```
org.springframework.cloud.contract.spec.Contract.make(c -> {
    c.request(r -> {
        r.method("GET");

        // Specifying `url` and `urlPath` in one contract is illegal.
        r.url("http://localhost:8888/users");
    });

    c.response(r -> {
        // ...
        r.status(200);
    });
});
```

Kotlin

```
contract {
    request {
        method = GET

        // Specifying `url` and `urlPath` in one contract is illegal.
        url("http://localhost:8888/users")
    }
    response {
        // ...
        status = OK
    }
}
```

`request` may contain query parameters, as the following example (which uses `urlPath`) shows:

Groovy

```
org.springframework.cloud.contract.spec.Contract.make {
    request {
        //...
        method GET()

        urlPath('/users') {

            // Each parameter is specified in form
            // `paramName` : paramValue` where parameter value
            // may be a simple literal or one of matcher functions,
            // all of which are used in this example.
            queryParameters {

                // If a simple literal is used as value
                // default matcher function is used (equalTo)
                parameter 'limit': 100

                // `equalTo` function simply compares passed value
                // using identity operator (==).
                parameter 'filter': equalTo("email")

                // `containing` function matches strings
                // that contains passed substring.
                parameter 'gender': value(consumer(containing("[mf]")),
producer('mf'))

                // `matching` function tests parameter
                // against passed regular expression.
                parameter 'offset': value(consumer(matching("[0-9]+")),
producer(123))

                // `notMatching` functions tests if parameter
                // does not match passed regular expression.
                parameter 'loginStartsWith':
value(consumer(notMatching(".{0,2}")), producer(3))
            }
        }

        //...
    }

    response {
        //...
        status 200
    }
}
```

YAML

```
request:  
...  
queryParameters:  
  a: b  
  b: c
```

```
org.springframework.cloud.contract.spec.Contract.make(c -> {
    c.request(r -> {
        // ...
        r.method(r.GET());

        r.urlPath("/users", u -> {

            // Each parameter is specified in form
            // `paramName` : paramValue` where parameter value
            // may be a simple literal or one of matcher functions,
            // all of which are used in this example.
            u.queryParameters(q -> {

                // If a simple literal is used as value
                // default matcher function is used (equalTo)
                q.parameter("limit", 100);

                // `equalTo` function simply compares passed value
                // using identity operator (==).
                q.parameter("filter", r.equalTo("email"));

                // `containing` function matches strings
                // that contains passed substring.
                q.parameter("gender", r.value(r.consumer(r.containing("[mf]")),
r.producer("mf"))));

                // `matching` function tests parameter
                // against passed regular expression.
                q.parameter("offset", r.value(r.consumer(r.matching("[0-9]+")),
r.producer(123)));

                // `notMatching` functions tests if parameter
                // does not match passed regular expression.
                q.parameter("loginStartsWith",
r.value(r.consumer(r.notMatching(".{0,2}")), r.producer(3)));
            });
        });

        // ...
    });

    c.response(r -> {
        // ...
        r.status(200);
    });
});
```

Kotlin

```
contract {
    request {
        // ...
        method = GET

        // Each parameter is specified in form
        // `paramName` : paramValue` where parameter value
        // may be a simple literal or one of matcher functions,
        // all of which are used in this example.
        urlPath = path("/users") withQueryParameters {
            // If a simple literal is used as value
            // default matcher function is used (equalTo)
            parameter("limit", 100)

            // `equalTo` function simply compares passed value
            // using identity operator (==).
            parameter("filter", equalTo("email"))

            // `containing` function matches strings
            // that contains passed substring.
            parameter("gender", value(consumer(containing("[mf]")),
producer("mf")))

            // `matching` function tests parameter
            // against passed regular expression.
            parameter("offset", value(consumer(matching("[0-9]+")),
producer(123)))

            // `notMatching` functions tests if parameter
            // does not match passed regular expression.
            parameter("loginStartsWith", value(consumer(notMatching(".{0,2}")),
producer(3)))
        }
    }
    // ...
}
response {
    // ...
    status = code(200)
}
}
```

request can contain additional request headers, as the following example shows:

Groovy

```
org.springframework.cloud.contract.spec.Contract.make {
    request {
        //...
        method GET()
        url "/foo"

        // Each header is added in form `Header-Name' : 'Header-Value'`.
        // there are also some helper methods
        headers {
            header 'key': 'value'
            contentType(applicationJson())
        }

        //...
    }

    response {
        //...
        status 200
    }
}
```

YAML

```
request:
  ...
headers:
  foo: bar
  fooReq: baz
```

Java

```
org.springframework.cloud.contract.spec.Contract.make(c -> {
    c.request(r -> {
        // ...
        r.method(r.GET());
        r.url("/foo");

        // Each header is added in form `Header-Name` : `Header-Value`.
        // there are also some helper methods
        r.headers(h -> {
            h.header("key", "value");
            h.contentType(h.applicationJson());
        });

        // ...
    });

    c.response(r -> {
        // ...
        r.status(200);
    });
});
```

Kotlin

```
contract {
    request {
        // ...
        method = GET
        url = url("/foo")

        // Each header is added in form `Header-Name` : `Header-Value`.
        // there are also some helper variables
        headers {
            header("key", "value")
            contentType = APPLICATION_JSON
        }

        // ...
    }
    response {
        // ...
        status = OK
    }
}
```

request may contain additional request cookies, as the following example shows:

Groovy

```
org.springframework.cloud.contract.spec.Contract.make {
    request {
        //...
        method GET()
        url "/foo"

        // Each Cookies is added in form `Cookie-Key' : 'Cookie-Value'`.
        // there are also some helper methods
        cookies {
            cookie 'key': 'value'
            cookie('another_key', 'another_value')
        }

        //...
    }

    response {
        //...
        status 200
    }
}
```

YAML

```
request:
  ...
cookies:
  foo: bar
  fooReq: baz
```

Java

```
org.springframework.cloud.contract.spec.Contract.make(c -> {
    c.request(r -> {
        // ...
        r.method(r.GET());
        r.url("/foo");

        // Each Cookies is added in form ``Cookie-Key' : 'Cookie-Value'`.
        // there are also some helper methods
        r.cookies(ck -> {
            ck.cookie("key", "value");
            ck.cookie("another_key", "another_value");
        });

        // ...
    });

    c.response(r -> {
        // ...
        r.status(200);
    });
});
```

Kotlin

```
contract {
    request {
        // ...
        method = GET
        url = url("/foo")

        // Each Cookies is added in form ``Cookie-Key' : 'Cookie-Value'`.
        // there are also some helper methods
        cookies {
            cookie("key", "value")
            cookie("another_key", "another_value")
        }

        // ...
    }

    response {
        // ...
        status = code(200)
    }
}
```

request may contain a request body, as the following example shows:

Groovy

```
org.springframework.cloud.contract.spec.Contract.make {
    request {
        //...
        method GET()
        url "/foo"

        // Currently only JSON format of request body is supported.
        // Format will be determined from a header or body's content.
        body '''{ "login" : "john", "name": "John The Contract" }'''
    }

    response {
        //...
        status 200
    }
}
```

YAML

```
request:
  ...
body:
  foo: bar
```

Java

```
org.springframework.cloud.contract.spec.Contract.make(c -> {
    c.request(r -> {
        // ...
        r.method(r.GET());
        r.url("/foo");

        // Currently only JSON format of request body is supported.
        // Format will be determined from a header or body's content.
        r.body("{\"login\" : \"john\", \"name\": \"John The Contract\" }");
    });

    c.response(r -> {
        // ...
        r.status(200);
    });
});
```

Kotlin

```
contract {
    request {
        // ...
        method = GET
        url = url("/foo")

        // Currently only JSON format of request body is supported.
        // Format will be determined from a header or body's content.
        body = body("{ \"login\" : \"john\", \"name\": \"John The Contract\" }")
    }
    response {
        // ...
        status = OK
    }
}
```

`request` can contain multipart elements. To include multipart elements, use the `multipart` method/section, as the following examples show:

Groovy

```
org.springframework.cloud.contract.spec.Contract contractDsl =
org.springframework.cloud.contract.spec.Contract.make {
    request {
        method 'PUT'
        url '/multipart'
        headers {
            contentType('multipart/form-data;boundary=AaB03x')
        }
        multipart(
            // key (parameter name), value (parameter value) pair
            formParameter: $(c(regex('.+')), p("formParameterValue")),
            someBooleanParameter: $(c(regex(anyBoolean())), p('true')),
            // a named parameter (e.g. with `file` name) that represents file
with
            // `name` and `content`. You can also call `named("fileName",
"fileContent")`
            file: named(
                // name of the file
                name: $(c(regex(nonEmpty())), p('filename.csv')),
                // content of the file
                content: $(c(regex(nonEmpty())), p('file content')),
                // content type for the part
                contentType: $(c(regex(nonEmpty()))),
p('application/json'))
        )
    }
    response {
        status OK()
    }
}

org.springframework.cloud.contract.spec.Contract contractDsl =
org.springframework.cloud.contract.spec.Contract.make {
    request {
        method "PUT"
        url "/multipart"
        headers {
            contentType('multipart/form-data;boundary=AaB03x')
        }
        multipart(
            file: named(
                name: value(stub(regex('.+')), test('file')),
                content: value(stub(regex('.+')), test([100, 117, 100, 97]
as byte[]))
            )
        )
    }
    response {
        status 200
    }
}
```

```
}  
}
```

YAML

```
request:  
  method: PUT  
  url: /multipart  
  headers:  
    Content-Type: multipart/form-data;boundary=AaB03x  
  multipart:  
    params:  
      # key (parameter name), value (parameter value) pair  
      formParameter: "formParameterValue"  
      someBooleanParameter: true  
    named:  
      - paramName: file  
        fileName: filename.csv  
        fileContent: file content  
  matchers:  
    multipart:  
      params:  
        - key: formParameter  
          regex: ".+"  
        - key: someBooleanParameter  
          predefined: any_boolean  
      named:  
        - paramName: file  
          fileName:  
            predefined: non_empty  
          fileContent:  
            predefined: non_empty  
  response:  
    status: 200
```

Java

```
import java.util.Collection;  
import java.util.Collections;  
import java.util.HashMap;  
import java.util.Map;  
import java.util.function.Supplier;  
  
import org.springframework.cloud.contract.spec.Contract;  
import org.springframework.cloud.contract.spec.internal.DslProperty;  
import org.springframework.cloud.contract.spec.internal.Request;  
import org.springframework.cloud.contract.verifier.util.ContractVerifierUtil;  
  
class contract_multipart implements Supplier<Collection<Contract>> {
```



```

private static Map<String, DslProperty> namedProps(Request r) {
    Map<String, DslProperty> map = new HashMap<>();
    // name of the file
    map.put("name", r.$(r.c(r.regex(r.nonEmpty())), r.p("filename.csv")));
    // content of the file
    map.put("content", r.$(r.c(r.regex(r.nonEmpty())), r.p("file content")));
    // content type for the part
    map.put("contentType", r.$(r.c(r.regex(r.nonEmpty())),
r.p("application/json")));
    return map;
}

@Override
public Collection<Contract> get() {
    return Collections.singletonList(Contract.make(c -> {
        c.request(r -> {
            r.method("PUT");
            r.url("/multipart");
            r.headers(h -> {
                h.contentType("multipart/form-data;boundary=AaB03x");
            });
            r.multipart(ContractVerifierUtil.map()
                // key (parameter name), value (parameter value) pair
                .entry("formParameter", r.$(r.c(r.regex("\\.+\\")),
r.p("\\formParameterValue\\")))
                .entry("someBooleanParameter",
r.$(r.c(r.regex(r.anyBoolean())), r.p("true")))
                // a named parameter (e.g. with `file` name) that
represents file
                // with
                // `name` and `content`. You can also call
`named("fileName",
                // "fileContent")`
                .entry("file", r.named(namedProps(r))));
            });
        c.response(r -> {
            r.status(r.OK());
        });
    }));
}
}

```

Kotlin

```
import org.springframework.cloud.contract.spec.ContractDsl.Companion.contract

contract {
    request {
        method = PUT
        url = url("/multipart")
        multipart {
            field("formParameter", value(consumer(regex("\\.+\\")),
producer("\\formParameterValue\\")))
            field("someBooleanParameter", value(consumer(anyBoolean),
producer("true")))
            field("file",
                named(
                    // name of the file
                    value(consumer(regex(nonEmpty)), producer("filename.csv")),
                    // content of the file
                    value(consumer(regex(nonEmpty)), producer("file content")),
                    // content type for the part
                    value(consumer(regex(nonEmpty)), producer("application/json"))
                )
            )
        }
        headers {
            contentType = "multipart/form-data;boundary=AaB03x"
        }
    }
    response {
        status = OK
    }
}
```

In the preceding example, we defined parameters in either of two ways:

Coded DSL

- Directly, by using the map notation, where the value can be a dynamic property (such as `formParameter: $(consumer(...), producer(...))`).
- By using the `named(...)` method that lets you set a named parameter. A named parameter can set a `name` and `content`. You can call it either by using a method with two arguments, such as `named("fileName", "fileContent")`, or by using a map notation, such as `named(name: "fileName", content: "fileContent")`.

YAML

- The multipart parameters are set in the `multipart.params` section.
- The named parameters (the `fileName` and `fileContent` for a given parameter name) can be set in the `multipart.named` section. That section contains the `paramName` (the name of the parameter), `fileName` (the name of the file), `fileContent` (the content of the file) fields.

- The dynamic bits can be set in the `matchers.multipart` section.
 - For parameters, use the `params` section, which can accept `regex` or a `predefined` regular expression.
 - For named parameters, use the `named` section where you first define the parameter name with `paramName`. Then you can pass the parametrization of either `fileName` or `fileContent` in a `regex` or in a `predefined` regular expression.



For the `named(...)` section you always have to add a pair of `value(producer(...), consumer(...))` calls. Just setting DSL properties such as just `value(producer(...))` or just `file(...)` will not work. Check this [issue](#) for more information.

From the contract in the preceding example, the generated test and stub look as follows:

Test

```
// given:
MockMvcRequestSpecification request = given()
    .header("Content-Type", "multipart/form-data;boundary=AaB03x")
    .param("formParameter", "\"formParameterValue\"")
    .param("someBooleanParameter", "true")
    .multiPart("file", "filename.csv", "file content".getBytes());

// when:
ResponseOptions response = given().spec(request)
    .put("/multipart");

// then:
assertThat(response.statusCode()).isEqualTo(200);
```

Stub

```
    ...
  {
    "request" : {
      "url" : "/multipart",
      "method" : "PUT",
      "headers" : {
        "Content-Type" : {
          "matches" : "multipart/form-data;boundary=AaB03x.*"
        }
      },
      "bodyPatterns" : [ {
        "matches" : ".*--(.*)\\r?\\nContent-Disposition: form-data;
name=\\\"formParameter\\\"\\r?\\n(Content-Type: .*\\r?\\n)?(Content-Transfer-
Encoding: .*\\r?\\n)?(Content-Length: \\d+\\r?\\n)?\\r?\\n\\\".+\\\"\\r?\\n--.*"
      }, {
        "matches" : ".*--(.*)\\r?\\nContent-Disposition: form-data;
name=\\\"someBooleanParameter\\\"\\r?\\n(Content-Type: .*\\r?\\n)?(Content-Transfer-
Encoding: .*\\r?\\n)?(Content-Length: \\d+\\r?\\n)?\\r?\\n(true|false)\\r?\\n--
.*"
      }, {
        "matches" : ".*--(.*)\\r?\\nContent-Disposition: form-data; name=\\\"file\\\";
filename=\\\"[\\\\S\\\\s]+\\\"\\r?\\n(Content-Type: .*\\r?\\n)?(Content-Transfer-
Encoding: .*\\r?\\n)?(Content-Length: \\d+\\r?\\n)?\\r?\\n[\\\\S\\\\s]+\\r?\\n--
.*"
      } ]
    },
    "response" : {
      "status" : 200,
      "transformers" : [ "response-template", "foo-transformer" ]
    }
  }
  ...
}
```

3.2.3. HTTP Response

The response must contain an HTTP status code and may contain other information. The following code shows an example:

Groovy

```
org.springframework.cloud.contract.spec.Contract.make {
    request {
        //...
        method GET()
        url "/foo"
    }
    response {
        // Status code sent by the server
        // in response to request specified above.
        status OK()
    }
}
```

YAML

```
response:
  ...
  status: 200
```

Java

```
org.springframework.cloud.contract.spec.Contract.make(c -> {
    c.request(r -> {
        // ...
        r.method(r.GET());
        r.url("/foo");
    });
    c.response(r -> {
        // Status code sent by the server
        // in response to request specified above.
        r.status(r.OK());
    });
});
```

Kotlin

```
contract {
  request {
    // ...
    method = GET
    url =url("/foo")
  }
  response {
    // Status code sent by the server
    // in response to request specified above.
    status = OK
  }
}
```

Besides status, the response may contain headers, cookies, and a body, which are specified the same way as in the request (see [HTTP Request](#)).



In the Groovy DSL, you can reference the `org.springframework.cloud.contract.spec.internal.HttpStatus` methods to provide a meaningful status instead of a digit. For example, you can call `OK()` for a status `200` or `BAD_REQUEST()` for `400`.

3.2.4. Dynamic properties

The contract can contain some dynamic properties: timestamps, IDs, and so on. You do not want to force the consumers to stub their clocks to always return the same value of time so that it gets matched by the stub.

For the Groovy DSL, you can provide the dynamic parts in your contracts in two ways: pass them directly in the body or set them in a separate section called `bodyMatchers`.



Before 2.0.0, these were set by using `testMatchers` and `stubMatchers`. See the [migration guide](#) for more information.

For YAML, you can use only the `matchers` section.



Entries inside the `matchers` must reference existing elements of the payload. For more information, see [this issue](#).

Dynamic Properties inside the Body



This section is valid only for the Coded DSL (Groovy, Java, and so on). See the [Dynamic Properties in the Matchers Sections](#) section for YAML examples of a similar feature.

You can set the properties inside the body either with the `value` method or, if you use the Groovy

map notation, with `$()`. The following example shows how to set dynamic properties with the `value` method:

value

```
value(consumer(...), producer(...))
value(c(...), p(...))
value(stub(...), test(...))
value(client(...), server(...))
```

\$

```
$(consumer(...), producer(...))
$(c(...), p(...))
$(stub(...), test(...))
$(client(...), server(...))
```

Both approaches work equally well. The `stub` and `client` methods are aliases over the `consumer` method. Subsequent sections take a closer look at what you can do with those values.

Regular Expressions



This section is valid only for the Groovy DSL. See the [Dynamic Properties in the Matchers Sections](#) section for YAML examples of a similar feature.

You can use regular expressions to write your requests in the contract DSL. Doing so is particularly useful when you want to indicate that a given response should be provided for requests that follow a given pattern. Also, you can use regular expressions when you need to use patterns and not exact values both for your tests and your server-side tests.

Make sure that regex matches a whole region of a sequence, as, internally, `Pattern.matches()` is called. For instance, `abc` does not match `abc`, but `.abc` does. There are several additional [known limitations](#) as well.

The following example shows how to use regular expressions to write a request:

Groovy

```
org.springframework.cloud.contract.spec.Contract.make {
    request {
        method('GET')
        url $(consumer(~/\[/[0-9]{2}/), producer('/12'))
    }
    response {
        status OK()
        body(
            id: $(anyNumber()),
            surname: $(
                consumer('Kowalsky'),
                producer(regex('[a-zA-Z]+'))
            ),
            name: 'Jan',
            created: $(consumer('2014-02-02 12:23:43')),
            producer(execute('currentDate(it)')),
            correlationId: value(consumer('5d1f9fef-e0dc-4f3d-a7e4-
72d2220dd827'),
                producer(regex('[a-fA-F0-9]{8}-[a-fA-F0-9]{4}-[a-fA-F0-
9]{4}-[a-fA-F0-9]{4}-[a-fA-F0-9]{12}'))
            )
        )
        headers {
            header 'Content-Type': 'text/plain'
        }
    }
}
```

Java

```
org.springframework.cloud.contract.spec.Contract.make(c -> {
    c.request(r -> {
        r.method("GET");
        r.url(r.$(r.consumer(r.regex("\\[/[0-9]{2}")), r.producer("/12"))));
    });
    c.response(r -> {
        r.status(r.OK());
        r.body(ContractVerifierUtil.map().entry("id",
r.$(r.anyNumber())).entry("surname",
            r.$(r.consumer("Kowalsky"), r.producer(r.regex("[a-zA-Z]+"))));
        r.headers(h -> {
            h.header("Content-Type", "text/plain");
        });
    });
});
```

Kotlin

```
contract {
    request {
        method = method("GET")
        url = url(v(consumer(regex("\\/[0-9]{2}")), producer("/12")))
    }
    response {
        status = OK
        body(mapOf(
            "id" to v(anyNumber),
            "surname" to v(consumer("Kowalsky"), producer(regex("[a-zA-Z]+")))
        ))
        headers {
            header("Content-Type", "text/plain")
        }
    }
}
```

You can also provide only one side of the communication with a regular expression. If you do so, then the contract engine automatically provides the generated string that matches the provided regular expression. The following code shows an example for Groovy:

```
org.springframework.cloud.contract.spec.Contract.make {
    request {
        method 'PUT'
        url value(consumer(regex('/foo/[0-9]{5}')))
        body([
            requestElement: $(consumer(regex('[0-9]{5}'))
        ])
        headers {
            header('header',
$(consumer(regex('application/vnd\\.fraud\\.v1\\+json;.*'))))
        }
    }
    response {
        status OK()
        body([
            responseElement: $(producer(regex('[0-9]{7}'))
        ])
        headers {
            contentType("application/vnd.fraud.v1+json")
        }
    }
}
```

In the preceding example, the opposite side of the communication has the respective data generated for request and response.

Spring Cloud Contract comes with a series of predefined regular expressions that you can use in your contracts, as the following example shows:

```
public static RegexProperty onlyAlphaUnicode() {
    return new RegexProperty(ONLY_ALPHA_UNICODE).asString();
}

public static RegexProperty alphaNumeric() {
    return new RegexProperty(ALPHA_NUMERIC).asString();
}

public static RegexProperty number() {
    return new RegexProperty(NUMBER).asDouble();
}

public static RegexProperty positiveInt() {
    return new RegexProperty(POSITIVE_INT).asInteger();
}

public static RegexProperty anyBoolean() {
    return new RegexProperty(TRUE_OR_FALSE).asBooleanType();
}

public static RegexProperty anInteger() {
    return new RegexProperty(INTEGER).asInteger();
}

public static RegexProperty aDouble() {
    return new RegexProperty(DOUBLE).asDouble();
}

public static RegexProperty ipAddress() {
    return new RegexProperty(IP_ADDRESS).asString();
}

public static RegexProperty hostname() {
    return new RegexProperty(HOSTNAME_PATTERN).asString();
}

public static RegexProperty email() {
    return new RegexProperty(EMAIL).asString();
}

public static RegexProperty url() {
    return new RegexProperty(URL).asString();
}

public static RegexProperty httpsUrl() {
    return new RegexProperty(HTTPS_URL).asString();
}
```

```
public static RegexProperty uuid() {
    return new RegexProperty(UUID).asString();
}

public static RegexProperty uuid4() {
    return new RegexProperty(UUID4).asString();
}

public static RegexProperty isoDate() {
    return new RegexProperty(ANY_DATE).asString();
}

public static RegexProperty isoDateTime() {
    return new RegexProperty(ANY_DATE_TIME).asString();
}

public static RegexProperty isoTime() {
    return new RegexProperty(ANY_TIME).asString();
}

public static RegexProperty iso8601WithOffset() {
    return new RegexProperty(ISO8601_WITH_OFFSET).asString();
}

public static RegexProperty nonEmpty() {
    return new RegexProperty(NON_EMPTY).asString();
}

public static RegexProperty nonBlank() {
    return new RegexProperty(NON_BLANK).asString();
}
```

In your contract, you can use it as follows (example for the Groovy DSL):

```

Contract dslWithOptionalsInString = Contract.make {
  priority 1
  request {
    method POST()
    url '/users/password'
    headers {
      contentType(applicationJson())
    }
    body(
      email: $(consumer(optional(regex(email()))), producer('abc@abc.com')),
      callback_url: $(consumer(regex(hostname()))),
      producer('http://partners.com'))
    )
  }
  response {
    status 404
    headers {
      contentType(applicationJson())
    }
    body(
      code: value(consumer("123123"), producer(optional("123123"))),
      message: "User not found by email = [${value(producer(regex(email()))),
      consumer('not.existing@user.com'))}]"
    )
  }
}

```

To make matters even simpler, you can use a set of predefined objects that automatically assume that you want a regular expression to be passed. All of those methods start with the **any** prefix, as follows:

```
T anyAlphaUnicode();
T anyAlphaNumeric();
T anyNumber();
T anyInteger();
T anyPositiveInt();
T anyDouble();
T anyHex();
T aBoolean();
T anyIpAddress();
T anyHostname();
T anyEmail();
T anyUrl();
T anyHttpsUrl();
T anyUuid();
T anyDate();
T anyDateTime();
T anyTime();
T anyIso8601WithOffset();
T anyNonBlankString();
T anyNonEmptyString();
T anyOf(String... values);
```

The following example shows how you can reference those methods:

Groovy

```
Contract contractDsl = Contract.make {
    name "foo"
    label 'trigger_event'
    input {
        triggeredBy('toString()')
    }
    outputMessage {
        sentTo 'topic.rateablequote'
        body([
            alpha          : $(anyAlphaUnicode()),
            number         : $(anyNumber()),
            anInteger      : $(anyInteger()),
            positiveInt    : $(anyPositiveInt()),
            aDouble        : $(anyDouble()),
            aBoolean       : $(aBoolean()),
            ip             : $(anyIpAddress()),
            hostname       : $(anyHostname()),
            email          : $(anyEmail()),
            url            : $(anyUrl()),
            httpsUrl       : $(anyHttpsUrl()),
            uuid           : $(anyUuid()),
            date           : $(anyDate()),
            dateTime       : $(anyDateTime()),
            time           : $(anyTime()),
            iso8601WithOffset: $(anyIso8601WithOffset()),
            nonBlankString : $(anyNonBlankString()),
            nonEmptyString : $(anyNonEmptyString()),
            anyOf          : $(anyOf('foo', 'bar'))
        ])
    }
}
```

Kotlin

```
contract {
    name = "foo"
    label = "trigger_event"
    input {
        triggeredBy = "toString()"
    }
    outputMessage {
        sentTo = sentTo("topic.rateablequote")
        body(mapOf(
            "alpha" to v(anyAlphaUnicode),
            "number" to v(anyNumber),
            "anInteger" to v(anyInteger),
            "positiveInt" to v(anyPositiveInt),
            "aDouble" to v(anyDouble),
            "aBoolean" to v(aBoolean),
            "ip" to v(anyIpAddress),
            "hostname" to v(anyAlphaUnicode),
            "email" to v(anyEmail),
            "url" to v(anyUrl),
            "httpsUrl" to v(anyHttpsUrl),
            "uuid" to v(anyUuid),
            "date" to v(anyDate),
            "dateTime" to v(anyDateTime),
            "time" to v(anyTime),
            "iso8601WithOffset" to v(anyIso8601WithOffset),
            "nonBlankString" to v(anyNonBlankString),
            "nonEmptyString" to v(anyNonEmptyString),
            "anyOf" to v(anyOf('foo', 'bar'))
        ))
        headers {
            header("Content-Type", "text/plain")
        }
    }
}
```

Limitations



Due to certain limitations of the [Xeger](#) library that generates a string out of a regex, do not use the `$` and `^` signs in your regex if you rely on automatic generation. See [Issue 899](#).



Do not use a `LocalDate` instance as a value for `$` (for example, `$(consumer(LocalDate.now()))`). It causes a `java.lang.StackOverflowError`. Use `$(consumer(LocalDate.now().toString()))` instead. See [Issue 900](#).

Passing Optional Parameters



This section is valid only for Groovy DSL. See the [Dynamic Properties in the Matchers Sections](#) section for YAML examples of a similar feature.

You can provide optional parameters in your contract. However, you can provide optional parameters only for the following:

- The STUB side of the Request
- The TEST side of the Response

The following example shows how to provide optional parameters:

Groovy

```
org.springframework.cloud.contract.spec.Contract.make {
    priority 1
    name "optionals"
    request {
        method 'POST'
        url '/users/password'
        headers {
            contentType(applicationJson())
        }
        body(
            email: $(consumer(optional(regex(email()))),
producer('abc@abc.com')),
            callback_url: $(consumer(regex(hostname())),
producer('https://partners.com'))
        )
    }
    response {
        status 404
        headers {
            header 'Content-Type': 'application/json'
        }
        body(
            code: value(consumer("123123"), producer(optional("123123")))
        )
    }
}
```

Java

```
org.springframework.cloud.contract.spec.Contract.make(c -> {
    c.priority(1);
    c.name("optionals");
    c.request(r -> {
        r.method("POST");
        r.url("/users/password");
        r.headers(h -> {
            h.contentType(h.applicationJson());
        });
        r.body(ContractVerifierUtil.map()
            .entry("email", r.$(r.consumer(r.optional(r.regex(r.email())))),
r.producer("abc@abc.com")))
            .entry("callback_url",
                r.$(r.consumer(r.regex(r.hostname()))),
r.producer("https://partners.com"))));
    });
    c.response(r -> {
        r.status(404);
        r.headers(h -> {
            h.header("Content-Type", "application/json");
        });
        r.body(ContractVerifierUtil.map().entry("code",
            r.value(r.consumer("123123"), r.producer(r.optional("123123")))));
    });
});
```

Kotlin

```
contract { c ->
    priority = 1
    name = "optionals"
    request {
        method = POST
        url = url("/users/password")
        headers {
            contentType = APPLICATION_JSON
        }
        body = body(mapOf(
            "email" to v(consumer(optional(regex(email))),
producer("abc@abc.com")),
            "callback_url" to v(consumer(regex(hostname)),
producer("https://partners.com"))
        ))
    }
    response {
        status = NOT_FOUND
        headers {
            header("Content-Type", "application/json")
        }
        body(mapOf(
            "code" to value(consumer("123123"), producer(optional("123123")))
        ))
    }
}
```

By wrapping a part of the body with the `optional()` method, you create a regular expression that must be present 0 or more times.

If you use Spock, the following test would be generated from the previous example:

Groovy

```
package com.example

import com.jayway.jsonpath.DocumentContext
import com.jayway.jsonpath.JsonPath
import spock.lang.Specification
import io.restassured.module.mockmvc.specification.MockMvcRequestSpecification
import io.restassured.response.ResponseOptions

import static
org.springframework.cloud.contract.verifier.assertion.SpringCloudContractAssertion
s.assertThat
import static
org.springframework.cloud.contract.verifier.util.ContractVerifierUtil.*
import static com.toomuchcoding.jsonassert.JsonAssertion.assertThatJson
import static io.restassured.module.mockmvc.RestAssuredMockMvc.*

@SuppressWarnings("rawtypes")
class FooSpec extends Specification {

    \tdef validate_optionals() throws Exception {
    \t\tgiven:
    \t\t\tMockMvcRequestSpecification request = given()
    \t\t\t\t.header("Content-Type", "application/json")
    \t\t\t\t.body('{"email":"abc@abc.com","callback_url":"https://partners.com"}')
    \t\t\t\t

    \t\t\twhen:
    \t\t\t\tResponseOptions response = given().spec(request)
    \t\t\t\t\t.post("/users/password")

    \t\t\tthen:
    \t\t\t\tresponse.statusCode() == 404
    \t\t\t\tresponse.header("Content-Type") == 'application/json'

    \t\t\tand:
    \t\t\t\tDocumentContext parsedJson = JsonPath.parse(response.body.asString())
    \t\t\t\tassertThatJson(parsedJson).field("[code]").matches("(123123)?")
    \t\t\t\t

    \t\t\t}
    }
}
```

The following stub would also be generated:

```

    ...
  {
    "request" : {
      "url" : "/users/password",
      "method" : "POST",
      "bodyPatterns" : [ {
        "matchesJsonPath" : "$[?(@['email'] =~ /([a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-
]+\\\\.\\\\[a-zA-Z]{2,6})?/)]"
      }, {
        "matchesJsonPath" : "$[?(@['callback_url'] =~
/((http[s]?|ftp):\\\\/\\\\/)?(?:[^\":\\\\/\\\\s]+)(:[0-9]{1,5})?/)]"
      } ],
      "headers" : {
        "Content-Type" : {
          "equalTo" : "application/json"
        }
      }
    },
    "response" : {
      "status" : 404,
      "body" : "{\\\"code\\\":\\\"123123\\\",\\\"message\\\":\\\"User not found by email ==
[not.existing@user.com]\\\"}",
      "headers" : {
        "Content-Type" : "application/json"
      }
    },
    "priority" : 1
  }
  ...

```

Calling Custom Methods on the Server Side



This section is valid only for the Groovy DSL. See the [Dynamic Properties in the Matchers Sections](#) section for YAML examples of a similar feature.

You can define a method call that runs on the server side during the test. Such a method can be added to the class defined as `baseClassForTests` in the configuration. The following code shows an example of the contract portion of the test case:

Groovy

```
method GET()
```

Java

```
r.method(r.GET());
```

Kotlin

```
method = GET
```

The following code shows the base class portion of the test case:

```
abstract class BaseMockMvcSpec extends Specification {  
  
    def setup() {  
        RestAssuredMockMvc.standaloneSetup(new PairIdController())  
    }  
  
    void isProperCorrelationId(Integer correlationId) {  
        assert correlationId == 123456  
    }  
  
    void isEmpty(String value) {  
        assert value == null  
    }  
  
}
```



You cannot use both a `String` and `execute` to perform concatenation. For example, calling `header('Authorization', 'Bearer ' + execute('authToken()'))` leads to improper results. Instead, call `header('Authorization', execute('authToken()'))` and ensure that the `authToken()` method returns everything you need.

The type of the object read from the JSON can be one of the following, depending on the JSON path:

- `String`: If you point to a `String` value in the JSON.
- `JSONArray`: If you point to a `List` in the JSON.
- `Map`: If you point to a `Map` in the JSON.
- `Number`: If you point to `Integer`, `Double`, and other numeric type in the JSON.
- `Boolean`: If you point to a `Boolean` in the JSON.

In the request part of the contract, you can specify that the `body` should be taken from a method.



You must provide both the consumer and the producer side. The `execute` part is applied for the whole body, not for parts of it.

The following example shows how to read an object from JSON:

```
Contract contractDsl = Contract.make {
  request {
    method 'GET'
    url '/something'
    body(
      $(c('foo'), p(execute('hashCode()')))
    )
  }
  response {
    status OK()
  }
}
```

The preceding example results in calling the `hashCode()` method in the request body. It should resemble the following code:

```
// given:
MockMvcRequestSpecification request = given()
  .body(hashCode());

// when:
ResponseOptions response = given().spec(request)
  .get("/something");

// then:
assertThat(response.statusCode()).isEqualTo(200);
```

Referencing the Request from the Response

The best situation is to provide fixed values, but sometimes you need to reference a request in your response.

If you write contracts in the Groovy DSL, you can use the `fromRequest()` method, which lets you reference a bunch of elements from the HTTP request. You can use the following options:

- `fromRequest().url()`: Returns the request URL and query parameters.
- `fromRequest().query(String key)`: Returns the first query parameter with the given name.
- `fromRequest().query(String key, int index)`: Returns the nth query parameter with the given name.
- `fromRequest().path()`: Returns the full path.
- `fromRequest().path(int index)`: Returns the nth path element.

- `fromRequest().header(String key)`: Returns the first header with the given name.
- `fromRequest().header(String key, int index)`: Returns the nth header with the given name.
- `fromRequest().body()`: Returns the full request body.
- `fromRequest().body(String jsonPath)`: Returns the element from the request that matches the JSON Path.

If you use the YAML contract definition or the Java one, you have to use the [Handlebars](#) `{{{ }}` notation with custom Spring Cloud Contract functions to achieve this. In that case, you can use the following options:

- `{{{ request.url }}}`: Returns the request URL and query parameters.
- `{{{ request.query.key.[index] }}}`: Returns the nth query parameter with the given name. For example, for a key of `thing`, the first entry is `{{{ request.query.thing.[0] }}}`
- `{{{ request.path }}}`: Returns the full path.
- `{{{ request.path.[index] }}}`: Returns the nth path element. For example, the first entry is `'{{{ request.path.[0] }}}'`
- `{{{ request.headers.key }}}`: Returns the first header with the given name.
- `{{{ request.headers.key.[index] }}}`: Returns the nth header with the given name.
- `{{{ request.body }}}`: Returns the full request body.
- `{{{ jsonpath this 'your.json.path' }}}`: Returns the element from the request that matches the JSON Path. For example, for a JSON path of `$.here`, use `{{{ jsonpath this '$.here' }}}`

Consider the following contract:

Groovy

```
Contract contractDsl = Contract.make {
    request {
        method 'GET'
        url('/api/v1/xxxx') {
            queryParameters {
                parameter('foo', 'bar')
                parameter('foo', 'bar2')
            }
        }
        headers {
            header(authorization(), 'secret')
            header(authorization(), 'secret2')
        }
        body(foo: 'bar', baz: 5)
    }
    response {
        status OK()
        headers {
            header(authorization(), "foo ${fromRequest().header(authorization())}
bar")
        }
        body(
            url: fromRequest().url(),
            path: fromRequest().path(),
            pathIndex: fromRequest().path(1),
            param: fromRequest().query('foo'),
            paramIndex: fromRequest().query('foo', 1),
            authorization: fromRequest().header('Authorization'),
            authorization2: fromRequest().header('Authorization', 1),
            fullBody: fromRequest().body(),
            responseFoo: fromRequest().body('$.foo'),
            responseBaz: fromRequest().body('$.baz'),
            responseBaz2: "Bla bla ${fromRequest().body('$.foo')} bla bla",
            rawUrl: fromRequest().rawUrl(),
            rawPath: fromRequest().rawPath(),
            rawPathIndex: fromRequest().rawPath(1),
            rawParam: fromRequest().rawQuery('foo'),
            rawParamIndex: fromRequest().rawQuery('foo', 1),
            rawAuthorization: fromRequest().rawHeader('Authorization'),
            rawAuthorization2: fromRequest().rawHeader('Authorization', 1),
            rawResponseFoo: fromRequest().rawBody('$.foo'),
            rawResponseBaz: fromRequest().rawBody('$.baz'),
            rawResponseBaz2: "Bla bla ${fromRequest().rawBody('$.foo')} bla
bla"
        )
    }
}
Contract contractDsl = Contract.make {
```

```

request {
  method 'GET'
  url('/api/v1/xxxx') {
    queryParameters {
      parameter('foo', 'bar')
      parameter('foo', 'bar2')
    }
  }
  headers {
    header(authorization(), 'secret')
    header(authorization(), 'secret2')
  }
  body(foo: "bar", baz: 5)
}
response {
  status OK()
  headers {
    contentType(applicationJson())
  }
  body(''
    {
      "responseFoo": "{{{ jsonPath request.body '$.foo' }}}",
      "responseBaz": {{{ jsonPath request.body '$.baz' }}}},
      "responseBaz2": "Bla bla {{{ jsonPath request.body '$.foo' }}}
bla bla"
    }
    ''.toString())
  }
}

```

YAML

```
request:
  method: GET
  url: /api/v1/xxxx
  queryParameters:
    foo:
      - bar
      - bar2
  headers:
    Authorization:
      - secret
      - secret2
  body:
    foo: bar
    baz: 5
response:
  status: 200
  headers:
    Authorization: "foo {{{ request.headers.Authorization.0 }}} bar"
  body:
    url: "{{{ request.url }}"
    path: "{{{ request.path }}"
    pathIndex: "{{{ request.path.1 }}"
    param: "{{{ request.query.foo }}"
    paramIndex: "{{{ request.query.foo.1 }}"
    authorization: "{{{ request.headers.Authorization.0 }}"
    authorization2: "{{{ request.headers.Authorization.1 }}"
    fullBody: "{{{ request.body }}"
    responseFoo: "{{{ jsonpath this '$.foo' }}"
    responseBaz: "{{{ jsonpath this '$.baz' }}"
    responseBaz2: "Bla bla {{{ jsonpath this '$.foo' }}} bla bla"
```

Java

```
package contracts.beer.rest;

import java.util.function.Supplier;

import org.springframework.cloud.contract.spec.Contract;

import static
org.springframework.cloud.contract.verifier.util.ContractVerifierUtil.map;

class shouldReturnStatsForAUser implements Supplier<Contract> {

    @Override
    public Contract get() {
        return Contract.make(c -> {
            c.request(r -> {
                r.method("POST");
                r.url("/stats");
                r.body(map().entry("name", r.anyAlphaUnicode()));
                r.headers(h -> {
                    h.contentType(h.applicationJson());
                });
            });
            c.response(r -> {
                r.status(r.OK());
                r.body(map()
                    .entry("text",
                        "Dear {{{jsonPath request.body '$.name'}}} thanks
for your interested in drinking beer")
                    .entry("quantity", r.$(r.c(5), r.p(r.anyNumber()))));
                r.headers(h -> {
                    h.contentType(h.applicationJson());
                });
            });
        });
    }
}
```

Kotlin

```
package contracts.beer.rest

import org.springframework.cloud.contract.spec.ContractDsl.Companion.contract

contract {
    request {
        method = method("POST")
        url = url("/stats")
        body(mapOf(
            "name" to anyAlphaUnicode
        ))
        headers {
            contentType = APPLICATION_JSON
        }
    }
    response {
        status = OK
        body(mapOf(
            "text" to "Don't worry ${fromRequest().body("$.name")} thanks for your
interested in drinking beer",
            "quantity" to v(c(5), p(anyNumber))
        ))
        headers {
            contentType = fromRequest().header(CONTENT_TYPE)
        }
    }
}
```

Running a JUnit test generation leads to a test that resembles the following example:

```

// given:
MockMvcRequestSpecification request = given()
    .header("Authorization", "secret")
    .header("Authorization", "secret2")
    .body("{\"foo\":\"bar\",\"baz\":5}");

// when:
ResponseOptions response = given().spec(request)
    .queryParams("foo","bar")
    .queryParams("foo","bar2")
    .get("/api/v1/xxxx");

// then:
assertThat(response.statusCode()).isEqualTo(200);
assertThat(response.header("Authorization")).isEqualTo("foo secret bar");
// and:
DocumentContext parsedJson = JsonPath.parse(response.getBody().asString());
assertThatJson(parsedJson).field("[ 'fullBody' ]").isEqualTo("{\"foo\":\"bar\",\"baz\":5}");
assertThatJson(parsedJson).field("[ 'authorization' ]").isEqualTo("secret");
assertThatJson(parsedJson).field("[ 'authorization2' ]").isEqualTo("secret2");
assertThatJson(parsedJson).field("[ 'path' ]").isEqualTo("/api/v1/xxxx");
assertThatJson(parsedJson).field("[ 'param' ]").isEqualTo("bar");
assertThatJson(parsedJson).field("[ 'paramIndex' ]").isEqualTo("bar2");
assertThatJson(parsedJson).field("[ 'pathIndex' ]").isEqualTo("v1");
assertThatJson(parsedJson).field("[ 'responseBaz' ]").isEqualTo(5);
assertThatJson(parsedJson).field("[ 'responseFoo' ]").isEqualTo("bar");
assertThatJson(parsedJson).field("[ 'url' ]").isEqualTo("/api/v1/xxxx?foo=bar&foo=bar2");
assertThatJson(parsedJson).field("[ 'responseBaz2' ]").isEqualTo("Bla bla bar bla bla");

```

As you can see, elements from the request have been properly referenced in the response.

The generated WireMock stub should resemble the following example:

```

{
  "request" : {
    "urlPath" : "/api/v1/xxxx",
    "method" : "POST",
    "headers" : {
      "Authorization" : {
        "equalTo" : "secret2"
      }
    },
    "queryParameters" : {
      "foo" : {
        "equalTo" : "bar2"
      }
    },
    "bodyPatterns" : [ {
      "matchesJsonPath" : "$[?(@.['baz'] == 5)]"
    }, {
      "matchesJsonPath" : "$[?(@.['foo'] == 'bar')]"
    } ]
  },
  "response" : {
    "status" : 200,
    "body" :
    "{\\"authorization\\":\\"{{{request.headers.Authorization.[0]}}}\\",\\"path\\":\\"{{{request.path}}}\\",\\"responseBaz\\":{{{jsonpath this '$.baz'}}},\\"param\\":\\"{{{request.query.foo.[0]}}}\\",\\"pathIndex\\":\\"{{{request.path.[1]}}}\\",\\"responseBaz2\\":\\"Bla bla {{{jsonpath this '$.foo'}}} bla bla\\",\\"responseFoo\\":\\"{{{jsonpath this '$.foo'}}}\\",\\"authorization2\\":\\"{{{request.headers.Authorization.[1]}}}\\",\\"full Body\\":\\"{{{escapejsonbody}}}\\",\\"url\\":\\"{{{request.url}}}\\",\\"paramIndex\\":\\"{{{request.query.foo.[1]}}}\\""},
    "headers" : {
      "Authorization" : "{{{{request.headers.Authorization.[0]}}};foo"
    },
    "transformers" : [ "response-template" ]
  }
}

```

Sending a request such as the one presented in the **request** part of the contract results in sending the following response body:


```

{
  "url" : "/api/v1/xxxx?foo=bar&foo=bar2",
  "path" : "/api/v1/xxxx",
  "pathIndex" : "v1",
  "param" : "bar",
  "paramIndex" : "bar2",
  "authorization" : "secret",
  "authorization2" : "secret2",
  "fullBody" : "{\"foo\":\"bar\",\"baz\":5}",
  "responseFoo" : "bar",
  "responseBaz" : 5,
  "responseBaz2" : "Bla bla bar bla bla"
}

```



This feature works only with WireMock versions greater than or equal to 2.5.1. The Spring Cloud Contract Verifier uses WireMock's `response-template` response transformer. It uses Handlebars to convert the Mustache `{{{ }}}` templates into proper values. Additionally, it registers two helper functions:

- `escapejsonbody`: Escapes the request body in a format that can be embedded in JSON.
- `jsonpath`: For a given parameter, finds an object in the request body.

Dynamic Properties in the Matchers Sections

If you work with [Pact](#), the following discussion may seem familiar. Quite a few users are used to having a separation between the body and setting the dynamic parts of a contract.

You can use the `bodyMatchers` section for two reasons:

- Define the dynamic values that should end up in a stub. You can set it in the `request` part of your contract.
- Verify the result of your test. This section is present in the `response` or `outputMessage` side of the contract.

Currently, Spring Cloud Contract Verifier supports only JSON path-based matchers with the following matching possibilities:

Coded DSL

For the stubs (in tests on the consumer's side):

- `byEquality()`: The value taken from the consumer's request in the provided JSON path must be equal to the value provided in the contract.
- `byRegex(...)`: The value taken from the consumer's request in the provided JSON path must match the regex. You can also pass the type of the expected matched value (for example, `asString()`, `asLong()`, and so on).

- `byDate()`: The value taken from the consumer's request in the provided JSON path must match the regex for an ISO Date value.
- `byTimestamp()`: The value taken from the consumer's request in the provided JSON path must match the regex for an ISO DateTime value.
- `byTime()`: The value taken from the consumer's request in the provided JSON path must match the regex for an ISO Time value.

For the verification (in generated tests on the Producer's side):

- `byEquality()`: The value taken from the producer's response in the provided JSON path must be equal to the provided value in the contract.
- `byRegex(...)`: The value taken from the producer's response in the provided JSON path must match the regex.
- `byDate()`: The value taken from the producer's response in the provided JSON path must match the regex for an ISO Date value.
- `byTimestamp()`: The value taken from the producer's response in the provided JSON path must match the regex for an ISO DateTime value.
- `byTime()`: The value taken from the producer's response in the provided JSON path must match the regex for an ISO Time value.
- `byType()`: The value taken from the producer's response in the provided JSON path needs to be of the same type as the type defined in the body of the response in the contract. `byType` can take a closure, in which you can set `minOccurrence` and `maxOccurrence`. For the request side, you should use the closure to assert size of the collection. That way, you can assert the size of the flattened collection. To check the size of an unflattened collection, use a custom method with the `byCommand(...)` `testMatcher`.
- `byCommand(...)`: The value taken from the producer's response in the provided JSON path is passed as an input to the custom method that you provide. For example, `byCommand('thing($it)')` results in calling a `thing` method to which the value matching the JSON Path gets passed. The type of the object read from the JSON can be one of the following, depending on the JSON path:
 - `String`: If you point to a `String` value.
 - `JSONArray`: If you point to a `List`.
 - `Map`: If you point to a `Map`.
 - `Number`: If you point to `Integer`, `Double`, or another kind of number.
 - `Boolean`: If you point to a `Boolean`.
- `byNull()`: The value taken from the response in the provided JSON path must be null.

YAML



See the Groovy section for a detailed explanation of what the types mean.

For YAML, the structure of a matcher resembles the following example:

```
- path: $.thing1
  type: by_regex
  value: thing2
  regexType: as_string
```

Alternatively, if you want to use one of the predefined regular expressions [`only_alpha_unicode`, `number`, `any_boolean`, `ip_address`, `hostname`, `email`, `url`, `uuid`, `iso_date`, `iso_date_time`, `iso_time`, `iso_8601_with_offset`, `non_empty`, `non_blank`], you can use something similar to the following example:

```
- path: $.thing1
  type: by_regex
  predefined: only_alpha_unicode
```

The following list shows the allowed list of `type` values:

- For `stubMatchers`:
 - `by_equality`
 - `by_regex`
 - `by_date`
 - `by_timestamp`
 - `by_time`
 - `by_type`
 - Two additional fields (`minOccurrence` and `maxOccurrence`) are accepted.
- For `testMatchers`:
 - `by_equality`
 - `by_regex`
 - `by_date`
 - `by_timestamp`
 - `by_time`
 - `by_type`
 - Two additional fields (`minOccurrence` and `maxOccurrence`) are accepted.
 - `by_command`
 - `by_null`

You can also define which type the regular expression corresponds to in the `regexType` field. The following list shows the allowed regular expression types:

- `as_integer`
- `as_double`

- `as_float`
- `as_long`
- `as_short`
- `as_boolean`
- `as_string`

Consider the following example:

Groovy

```
Contract contractDsl = Contract.make {
  request {
    method 'GET'
    urlPath '/get'
    body([
      duck           : 123,
      alpha          : 'abc',
      number         : 123,
      aBoolean       : true,
      date           : '2017-01-01',
      dateTime       : '2017-01-01T01:23:45',
      time           : '01:02:34',
      valueWithoutAMatcher: 'foo',
      valueWithTypeMatch : 'string',
      key            : [
        'complex.key': 'foo'
      ]
    ])
    bodyMatchers {
      jsonPath('$.duck', byRegex("[0-9]{3}").asInteger())
      jsonPath('$.duck', byEquality())
      jsonPath('$.alpha', byRegex(onlyAlphaUnicode()).asString())
      jsonPath('$.alpha', byEquality())
      jsonPath('$.number', byRegex(number()).asInteger())
      jsonPath('$.aBoolean', byRegex(anyBoolean()).asBooleanType())
      jsonPath('$.date', byDate())
      jsonPath('$.dateTime', byTimestamp())
      jsonPath('$.time', byTime())
      jsonPath("\$.['key'].['complex.key']", byEquality())
    }
    headers {
      contentType(applicationJson())
    }
  }
  response {
    status OK()
    body([
      duck           : 123,
      alpha          : 'abc',
      number         : 123,
      positiveInteger : 1234567890,
      negativeInteger : -1234567890,
      positiveDecimalNumber: 123.4567890,
      negativeDecimalNumber: -123.4567890,
      aBoolean       : true,
      date           : '2017-01-01',
      dateTime       : '2017-01-01T01:23:45',
      time           : "01:02:34",
```

```

        valueWithoutAMatcher : 'foo',
        valueWithTypeMatch   : 'string',
        valueWithMin         : [
            1, 2, 3
        ],
        valueWithMax         : [
            1, 2, 3
        ],
        valueWithMinMax      : [
            1, 2, 3
        ],
        valueWithMinEmpty    : [],
        valueWithMaxEmpty    : [],
        key                   : [
            'complex.key': 'foo'
        ],
        nullValue             : null
    ])
    bodyMatchers {
        // asserts the jsonpath value against manual regex
        jsonPath('$.duck', byRegex("[0-9]{3}").asInteger())
        // asserts the jsonpath value against the provided value
        jsonPath('$.duck', byEquality())
        // asserts the jsonpath value against some default regex
        jsonPath('$.alpha', byRegex(onlyAlphaUnicode()).asString())
        jsonPath('$.alpha', byEquality())
        jsonPath('$.number', byRegex(number()).asInteger())
        jsonPath('$.positiveInteger', byRegex(anInteger()).asInteger())
        jsonPath('$.negativeInteger', byRegex(anInteger()).asInteger())
        jsonPath('$.positiveDecimalNumber', byRegex(aDouble()).asDouble())
        jsonPath('$.negativeDecimalNumber', byRegex(aDouble()).asDouble())
        jsonPath('$.aBoolean', byRegex(anyBoolean()).asBooleanType())
        // asserts vs inbuilt time related regex
        jsonPath('$.date', byDate())
        jsonPath('$.dateTime', byTimestamp())
        jsonPath('$.time', byTime())
        // asserts that the resulting type is the same as in response body
        jsonPath('$.valueWithTypeMatch', byType())
        jsonPath('$.valueWithMin', byType {
            // results in verification of size of array (min 1)
            minOccurrence(1)
        })
        jsonPath('$.valueWithMax', byType {
            // results in verification of size of array (max 3)
            maxOccurrence(3)
        })
        jsonPath('$.valueWithMinMax', byType {
            // results in verification of size of array (min 1 & max 3)
            minOccurrence(1)
            maxOccurrence(3)
        })
    }
}

```

```

    jsonPath('$.valueWithMinEmpty', byType {
        // results in verification of size of array (min 0)
        minOccurrence(0)
    })
    jsonPath('$.valueWithMaxEmpty', byType {
        // results in verification of size of array (max 0)
        maxOccurrence(0)
    })
    // will execute a method `assertThatValueIsANumber`
    jsonPath('$.duck', byCommand('assertThatValueIsANumber($it)'))
    jsonPath("\$.['key']['complex.key']", byEquality())
    jsonPath('$.nullValue', byNull())
}
headers {
    contentType(applicationJson())
    header('Some-Header', $(c('someValue'), p(regex('[a-zA-Z]{9}'))))
}
}
}

```

YAML

```

request:
  method: GET
  urlPath: /get/1
  headers:
    Content-Type: application/json
  cookies:
    foo: 2
    bar: 3
  queryParameters:
    limit: 10
    offset: 20
    filter: 'email'
    sort: name
    search: 55
    age: 99
    name: John.Doe
    email: 'bob@email.com'
  body:
    duck: 123
    alpha: "abc"
    number: 123
    aBoolean: true
    date: "2017-01-01"
    dateTime: "2017-01-01T01:23:45"
    time: "01:02:34"
    valueWithoutAMatcher: "foo"
    valueWithTypeMatch: "string"
  key:
    "complex.key": 'foo'

```

```
nullValue: null
valueWithMin:
  - 1
  - 2
  - 3
valueWithMax:
  - 1
  - 2
  - 3
valueWithMinMax:
  - 1
  - 2
  - 3
valueWithMinEmpty: []
valueWithMaxEmpty: []
matchers:
  url:
    regex: /get/[0-9]
    # predefined:
    # execute a method
    #command: 'equals($it)'
  queryParameters:
    - key: limit
      type: equal_to
      value: 20
    - key: offset
      type: containing
      value: 20
    - key: sort
      type: equal_to
      value: name
    - key: search
      type: not_matching
      value: '^[[0-9]]{2}$'
    - key: age
      type: not_matching
      value: '^\\w*$'
    - key: name
      type: matching
      value: 'John.*'
    - key: hello
      type: absent
  cookies:
    - key: foo
      regex: '[0-9]'
    - key: bar
      command: 'equals($it)'
  headers:
    - key: Content-Type
      regex: "application/json.*"
  body:
```


- path: \$.duck
type: by_regex
value: "[0-9]{3}"
- path: \$.duck
type: by_equality
- path: \$.alpha
type: by_regex
predefined: only_alpha_unicode
- path: \$.alpha
type: by_equality
- path: \$.number
type: by_regex
predefined: number
- path: \$.aBoolean
type: by_regex
predefined: any_boolean
- path: \$.date
type: by_date
- path: \$.dateTime
type: by_timestamp
- path: \$.time
type: by_time
- path: "\$.['key'].['complex.key']"
type: by_equality
- path: \$.nullvalue
type: by_null
- path: \$.valueWithMin
type: by_type
minOccurrence: 1
- path: \$.valueWithMax
type: by_type
maxOccurrence: 3
- path: \$.valueWithMinMax
type: by_type
minOccurrence: 1
maxOccurrence: 3

response:

status: 200

cookies:

foo: 1

bar: 2

body:

duck: 123

alpha: "abc"

number: 123

aBoolean: true

date: "2017-01-01"

dateTime: "2017-01-01T01:23:45"

time: "01:02:34"

valueWithoutAMatcher: "foo"

valueWithTypeMatch: "string"

```
valueWithMin:
  - 1
  - 2
  - 3
valueWithMax:
  - 1
  - 2
  - 3
valueWithMinMax:
  - 1
  - 2
  - 3
valueWithMinEmpty: []
valueWithMaxEmpty: []
key:
  'complex.key': 'foo'
nulValue: null
matchers:
  headers:
    - key: Content-Type
      regex: "application/json.*"
  cookies:
    - key: foo
      regex: '[0-9]'
    - key: bar
      command: 'equals($it)'
  body:
    - path: $.duck
      type: by_regex
      value: "[0-9]{3}"
    - path: $.duck
      type: by_equality
    - path: $.alpha
      type: by_regex
      predefined: only_alpha_unicode
    - path: $.alpha
      type: by_equality
    - path: $.number
      type: by_regex
      predefined: number
    - path: $.aBoolean
      type: by_regex
      predefined: any_boolean
    - path: $.date
      type: by_date
    - path: $.dateTime
      type: by_timestamp
    - path: $.time
      type: by_time
    - path: $.valueWithTypeMatch
      type: by_type
```

```

- path: $.valueWithMin
  type: by_type
  minOccurrence: 1
- path: $.valueWithMax
  type: by_type
  maxOccurrence: 3
- path: $.valueWithMinMax
  type: by_type
  minOccurrence: 1
  maxOccurrence: 3
- path: $.valueWithMinEmpty
  type: by_type
  minOccurrence: 0
- path: $.valueWithMaxEmpty
  type: by_type
  maxOccurrence: 0
- path: $.duck
  type: by_command
  value: assertThatValueIsANumber($it)
- path: $.nullValue
  type: by_null
  value: null
headers:
  Content-Type: application/json

```

In the preceding example, you can see the dynamic portions of the contract in the `matchers` sections. For the request part, you can see that, for all fields but `valueWithoutAMatcher`, the values of the regular expressions that the stub should contain are explicitly set. For `valueWithoutAMatcher`, the verification takes place in the same way as without the use of matchers. In that case, the test performs an equality check.

For the response side in the `bodyMatchers` section, we define the dynamic parts in a similar manner. The only difference is that the `byType` matchers are also present. The verifier engine checks four fields to verify whether the response from the test has a value for which the JSON path matches the given field, is of the same type as the one defined in the response body, and passes the following check (based on the method being called):

- For `$.valueWithTypeMatch`, the engine checks whether the type is the same.
- For `$.valueWithMin`, the engine checks the type and asserts whether the size is greater than or equal to the minimum occurrence.
- For `$.valueWithMax`, the engine checks the type and asserts whether the size is smaller than or equal to the maximum occurrence.
- For `$.valueWithMinMax`, the engine checks the type and asserts whether the size is between the minimum and maximum occurrence.

The resulting test resembles the following example (note that an `and` section separates the autogenerated assertions and the assertion from matchers):

```

// given:
MockMvcRequestSpecification request = given()
    .header("Content-Type", "application/json")

    .body("{\"duck\":123,\"alpha\":\"abc\",\"number\":123,\"aBoolean\":true,\"date\":\"2017-01-01\",\"dateTime\":\"2017-01-01T01:23:45\",\"time\":\"01:02:34\",\"valueWithoutAMatcher\":\"foo\",\"valueWithTypeMatch\":\"string\",\"key\":{\"complex.key\":\"foo\"}}");

// when:
ResponseOptions response = given().spec(request)
    .get("/get");

// then:
assertThat(response.statusCode()).isEqualTo(200);
assertThat(response.header("Content-Type")).matches("application/json.*");
// and:
DocumentContext parsedJson = JsonPath.parse(response.getBody().asString());
assertThatJson(parsedJson).field("'valueWithoutAMatcher']").isEqualTo("foo");
// and:
assertThat(parsedJson.read("$.duck", String.class)).matches("[0-9]{3}");
assertThat(parsedJson.read("$.duck", Integer.class)).isEqualTo(123);
assertThat(parsedJson.read("$.alpha", String.class)).matches("[\\p{L}]*");
assertThat(parsedJson.read("$.alpha", String.class)).isEqualTo("abc");
assertThat(parsedJson.read("$.number", String.class)).matches("-?(\\d*\\.\\d+|\\d+)");
assertThat(parsedJson.read("$.aBoolean", String.class)).matches("(true|false)");
assertThat(parsedJson.read("$.date", String.class)).matches("(\\d\\d\\d\\d\\d)-(0[1-9]|1[012])-(0[1-9]|[12][0-9]|3[01])");
assertThat(parsedJson.read("$.dateTime", String.class)).matches("([0-9]{4})-(1[0-2]|0[1-9])-(3[01]|0[1-9]|[12][0-9])T(2[0-3]|[01][0-9]):([0-5][0-9]):([0-5][0-9])");
assertThat(parsedJson.read("$.time", String.class)).matches("(2[0-3]|[01][0-9]):([0-5][0-9]):([0-5][0-9])");
assertThat((Object)
    parsedJson.read("$.valueWithTypeMatch")).isInstanceOf(java.lang.String.class);
assertThat((Object)
    parsedJson.read("$.valueWithMin")).isInstanceOf(java.util.List.class);
assertThat((java.lang.Iterable) parsedJson.read("$.valueWithMin",
    java.util.Collection.class)).as("$.valueWithMin").hasSizeGreaterThanOrEqualTo(1);
assertThat((Object)
    parsedJson.read("$.valueWithMax")).isInstanceOf(java.util.List.class);
assertThat((java.lang.Iterable) parsedJson.read("$.valueWithMax",
    java.util.Collection.class)).as("$.valueWithMax").hasSizeLessThanOrEqualTo(3);
assertThat((Object)
    parsedJson.read("$.valueWithMinMax")).isInstanceOf(java.util.List.class);
assertThat((java.lang.Iterable) parsedJson.read("$.valueWithMinMax",
    java.util.Collection.class)).as("$.valueWithMinMax").hasSizeBetween(1, 3);
assertThat((Object)
    parsedJson.read("$.valueWithMinEmpty")).isInstanceOf(java.util.List.class);
assertThat((java.lang.Iterable) parsedJson.read("$.valueWithMinEmpty",
    java.util.Collection.class)).as("$.valueWithMinEmpty").hasSizeGreaterThanOrEqualTo(0);

```

```

assertThat((Object)
parsedJson.read("$.valueWithMaxEmpty")).assertInstanceOf(java.util.List.class);
assertThat((java.lang.Iterable) parsedJson.read("$.valueWithMaxEmpty",
java.util.Collection.class)).as("$.valueWithMaxEmpty").hasSizeLessThanOrEqualTo(0);
assertThatValueIsANumber(parsedJson.read("$.duck"));
assertThat(parsedJson.read("$.['key'].['complex.key']",
String.class)).isEqualTo("foo");

```



Notice that, for the `byCommand` method, the example calls the `assertThatValueIsANumber`. This method must be defined in the test base class or be statically imported to your tests. Notice that the `byCommand` call was converted to `assertThatValueIsANumber(parsedJson.read("$.duck"))`; That means that the engine took the method name and passed the proper JSON path as a parameter to it.

The resulting WireMock stub is in the following example:

```

'''
{
  "request" : {
    "urlPath" : "/get",
    "method" : "POST",
    "headers" : {
      "Content-Type" : {
        "matches" : "application/json.*"
      }
    },
    "bodyPatterns" : [ {
      "matchesJsonPath" : "$.['list'].['some'].['nested'][?(@.['anothervalue'] == 4)]"
    }, {
      "matchesJsonPath" : "$[?(@.['valueWithoutAMatcher'] == 'foo')]"
    }, {
      "matchesJsonPath" : "$[?(@.['valueWithTypeMatch'] == 'string')]"
    }, {
      "matchesJsonPath" : "$.['list'].['someother'].['nested'][?(@.['json'] == 'with
value')]"
    }, {
      "matchesJsonPath" : "$.['list'].['someother'].['nested'][?(@.['anothervalue'] ==
4)]"
    }, {
      "matchesJsonPath" : "$[?(@.duck =~ /[0-9]{3})/]"
    }, {
      "matchesJsonPath" : "$[?(@.duck == 123)]"
    }, {
      "matchesJsonPath" : "$[?(@.alpha =~ /([\\p{L}]+)/]"
    }, {
      "matchesJsonPath" : "$[?(@.alpha == 'abc')]"
    }, {
      "matchesJsonPath" : "$[?(@.number =~ /(-?(\\d*\\.\\d+|\\d+))/]"
    }, {

```

```

    "matchesJsonPath" : "$[?(@.aBoolean =~ /((true|false))/)]"
  }, {
    "matchesJsonPath" : "$[?(@.date =~ /((\\d\\d\\d\\d\\d\\d\\d\\d\\d\\d)-(0[1-9]|1[012])-(0[1-9]|1[12][0-9]|3[01]))/)]"
  }, {
    "matchesJsonPath" : "$[?(@.dateTime =~ /((([0-9]{4})-(1[0-2]|0[1-9])-(3[01]|0[1-9]|1[12][0-9])T(2[0-3]|[01][0-9]):([0-5][0-9]):([0-5][0-9]))/)]"
  }, {
    "matchesJsonPath" : "$[?(@.time =~ /((2[0-3]|[01][0-9]):([0-5][0-9]):([0-5][0-9]))/)]"
  }, {
    "matchesJsonPath" : "$.list.some.nested[?(@.json =~ /(.*)/)]"
  }, {
    "matchesJsonPath" : "$[?(@.valueWithMin.size() >= 1)]"
  }, {
    "matchesJsonPath" : "$[?(@.valueWithMax.size() <= 3)]"
  }, {
    "matchesJsonPath" : "$[?(@.valueWithMinMax.size() >= 1 && @.valueWithMinMax.size() <= 3)]"
  }, {
    "matchesJsonPath" : "$[?(@.valueWithOccurrence.size() >= 4 && @.valueWithOccurrence.size() <= 4)]"
  } ]
},
"response" : {
  "status" : 200,
  "body" :
"{\\\"duck\\\":123,\\\"alpha\\\":\\\"abc\\\",\\\"number\\\":123,\\\"aBoolean\\\":true,\\\"date\\\":\\\"2017-01-01\\\",\\\"dateTime\\\":\\\"2017-01-01T01:23:45\\\",\\\"time\\\":\\\"01:02:34\\\",\\\"valueWithoutAMatcher\\\":\\\"foo\\\",\\\"valueWithTypeMatch\\\":\\\"string\\\",\\\"valueWithMin\\\":[1,2,3],\\\"valueWithMax\\\":[1,2,3],\\\"valueWithMinMax\\\":[1,2,3],\\\"valueWithOccurrence\\\":[1,2,3,4]}",
  "headers" : {
    "Content-Type" : "application/json"
  },
  "transformers" : [ "response-template", "spring-cloud-contract" ]
}
}
...

```



If you use a **matcher**, the part of the request and response that the **matcher** addresses with the JSON Path gets removed from the assertion. In the case of verifying a collection, you must create matchers for **all** the elements of the collection.

Consider the following example:

```

Contract.make {
  request {
    method 'GET'
    url("/foo")
  }
  response {
    status OK()
    body(events: [[
      [
        operation      : 'EXPORT',
        eventId        : '16f1ed75-0bcc-4f0d-a04d-
3121798faf99',
        status         : 'OK'
      ], [
        operation      : 'INPUT_PROCESSING',
        eventId        : '3bb4ac82-6652-462f-b6d1-
75e424a0024a',
        status         : 'OK'
      ]
    ]
    )
    bodyMatchers {
      jsonPath('$.events[0].operation', byRegex('.+'))
      jsonPath('$.events[0].eventId', byRegex('^([a-fA-F0-9]{8}-[a-fA-F0-
9]{4}-[a-fA-F0-9]{4}-[a-fA-F0-9]{4}-[a-fA-F0-9]{12})$'))
      jsonPath('$.events[0].status', byRegex('.+'))
    }
  }
}

```

The preceding code leads to creating the following test (the code block shows only the assertion section):

```

and:
    DocumentContext parsedJson = JsonPath.parse(response.body.asString())

assertThatJson(parsedJson).array(["'events'"]).contains(["'eventId'"]).isEqualTo("
16f1ed75-0bcc-4f0d-a04d-3121798faf99")

assertThatJson(parsedJson).array(["'events'"]).contains(["'operation'"]).isEqualTo(
("EXPORT")

assertThatJson(parsedJson).array(["'events'"]).contains(["'operation'"]).isEqualTo(
("INPUT_PROCESSING")

assertThatJson(parsedJson).array(["'events'"]).contains(["'eventId'"]).isEqualTo("
3bb4ac82-6652-462f-b6d1-75e424a0024a")

assertThatJson(parsedJson).array(["'events'"]).contains(["'status'"]).isEqualTo("0
K")
and:
    assertThat(parsedJson.read("\$.events[0].operation",
String.class)).matches(".*")
    assertThat(parsedJson.read("\$.events[0].eventId",
String.class)).matches("^[a-fA-F0-9]{8}-[a-fA-F0-9]{4}-[a-fA-F0-9]{4}-[a-fA-F0-
9]{4}-[a-fA-F0-9]{12}$")
    assertThat(parsedJson.read("\$.events[0].status", String.class)).matches(".*")

```

Note that the assertion is malformed. Only the first element of the array got asserted. To fix this, apply the assertion to the whole `$.events` collection and assert it with the `byCommand(...)` method.

3.2.5. Asynchronous Support

If you use asynchronous communication on the server side (your controllers are returning `Callable`, `DeferredResult`, and so on), then, inside your contract, you must provide an `async()` method in the `response` section. The following code shows an example:

Groovy

```
org.springframework.cloud.contract.spec.Contract.make {
    request {
        method GET()
        url '/get'
    }
    response {
        status OK()
        body 'Passed'
        async()
    }
}
```

YAML

```
response:
  async: true
```

Java

```
class contract implements Supplier<Collection<Contract>> {

    @Override
    public Collection<Contract> get() {
        return Collections.singletonList(Contract.make(c -> {
            c.request(r -> {
                // ...
            });
            c.response(r -> {
                r.async();
                // ...
            });
        }));
    }
}
```

Kotlin

```
import org.springframework.cloud.contract.spec.ContractDsl.Companion.contract

contract {
    request {
        // ...
    }
    response {
        async = true
        // ...
    }
}
```

You can also use the `fixedDelayMilliseconds` method or property to add delay to your stubs. The following example shows how to do so:

Groovy

```
org.springframework.cloud.contract.spec.Contract.make {
    request {
        method GET()
        url '/get'
    }
    response {
        status 200
        body 'Passed'
        fixedDelayMilliseconds 1000
    }
}
```

YAML

```
response:
  fixedDelayMilliseconds: 1000
```

Java

```
class contract implements Supplier<Collection<Contract>> {

    @Override
    public Collection<Contract> get() {
        return Collections.singletonList(Contract.make(c -> {
            c.request(r -> {
                // ...
            });
            c.response(r -> {
                r.fixedDelayMilliseconds(1000);
                // ...
            });
        }));
    }
}
```

Kotlin

```
import org.springframework.cloud.contract.spec.ContractDsl.Companion.contract

contract {
    request {
        // ...
    }
    response {
        delay = fixedMilliseconds(1000)
        // ...
    }
}
```

3.2.6. XML Support for HTTP

For HTTP contracts, we also support using XML in the request and response body. The XML body has to be passed within the `body` element as a `String` or `GString`. Also, body matchers can be provided for both the request and the response. In place of the `jsonPath(...)` method, the `org.springframework.cloud.contract.spec.internal.BodyMatchers.xpath` method should be used, with the desired `xPath` provided as the first argument and the appropriate `MatchingType` as the second argument. All the body matchers apart from `byType()` are supported.

The following example shows a Groovy DSL contract with XML in the response body:

```

Contract.make {
    request {
        method GET()
        urlPath '/get'
        headers {
            contentType(applicationXml())
        }
    }
    response {
        status(OK())
        headers {
            contentType(applicationXml())
        }
        body ""
    }
}

<test>
<duck type='xtype'>123</duck>
<alpha>abc</alpha>
<list>
<elem>abc</elem>
<elem>def</elem>
<elem>ghi</elem>
</list>
<number>123</number>
<aBoolean>true</aBoolean>
<date>2017-01-01</date>
<dateTime>2017-01-01T01:23:45</dateTime>
<time>01:02:34</time>
<valueWithoutAMatcher>foo</valueWithoutAMatcher>
<key><complex>foo</complex></key>
</test>""

    bodyMatchers {
        XPath('/test/duck/text()', byRegexp("[0-9]{3}"))
        XPath('/test/duck/text()',
byCommand('equals($it)'))
        XPath('/test/duck/xxx', byNull())
        XPath('/test/duck/text()', byEquality())
        XPath('/test/alpha/text()',
byRegex(onlyAlphaUnicode()))
        XPath('/test/alpha/text()', byEquality())
        XPath('/test/number/text()', byRegex(number()))
        XPath('/test/date/text()', byDate())
        XPath('/test/dateTime/text()', byTimestamp())
        XPath('/test/time/text()', byTime())
        XPath('/test/*/complex/text()', byEquality())
        XPath('/test/duck/@type', byEquality())
    }
}

```

```

Contract.make {
  request {
    method GET()
    urlPath '/get'
    headers {
      contentType(applicationXml())
    }
  }
  response {
    status(OK())
    headers {
      contentType(applicationXml())
    }
    body ""
  }
}

<ns1:test xmlns:ns1="http://demo.com/testns">
  <ns1:header>
    <duck-bucket type='bigbucket'>
      <duck>duck5150</duck>
    </duck-bucket>
  </ns1:header>
</ns1:test>
""

bodyMatchers {
  XPath('/test/duck/text()', byRegex("[0-9]{3}"))
  XPath('/test/duck/text()',
byCommand('equals($it)'))

  XPath('/test/duck/xxx', byNull())
  XPath('/test/duck/text()', byEquality())
  XPath('/test/alpha/text()',
byRegex(onlyAlphaUnicode()))

  XPath('/test/alpha/text()', byEquality())
  XPath('/test/number/text()', byRegex(number()))
  XPath('/test/date/text()', byDate())
  XPath('/test/dateTime/text()', byTimestamp())
  XPath('/test/time/text()', byTime())
  XPath('/test/duck/@type', byEquality())
}
}

Contract.make {
  request {
    method GET()
    urlPath '/get'
    headers {
      contentType(applicationXml())
    }
  }
  response {
    status(OK())
    headers {
      contentType(applicationXml())
    }
  }
}

```

```

    }
    body ""
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
  <SOAP-ENV:Header>
    <RsHeader xmlns="http://schemas.xmlsoap.org/soap/custom">
      <MsgSeqId>1234</MsgSeqId>
    </RsHeader>
  </SOAP-ENV:Header>
</SOAP-ENV:Envelope>
"""

    bodyMatchers {
      xPath('//*[local-name()=\'RsHeader\' and
namespace-uri()=\'http://schemas.xmlsoap.org/soap/custom\' ]/*[local-
name()=\'MsgSeqId\' ]/text()', byEquality())
    }
  }
}
Contract.make {
  request {
    method GET()
    urlPath '/get'
    headers {
      contentType(applicationXml())
    }
  }
  response {
    status(OK())
    headers {
      contentType(applicationXml())
    }
    body ""
  }
}
<ns1:customer xmlns:ns1="http://demo.com/customer"
xmlns:addr="http://demo.com/address">
  <email>customer@test.com</email>
  <contact-info xmlns="http://demo.com/contact-info">
    <name>Krombopulous</name>
    <address>
      <addr:gps>
        <lat>51</lat>
        <addr:lon>50</addr:lon>
      </addr:gps>
    </address>
  </contact-info>
</ns1:customer>
"""
}
}

```

YAML

```
request:
```

```
method: GET
url: /getyamlResponse
headers:
  Content-Type: application/xml
body: |
  <test>
  <duck type='xtype'>123</duck>
  <alpha>abc</alpha>
  <list>
  <elem>abc</elem>
  <elem>def</elem>
  <elem>ghi</elem>
  </list>
  <number>123</number>
  <aBoolean>>true</aBoolean>
  <date>2017-01-01</date>
  <dateTime>2017-01-01T01:23:45</dateTime>
  <time>01:02:34</time>
  <valueWithoutAMatcher>foo</valueWithoutAMatcher>
  <valueWithTypeMatch>string</valueWithTypeMatch>
  <key><complex>foo</complex></key>
  </test>
matchers:
  body:
    - path: /test/duck/text()
      type: by_regex
      value: "[0-9]{10}"
    - path: /test/duck/text()
      type: by_equality
    - path: /test/time/text()
      type: by_time
response:
  status: 200
  headers:
    Content-Type: application/xml
  body: |
    <test>
    <duck type='xtype'>123</duck>
    <alpha>abc</alpha>
    <list>
    <elem>abc</elem>
    <elem>def</elem>
    <elem>ghi</elem>
    </list>
    <number>123</number>
    <aBoolean>>true</aBoolean>
    <date>2017-01-01</date>
    <dateTime>2017-01-01T01:23:45</dateTime>
    <time>01:02:34</time>
    <valueWithoutAMatcher>foo</valueWithoutAMatcher>
    <valueWithTypeMatch>string</valueWithTypeMatch>
```



```

<key><complex>foo</complex></key>
</test>
matchers:
  body:
    - path: /test/duck/text()
      type: by_regex
      value: "[0-9]{10}"
    - path: /test/duck/text()
      type: by_command
      value: "test($it)"
    - path: /test/duck/xxx
      type: by_null
    - path: /test/duck/text()
      type: by_equality
    - path: /test/time/text()
      type: by_time

```

Java

```

import java.util.function.Supplier;

import org.springframework.cloud.contract.spec.Contract;

class contract_xml implements Supplier<Contract> {

    @Override
    public Contract get() {
        return Contract.make(c -> {
            c.request(r -> {
                r.method(r.GET());
                r.urlPath("/get");
                r.headers(h -> {
                    h.contentType(h.applicationXml());
                });
            });
            c.response(r -> {
                r.status(r.OK());
                r.headers(h -> {
                    h.contentType(h.applicationXml());
                });
                r.body("<test>\n" + "<duck type='xtype'>123</duck>\n" +
"<alpha>abc</alpha>\n" + "<list>\n"
                    + "<elem>abc</elem>\n" + "<elem>def</elem>\n" +
"<elem>ghi</elem>\n" + "</list>\n"
                    + "<number>123</number>\n" + "<aBoolean>true</aBoolean>\n"
+ "<date>2017-01-01</date>\n"
                    + "<dateTime>2017-01-01T01:23:45</dateTime>\n" +
"<time>01:02:34</time>\n"
                    + "<valueWithoutAMatcher>foo</valueWithoutAMatcher>\n" +
"<key><complex>foo</complex></key>\n"
                    + "</test>");
            });
        });
    }
}

```

```
    r.bodyMatchers(m -> {
      m.xpath("/test/duck/text()", m.byRegex("[0-9]{3}"));
      m.xpath("/test/duck/text()", m.byCommand("equals($it)"));
      m.xpath("/test/duck/xxx", m.byNull());
      m.xpath("/test/duck/text()", m.byEquality());
      m.xpath("/test/alpha/text()",
m.byRegex(r.onlyAlphaUnicode()));
      m.xpath("/test/alpha/text()", m.byEquality());
      m.xpath("/test/number/text()", m.byRegex(r.number()));
      m.xpath("/test/date/text()", m.byDate());
      m.xpath("/test/dateTime/text()", m.byTimestamp());
      m.xpath("/test/time/text()", m.byTime());
      m.xpath("/test/*/complex/text()", m.byEquality());
      m.xpath("/test/duck/@type", m.byEquality());
    });
  });
};
}
```

```

import org.springframework.cloud.contract.spec.ContractDsl.Companion.contract

contract {
    request {
        method = GET
        urlPath = path("/get")
        headers {
            contentType = APPLICATION_XML
        }
    }
    response {
        status = OK
        headers {
            contentType =APPLICATION_XML
        }
        body = body("<test>\n" + "<duck type='xtype'>123</duck>\n"
            + "<alpha>abc</alpha>\n" + "<list>\n" + "<elem>abc</elem>\n"
            + "<elem>def</elem>\n" + "<elem>ghi</elem>\n" + "</list>\n"
            + "<number>123</number>\n" + "<aBoolean>true</aBoolean>\n"
            + "<date>2017-01-01</date>\n"
            + "<dateTime>2017-01-01T01:23:45</dateTime>\n"
            + "<time>01:02:34</time>\n"
            + "<valueWithoutAMatcher>foo</valueWithoutAMatcher>\n"
            + "<key><complex>foo</complex></key>\n" + "</test>")
        bodyMatchers {
            xPath("/test/duck/text()", byRegex("[0-9]{3}"))
            xPath("/test/duck/text()", byCommand("equals(\$it)"))
            xPath("/test/duck/xxx", byNull)
            xPath("/test/duck/text()", byEquality)
            xPath("/test/alpha/text()", byRegex(onlyAlphaUnicode))
            xPath("/test/alpha/text()", byEquality)
            xPath("/test/number/text()", byRegex(number))
            xPath("/test/date/text()", byDate)
            xPath("/test/dateTime/text()", byTimestamp)
            xPath("/test/time/text()", byTime)
            xPath("/test/*/complex/text()", byEquality)
            xPath("/test/duck/@type", byEquality)
        }
    }
}

```

The following example shows an automatically generated test for XML in the response body:

```

@Test
public void validate_xmlMatches() throws Exception {
    // given:
    MockMvcRequestSpecification request = given()
        .header("Content-Type", "application/xml");

    // when:
    ResponseOptions response = given().spec(request).get("/get");

    // then:
    assertThat(response.statusCode()).isEqualTo(200);
    // and:
    DocumentBuilder documentBuilder = DocumentBuilderFactory.newInstance()
        .newDocumentBuilder();
    Document parsedXml = documentBuilder.parse(new InputSource(
        new StringReader(response.getBody().asString())));
    // and:
    assertThat(valueFromXPath(parsedXml,
"/test/list/elm/text(")).isEqualTo("abc");

    assertThat(valueFromXPath(parsedXml, "/test/list/elm[2]/text(")).isEqualTo("def")
;
    assertThat(valueFromXPath(parsedXml, "/test/duck/text(")).matches("[0-9]{3}");
    assertThat(nodeFromXPath(parsedXml, "/test/duck/xxx").isNull());
    assertThat(valueFromXPath(parsedXml,
"/test/alpha/text(")).matches("[\\p{L}]*");
    assertThat(valueFromXPath(parsedXml,
"/test/*/complex/text(")).isEqualTo("foo");
    assertThat(valueFromXPath(parsedXml, "/test/duck/@type")).isEqualTo("xtype");
}

```

XML Support for Namespaces

Namespaced XML is supported. However, any XPath expressions used to select namespaced content must be updated.

Consider the following explicitly namespaced XML document:

```

<ns1:customer xmlns:ns1="http://demo.com/customer">
  <email>customer@test.com</email>
</ns1:customer>

```

The XPath expression to select the email address is: `/ns1:customer/email/text()`.



Beware as the unqualified expression (`/customer/email/text()`) results in "".

For content that uses an unqualified namespace, the expression is more verbose. Consider the following XML document that uses an unqualified namespace:

```
<customer xmlns="http://demo.com/customer">
  <email>customer@test.com</email>
</customer>
```

The XPath expression to select the email address is

```
*/[local-name()='customer' and namespace-uri()='http://demo.com/customer']/*[local-name()='email']/text()
```



Beware, as the unqualified expressions (`/customer/email/text()` or `*/[local-name()='customer' and namespace-uri()='http://demo.com/customer']/email/text()`) result in `""`. Even the child elements have to be referenced with the `local-name` syntax.

General Namespaced Node Expression Syntax

- Node using qualified namespace:

```
/<node-name>
```

- Node using and defining an unqualified namespace:

```
/*[local-name=()='<node-name>' and namespace-uri=()='<namespace-uri>']
```



In some cases, you can omit the `namespace_uri` portion, but doing so may lead to ambiguity.

- Node using an unqualified namespace (one of its ancestor's defines the `xmlns` attribute):

```
/*[local-name=()='<node-name>']
```

3.2.7. Multiple Contracts in One File

You can define multiple contracts in one file. Such a contract might resemble the following example:

Groovy

```
import org.springframework.cloud.contract.spec.Contract

[
  Contract.make {
    name("should post a user")
    request {
      method 'POST'
      url('/users/1')
    }
    response {
      status OK()
    }
  },
  Contract.make {
    request {
      method 'POST'
      url('/users/2')
    }
    response {
      status OK()
    }
  }
]
```

YAML

```
---
name: should post a user
request:
  method: POST
  url: /users/1
response:
  status: 200
---
request:
  method: POST
  url: /users/2
response:
  status: 200
---
request:
  method: POST
  url: /users/3
response:
  status: 200
```

Java

```
class contract implements Supplier<Collection<Contract>> {

    @Override
    public Collection<Contract> get() {
        return Arrays.asList(
            Contract.make(c -> {
                c.name("should post a user");
                // ...
            }), Contract.make(c -> {
                // ...
            }), Contract.make(c -> {
                // ...
            })
        );
    }
}
```

Kotlin

```
import org.springframework.cloud.contract.spec.ContractDsl.Companion.contract

arrayOf(
    contract {
        name("should post a user")
        // ...
    },
    contract {
        // ...
    },
    contract {
        // ...
    }
}
```

In the preceding example, one contract has the `name` field and the other does not. This leads to generation of two tests that look like the following:

```

package org.springframework.cloud.contract.verifier.tests.com.hello;

import com.example.TestBase;
import com.jayway.jsonpath.DocumentContext;
import com.jayway.jsonpath.JsonPath;
import
com.jayway.restassured.module.mockmvc.specification.MockMvcRequestSpecification;
import com.jayway.restassured.response.ResponseOptions;
import org.junit.Test;

import static com.jayway.restassured.module.mockmvc.RestAssuredMockMvc.*;
import static com.toomuchcoding.jsonassert.JsonAssertion.assertThatJson;
import static org.assertj.core.api.Assertions.assertThat;

public class V1Test extends TestBase {

    @Test
    public void validate_should_post_a_user() throws Exception {
        // given:
        MockMvcRequestSpecification request = given();

        // when:
        ResponseOptions response = given().spec(request)
            .post("/users/1");

        // then:
        assertThat(response.statusCode()).isEqualTo(200);
    }

    @Test
    public void validate_withList_1() throws Exception {
        // given:
        MockMvcRequestSpecification request = given();

        // when:
        ResponseOptions response = given().spec(request)
            .post("/users/2");

        // then:
        assertThat(response.statusCode()).isEqualTo(200);
    }
}

```

Notice that, for the contract that has the `name` field, the generated test method is named `validate_should_post_a_user`. The one that does not have the `name` field is called `validate_withList_1`. It corresponds to the name of the file `WithList.groovy` and the index of the contract in the list.

The generated stubs are shown in the following example:

```
should post a user.json
1_WithList.json
```

The first file got the `name` parameter from the contract. The second got the name of the contract file (`WithList.groovy`) prefixed with the index (in this case, the contract had an index of `1` in the list of contracts in the file).



It is much better to name your contracts, because doing so makes your tests far more meaningful.

3.2.8. Stateful Contracts

Stateful contracts (also known as scenarios) are contract definitions that should be read in order. This might be useful in the following situations:

- You want to invoke the contract in a precisely defined order, since you use Spring Cloud Contract to test your stateful application.



We really discourage you from doing that, since contract tests should be stateless.

- You want the same endpoint to return different results for the same request.

To create stateful contracts (or scenarios), you need to use the proper naming convention while creating your contracts. The convention requires including an order number followed by an underscore. This works regardless of whether you work with YAML or Groovy. The following listing shows an example:

```
my_contracts_dir\  
  scenario1\  
    1_login.groovy  
    2_showCart.groovy  
    3_logout.groovy
```

Such a tree causes Spring Cloud Contract Verifier to generate WireMock's scenario with a name of `scenario1` and the three following steps:

1. `login`, marked as `Started` pointing to...
2. `showCart`, marked as `Step1` pointing to...
3. `logout`, marked as `Step2` (which closes the scenario).

You can find more details about WireMock scenarios at <https://wiremock.org/docs/stateful->

[behaviour/](#).

3.3. Integrations

3.3.1. JAX-RS

The Spring Cloud Contract supports the JAX-RS 2 Client API. The base class needs to define `protected WebTarget webTarget` and server initialization. The only option for testing JAX-RS API is to start a web server. Also, a request with a body needs to have a content type be set. Otherwise, the default of `application/octet-stream` gets used.

To use JAX-RS mode, use the following setting:

```
testMode = 'JAXRSCLIENT'
```

The following example shows a generated test API:

```

package com.example;

import com.jayway.jsonpath.DocumentContext;
import com.jayway.jsonpath.JsonPath;
import org.junit.Test;
import org.junit.Rule;
import javax.ws.rs.client.Entity;
import javax.ws.rs.core.Response;

import static
org.springframework.cloud.contract.verifier.assertion.SpringCloudContractAssertion
s.assertThat;
import static
org.springframework.cloud.contract.verifier.util.ContractVerifierUtil.*;
import static com.toomuchcoding.jsonassert.JsonAssertion.assertThatJson;
import static javax.ws.rs.client.Entity.*;

@SuppressWarnings("rawtypes")
public class FooTest {
    WebTarget webTarget;

    @Test
    public void validate_() throws Exception {

        // when:
        Response response = webTarget
            .path("/users")
            .queryParams("limit", "10")
            .queryParams("offset", "20")
            .queryParams("filter", "email")
            .queryParams("sort", "name")
            .queryParams("search", "55")
            .queryParams("age", "99")
            .queryParams("name", "Denis.Stepanov")
            .queryParams("email", "bob@email.com")
            .request()
            .build("GET")
            .invoke();
        String responseAsString = response.readEntity(String.class);

        // then:
        assertThat(response.getStatus()).isEqualTo(200);

        // and:
        DocumentContext parsedJson = JsonPath.parse(responseAsString);
        assertThatJson(parsedJson).field("[ 'property1' ]").isEqualTo("a");
    }
}

```

3.3.2. WebFlux with WebTestClient

You can work with WebFlux by using WebTestClient. The following listing shows how to configure WebTestClient as the test mode:

Maven

```
<plugin>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-contract-maven-plugin</artifactId>
  <version>${spring-cloud-contract.version}</version>
  <extensions>true</extensions>
  <configuration>
    <testMode>WEBTESTCLIENT</testMode>
  </configuration>
</plugin>
```

Gradle

```
contracts {
    testMode = 'WEBTESTCLIENT'
}
```

The following example shows how to set up a WebTestClient base class and RestAssured for WebFlux:

```
import io.restassured.module.webtestclient.RestAssuredWebTestClient;
import org.junit.Before;

public abstract class BeerRestBase {

    @Before
    public void setup() {
        RestAssuredWebTestClient.standaloneSetup(
            new ProducerController(personToCheck -> personToCheck.age >= 20));
    }
}
```



The `WebTestClient` mode is faster than the `EXPLICIT` mode.

3.3.3. WebFlux with Explicit Mode

You can also use WebFlux with the explicit mode in your generated tests to work with WebFlux. The following example shows how to configure using explicit mode:

Maven

```
<plugin>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-contract-maven-plugin</artifactId>
  <version>${spring-cloud-contract.version}</version>
  <extensions>>true</extensions>
  <configuration>
    <testMode>EXPLICIT</testMode>
  </configuration>
</plugin>
```

Gradle

```
contracts {
    testMode = 'EXPLICIT'
}
```

The following example shows how to set up a base class and RestAssured for Web Flux:

```
@SpringBootTest(classes = BeerRestBase.Config.class,
    webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT,
    properties = "server.port=0")
public abstract class BeerRestBase {

    // your tests go here

    // in this config class you define all controllers and mocked services
    @Configuration
    @EnableAutoConfiguration
    static class Config {

        @Bean
        PersonCheckingService personCheckingService() {
            return personToCheck -> personToCheck.age >= 20;
        }

        @Bean
        ProducerController producerController() {
            return new ProducerController(personCheckingService());
        }
    }
}
```

3.3.4. Custom Mode



This mode is experimental and can change in the future.

The Spring Cloud Contract lets you provide your own, custom, implementation of the `org.springframework.cloud.contract.verifier.http.HttpVerifier`. That way, you can use any client you want to send and receive a request. The default implementation in Spring Cloud Contract is `OkHttpHttpVerifier` and it uses OkHttp3 http client.

To get started, set `testMode` to `CUSTOM`:

```
testMode = 'CUSTOM'
```

The following example shows a generated test:

```

package com.example.beer;

import com.example.BeerRestBase;
import javax.inject.Inject;
import org.springframework.cloud.contract.verifier.http.HttpVerifier;
import org.springframework.cloud.contract.verifier.http.Request;
import org.springframework.cloud.contract.verifier.http.Response;
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.extension.ExtendWith;

import static
org.springframework.cloud.contract.verifier.assertion.SpringCloudContractAssertion
s.assertThat;
import static
org.springframework.cloud.contract.verifier.util.ContractVerifierUtil.*;
import static org.springframework.cloud.contract.verifier.http.Request.given;

@SuppressWarnings("rawtypes")
public class RestTest extends BeerRestBase {
    @Inject HttpVerifier httpVerifier;

    @Test
    public void validate_shouldGrantABeerIfOldEnough() throws Exception {
        // given:
        Request request = given()
            .post("/beer.BeerService/check")
            .scheme("HTTP")
            .protocol("h2_prior_knowledge")
            .header("Content-Type", "application/grpc")
            .header("te", "trailers")
            .body(fileToBytes(this,
"shouldGrantABeerIfOldEnough_request_PersonToCheck_old_enough.bin"))
            .build();

        // when:
        Response response = httpVerifier.exchange(request);

        // then:
        assertThat(response.statusCode()).isEqualTo(200);
        assertThat(response.header("Content-
Type")).matches("application/grpc.*");
        assertThat(response.header("grpc-encoding")).isEqualTo("identity");
        assertThat(response.header("grpc-accept-encoding")).isEqualTo("gzip");

        // and:

        assertThat(response.getBody().asByteArray()).isEqualTo(fileToBytes(this,

```

```
"shouldGrantABeerIfOldEnough_response_Response_old_enough.bin"));  
    }  
  
}
```

The following example shows a corresponding base class:

```
@SpringBootTest(classes = BeerRestBase.Config.class,  
    webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT)  
public abstract class BeerRestBase {  
  
    @Configuration  
    @EnableAutoConfiguration  
    static class Config {  
  
        @Bean  
        ProducerController producerController(PersonCheckingService  
personCheckingService) {  
            return new ProducerController(personCheckingService);  
        }  
  
        @Bean  
        PersonCheckingService testPersonCheckingService() {  
            return argument -> argument.getAge() >= 20;  
        }  
  
        @Bean  
        HttpVerifier httpOkVerifier(@LocalServerPort int port) {  
            return new OkHttpHttpVerifier("localhost:" + port);  
        }  
    }  
}
```

3.3.5. Working with Context Paths

Spring Cloud Contract supports context paths.

The only change needed to fully support context paths is the switch on the producer side. Also, the autogenerated tests must use explicit mode. The consumer side remains untouched. In order for the generated test to pass, you must use explicit mode. The following example shows how to set the test mode to **EXPLICIT**:



Maven

```
<plugin>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-contract-maven-plugin</artifactId>
  <version>${spring-cloud-contract.version}</version>
  <extensions>>true</extensions>
  <configuration>
    <testMode>EXPLICIT</testMode>
  </configuration>
</plugin>
```

Gradle

```
contracts {
    testMode = 'EXPLICIT'
}
```

That way, you generate a test that does not use MockMvc. It means that you generate real requests and you need to set up your generated test's base class to work on a real socket.

Consider the following contract:

```
org.springframework.cloud.contract.spec.Contract.make {
    request {
        method 'GET'
        url '/my-context-path/url'
    }
    response {
        status OK()
    }
}
```

The following example shows how to set up a base class and RestAssured:

```

import io.restassured.RestAssured;
import org.junit.Before;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.boot.test.web.server.LocalServerPort;

@SpringBootTest(classes = ContextPathTestingBaseClass.class, webEnvironment =
SpringBootTest.WebEnvironment.RANDOM_PORT)
class ContextPathTestingBaseClass {

    @LocalServerPort int port;

    @Before
    public void setup() {
        RestAssured.baseURI = "http://localhost";
        RestAssured.port = this.port;
    }
}

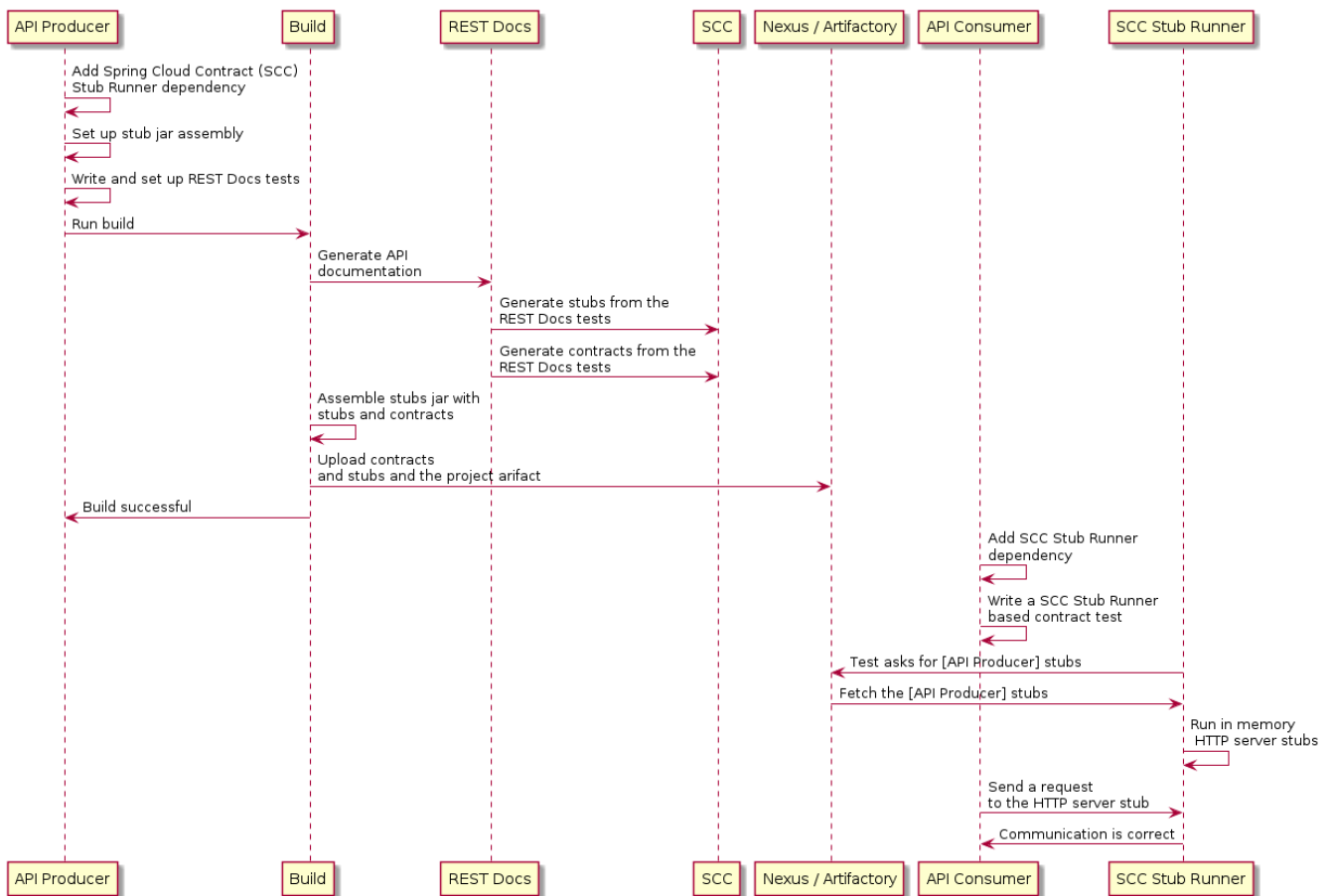
```

If you do it this way:

- All of your requests in the autogenerated tests are sent to the real endpoint with your context path included (for example, `/my-context-path/url`).
- Your contracts reflect that you have a context path. Your generated stubs also have that information (for example, in the stubs, you have to call `/my-context-path/url`).

3.3.6. Working with REST Docs

You can use [Spring REST Docs](#) to generate documentation (for example, in AsciiDoc format) for an HTTP API with Spring MockMvc, WebTestClient, or RestAssured. At the same time that you generate documentation for your API, you can also generate WireMock stubs by using Spring Cloud Contract WireMock. To do so, write your normal REST Docs test cases and use `@AutoConfigureRestDocs` to have stubs be automatically generated in the REST Docs output directory. The following UML diagram shows the REST Docs flow:



The following example uses `MockMvc`:

```

@RunWith(SpringRunner.class)
@SpringBootTest
@AutoConfigureRestDocs(outputDir = "target/snippets")
@AutoConfigureMockMvc
public class ApplicationTests {

    @Autowired
    private MockMvc mockMvc;

    @Test
    public void contextLoads() throws Exception {
        mockMvc.perform(get("/resource"))
            .andExpect(content().string("Hello World"))
            .andDo(document("resource"));
    }
}
  
```

This test generates a WireMock stub at `target/snippets/stubs/resource.json`. It matches all `GET` requests to the `/resource` path. The same example with `WebTestClient` (used for testing Spring WebFlux applications) would be as follows:

```

@RunWith(SpringRunner.class)
@SpringBootTest
@AutoConfigureRestDocs(outputDir = "target/snippets")
@AutoConfigureWebTestClient
public class ApplicationTests {

    @Autowired
    private WebTestClient client;

    @Test
    public void contextLoads() throws Exception {
        client.get().uri("/resource").exchange()
            .expectBody(String.class).isEqualTo("Hello World")
            .consumeWith(document("resource"));
    }
}

```

Without any additional configuration, these tests create a stub with a request matcher for the HTTP method and all headers except `host` and `content-length`. To match the request more precisely (for example, to match the body of a POST or PUT), we need to explicitly create a request matcher. Doing so has two effects:

- Creating a stub that matches only in the way you specify.
- Asserting that the request in the test case also matches the same conditions.

The main entry point for this feature is `WireMockRestDocs.verify()`, which can be used as a substitute for the `document()` convenience method, as the following example shows:

```

import static
org.springframework.cloud.contract.wiremock.restdocs.WireMockRestDocs.verify;

@RunWith(SpringRunner.class)
@SpringBootTest
@AutoConfigureRestDocs(outputDir = "target/snippets")
@AutoConfigureMockMvc
public class ApplicationTests {

    @Autowired
    private MockMvc mockMvc;

    @Test
    public void contextLoads() throws Exception {
        mockMvc.perform(post("/resource")
            .content("{\"id\":\"123456\",\"message\":\"Hello World\"}")
            .andExpect(status().isOk())
            .andDo(verify().jsonPath("$.id"))
            .andDo(document("resource")));
    }
}

```

The preceding contract specifies that any valid POST with an `id` field receives the response defined in this test. You can chain together calls to `.jsonPath()` to add additional matchers. If JSON Path is unfamiliar, the [JayWay documentation](#) can help you get up to speed. The `WebTestClient` version of this test has a similar `verify()` static helper that you insert in the same place.

Instead of the `jsonPath` and `contentType` convenience methods, you can also use the WireMock APIs to verify that the request matches the created stub, as the following example shows:

```

@Test
public void contextLoads() throws Exception {
    mockMvc.perform(post("/resource")
        .content("{\"id\":\"123456\",\"message\":\"Hello World\"}")
        .andExpect(status().isOk())
        .andDo(verify()
            .wiremock(WireMock.post(urlPathEquals("/resource"))
                .withRequestBody(matchingJsonPath("$.id"))
                .andDo(document("post-resource"))));
    }
}

```

The WireMock API is rich. You can match headers, query parameters, and the request body by regex as well as by JSON path. You can use these features to create stubs with a wider range of parameters. The preceding example generates a stub resembling the following example:

post-resource.json

```
{
  "request" : {
    "url" : "/resource",
    "method" : "POST",
    "bodyPatterns" : [ {
      "matchesJsonPath" : "$.id"
    }
  ],
  "response" : {
    "status" : 200,
    "body" : "Hello World",
    "headers" : {
      "X-Application-Context" : "application:-1",
      "Content-Type" : "text/plain"
    }
  }
}
```



You can use either the `wiremock()` method or the `jsonPath()` and `contentType()` methods to create request matchers, but you cannot use both approaches.

On the consumer side, you can make the `resource.json` generated earlier in this section available on the classpath (by [Publishing Stubs as JARs](#), for example). After that, you can create a stub that uses WireMock in a number of different ways, including by using `@AutoConfigureWireMock(stubs="classpath:resource.json")`, as described earlier in this document.

Generating Contracts with REST Docs

You can also generate Spring Cloud Contract DSL files and documentation with Spring REST Docs. If you do so in combination with Spring Cloud WireMock, you get both the contracts and the stubs.

Why would you want to use this feature? Some people in the community asked questions about a situation in which they would like to move to DSL-based contract definition, but they already have a lot of Spring MVC tests. Using this feature lets you generate the contract files that you can later modify and move to folders (defined in your configuration) so that the plugin finds them.



You might wonder why this functionality is in the WireMock module. The functionality is there because it makes sense to generate both the contracts and the stubs.

Consider the following test:

```

this.mockMvc

.perform(post("/foo").accept(MediaType.APPLICATION_PDF).accept(MediaType.APPLICATION_JSON)

.contentType(MediaType.APPLICATION_JSON).content("{\"foo\": 23, \"bar\" : \"baz\"
}"))

    .andExpect(status().isOk()).andExpect(content().string("bar"))
    // first WireMock
    .andDo(WireMockRestDocs.verify().jsonPath("$.foo >= 20)")
        .jsonPath("$.bar in ['baz', 'bazz', 'bazzz']")
        .contentType(MediaType.valueOf("application/json")))
    // then Contract DSL documentation
    .andDo(document("index",
SpringCloudContractRestDocs.dslContract(Maps.of("priority", 1))));

```

The preceding test creates the stub presented in the previous section, generating both the contract and a documentation file.

The contract is called `index.groovy` and might resemble the following example:

```

import org.springframework.cloud.contract.spec.Contract

Contract.make {
    request {
        method 'POST'
        url '/foo'
        body(''
            {"foo": 23 }
        '')
        headers {
            header(''Accept'', ''application/json'')
            header(''Content-Type'', ''application/json'')
        }
    }
    response {
        status OK()
        body(''
            bar
        '')
        headers {
            header(''Content-Type'', ''application/json;charset=UTF-8'')
            header(''Content-Length'', ''3'')
        }
        bodyMatchers {
            jsonPath('$[?(@.foo >= 20)]', byType())
        }
    }
}

```

The generated document (formatted in AsciiDoc in this case) contains a formatted contract. The location of this file would be [index/dsl-contract.adoc](#).

Specifying the priority attribute

The method `SpringCloudContractRestDocs.dslContract()` takes an optional `Map` parameter that allows you to specify additional attributes in the template.

One of these attributes is the [priority](#) field that you may specify as follows:

```
SpringCloudContractRestDocs.dslContract(Map.of("priority", 1))
```

Overriding the DSL contract template

By default, the output of the contract is based on a file named `default-dsl-contract-only.snippet`.

You may provide a custom template file instead by overriding the `getTemplate()` method as follows:


```
new ContractDslSnippet(){
    @Override
    protected String getTemplate() {
        return "custom-dsl-contract";
    }
});
```

so the example above showing this line

```
.andDo(document("index", SpringCloudContractRestDocs.dslContract()));
```

should be changed to:

```
.andDo(document("index", new ContractDslSnippet(){
    @Override
    protected String getTemplate() {
        return "custom-dsl-template";
    }
}));
```

Templates are resolved by looking for resources on the classpath. The following locations are checked in order:

- `org/springframework/restdocs/templates/${templateFormatId}/${name}.snippet`
- `org/springframework/restdocs/templates/${name}.snippet`
- `org/springframework/restdocs/templates/${templateFormatId}/default-${name}.snippet`

Therefore in the example above you should place a file named `custom-dsl-template.snippet` in `src/test/resources/org/springframework/restdocs/templates/custom-dsl-template.snippet`

3.3.7. GraphQL

Since [GraphQL](#) is essentially HTTP you can write a contract for it by creating a standard HTTP contract with an additional `metadata` entry with key `verifier` and a mapping `tool=graphql`.

Groovy

```
import org.springframework.cloud.contract.spec.Contract

Contract.make {

    request {
        method(POST())
        url("/graphql")
        headers {
            contentType("application/json")
        }
        body('''
{
    "query":"query queryName($personName: String!) {\n  personToCheck(name:
$personName) {\n    name\n    age\n  }\n}\n\n\n\n",
    "variables":{"personName":"Old Enough"},
    "operationName":"queryName"
}
''')
    }

    response {
        status(200)
        headers {
            contentType("application/json")
        }
        body('''\
{
  "data": {
    "personToCheck": {
      "name": "Old Enough",
      "age": "40"
    }
  }
}
''')
    }

    metadata(verifier: [
        tool: "graphql"
    ])
}
```

YAML

```
---
request:
  method: "POST"
  url: "/graphql"
  headers:
    Content-Type: "application/json"
  body:
    query: "query queryName($personName: String!) { personToCheck(name:
$personName)
      {      name      age  } }"
    variables:
      personName: "Old Enough"
      operationName: "queryName"
  matchers:
    headers:
      - key: "Content-Type"
        regex: "application/json.*"
        regexType: "as_string"
response:
  status: 200
  headers:
    Content-Type: "application/json"
  body:
    data:
      personToCheck:
        name: "Old Enough"
        age: "40"
  matchers:
    headers:
      - key: "Content-Type"
        regex: "application/json.*"
        regexType: "as_string"
name: "shouldRetrieveOldEnoughPerson"
metadata:
  verifier:
    tool: "graphql"
```

Adding the metadata section will change the way the default, WireMock stub is built. It will now use the Spring Cloud Contract request matcher, so that e.g. the **query** part of the GraphQL request gets compared against the real request by ignoring whitespaces.

Producer Side Setup

On the producer side your configuration can look as follows.

Maven

```
<plugin>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-contract-maven-plugin</artifactId>
  <version>${spring-cloud-contract.version}</version>
  <extensions>true</extensions>
  <configuration>
    <testMode>EXPLICIT</testMode>
    <baseClassForTests>com.example.BaseClass</baseClassForTests>
  </configuration>
</plugin>
```

Gradle

```
contracts {
  testMode = "EXPLICIT"
  baseClassForTests = "com.example.BaseClass"
}
```

The base class would set up the application running on a random port.

Base Class

```
@SpringBootTest(classes = ProducerApplication.class,
  properties = "graphql.servlet.websocket.enabled=false",
  webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT)
public abstract class BaseClass {

  @LocalServerPort int port;

  @BeforeEach
  public void setup() {
    RestAssured.baseURI = "http://localhost:" + port;
  }
}
```

Consumer Side Setup

Example of a consumer side test of the GraphQL API.

Consumer Side Test

```
@SpringBootTest(webEnvironment = WebEnvironment.NONE)
public class BeerControllerGraphQLTest {

    @RegisterExtension
    static StubRunnerExtension rule = new StubRunnerExtension()
        .downloadStub("com.example", "beer-api-producer-graphql")
        .stubsMode(StubRunnerProperties.StubsMode.LOCAL);

    private static final String REQUEST_BODY = "{\n"
        + "\"query\": \"query queryName($personName: String!) {\n"
        + "  personToCheck(name: $personName) {\n"
        + "    name\n"
        + "    age\n"
        + "  }\n"
        + "}\""}";

    @Test
    public void should_send_a_graphql_request() {
        ResponseEntity<String> responseEntity = new RestTemplate()
            .exchange(RequestEntity
                .post(URI.create("http://localhost:" +
                    rule.findStubUrl("beer-api-producer-graphql").getPort() + "/graphql"))
                .contentType(MediaType.APPLICATION_JSON)
                .body(REQUEST_BODY), String.class);

        BDDAssertions.then(responseEntity.getStatusCodeValue()).isEqualTo(200);
    }
}
```

3.3.8. GRPC

GRPC is an RPC framework built on top of HTTP/2 for which Spring Cloud Contract has basic support.



Spring Cloud Contract has an experimental support for basic use cases of GRPC. Unfortunately, due to GRPC's tweaking of the HTTP/2 Header frames, it's impossible to assert the `grpc-status` header.

Let's look at the following contract.

Groovy contract

```
package contracts.beer.rest

import org.springframework.cloud.contract.spec.Contract
import org.springframework.cloud.contract.verifier.http.ContractVerifierHttpMetaData

Contract.make {
    description("""
Represents a successful scenario of getting a beer
...
given:
    client is old enough
when:
    he applies for a beer
then:
    we'll grant him the beer
...

""")
    request {
        method 'POST'
        url '/beer.BeerService/check'
        body(fileAsBytes("PersonToCheck_old_enough.bin"))
        headers {
            contentType("application/grpc")
            header("te", "trailers")
        }
    }
    response {
        status 200
        body(fileAsBytes("Response_old_enough.bin"))
        headers {
            contentType("application/grpc")
            header("grpc-encoding", "identity")
            header("grpc-accept-encoding", "gzip")
        }
    }
    metadata([
        "verifierHttp": [
            "protocol":
ContractVerifierHttpMetaData.Protocol.H2_PRIOR_KNOWLEDGE.toString()
        ]
    ])
}
```

Producer Side Setup

In order to leverage the HTTP/2 support you must set the **CUSTOM** test mode as follow.

Maven

```
<plugin>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-contract-maven-plugin</artifactId>
  <version>${spring-cloud-contract.version}</version>
  <extensions>true</extensions>
  <configuration>
    <testMode>CUSTOM</testMode>
    <packageWithBaseClasses>com.example</packageWithBaseClasses>
  </configuration>
</plugin>
```

Gradle

```
contracts {
  packageWithBaseClasses = 'com.example'
  testMode = "CUSTOM"
}
```

The base class would set up the application running on a random port. It will also set the **HttpVerifier** implementation to one that can use the HTTP/2 protocol. Spring Cloud Contract comes with the **OkHttpHttpVerifier** implementation.

Base Class

```
@SpringBootTest(classes = BeerRestBase.Config.class,
    webEnvironment = SpringBootTest.WebEnvironment.NONE,
    properties = {
        "grpc.server.port=0"
    })
public abstract class BeerRestBase {

    @Autowired
    GrpcServerProperties properties;

    @Configuration
    @EnableAutoConfiguration
    static class Config {

        @Bean
        ProducerController producerController(PersonCheckingService
personCheckingService) {
            return new ProducerController(personCheckingService);
        }

        @Bean
        PersonCheckingService testPersonCheckingService() {
            return argument -> argument.getAge() >= 20;
        }

        @Bean
        HttpVerifier httpOkVerifier(GrpcServerProperties properties) {
            return new OkHttpHttpVerifier("localhost:" + properties.getPort());
        }
    }
}
```

Consumer Side Setup

Example of GRPC consumer side test. Due to the unusual behaviour of the GRPC server side, the stub is unable to return the `grpc-status` header in the proper moment. This is why we need to manually set the return status.

Consumer Side Test

```
@SpringBootTest(webEnvironment = WebEnvironment.NONE, classes =
GrpcTests.TestConfiguration.class, properties = {
    "grpc.client.beerService.address=static://localhost:5432",
    "grpc.client.beerService.negotiationType=TLS"
})
public class GrpcTests {

    @GrpcClient(value = "beerService", interceptorNames =
"fixedStatusSendingClientInterceptor")
    BeerServiceGrpc.BeerServiceBlockingStub beerServiceBlockingStub;

    int port;

    @RegisterExtension
    static StubRunnerExtension rule = new StubRunnerExtension()
        .downloadStub("com.example", "beer-api-producer-grpc")
        // With WireMock PlainText mode you can just set an HTTP port
        .withPort(5432)
        .stubsMode(StubRunnerProperties.StubsMode.LOCAL)
        .withHttpServerStubConfigurer(MyWireMockConfigurer.class);

    @BeforeEach
    public void setupPort() {
        this.port = rule.findStubUrl("beer-api-producer-grpc").getPort();
    }

    @Test
    public void should_give_me_a_beer_when_im_old_enough() throws Exception {
        Response response =
beerServiceBlockingStub.check(PersonToCheck.newBuilder().setAge(23).build());

        BDDAssertions.then(response.getStatus()).isEqualTo(Response.BeerCheckStatus.OK);
    }

    @Test
    public void should_reject_a_beer_when_im_too_young() throws Exception {
        Response response =
beerServiceBlockingStub.check(PersonToCheck.newBuilder().setAge(17).build());
        response = response == null ? Response.newBuilder().build() : response;

        BDDAssertions.then(response.getStatus()).isEqualTo(Response.BeerCheckStatus.NOT_OK
);
    }

    // Not necessary with WireMock PlainText mode
    static class MyWireMockConfigurer extends WireMockHttpServerStubConfigurer {
```

```

@Override
public WireMockConfiguration configure(WireMockConfiguration
httpStubConfiguration, HttpServerStubConfiguration httpServerStubConfiguration) {
    return httpStubConfiguration
        .httpsPort(5432);
}
}

@Configuration
@ImportAutoConfiguration(GrpcClientAutoConfiguration.class)
static class TestConfiguration {

    // Not necessary with WireMock PlainText mode
    @Bean
    public GrpcChannelConfigurer keepAliveClientConfigurer() {
        return (channelBuilder, name) -> {
            if (channelBuilder instanceof NettyChannelBuilder) {
                try {
                    ((NettyChannelBuilder) channelBuilder)
                        .sslContext(GrpcSslContexts.forClient()

.trustManager(InsecureTrustManagerFactory.INSTANCE)
                            .build());
                }
                catch (SSLException e) {
                    throw new IllegalStateException(e);
                }
            }
        };
    }

    /**
     * GRPC client interceptor that sets the returned status always to OK.
     * You might want to change the return status depending on the received
stub payload.
     *
     * Hopefully in the future this will be unnecessary and will be removed.
     */
    @Bean
    ClientInterceptor fixedStatusSendingClientInterceptor() {
        return new ClientInterceptor() {
            @Override
            public <ReqT, RespT> ClientCall<ReqT, RespT>
interceptCall(MethodDescriptor<ReqT, RespT> method, CallOptions callOptions,
Channel next) {
                ClientCall<ReqT, RespT> call = next.newCall(method,
callOptions);
                return new ClientCall<ReqT, RespT>() {
                    @Override
                    public void start(Listener<RespT> responseListener,
Metadata headers) {

```

```

Listener<RespT> listener = new Listener<RespT>() {
    @Override
    public void onHeaders(Metadata headers) {
        responselister.onHeaders(headers);
    }

    @Override
    public void onMessage(RespT message) {
        responselister.onMessage(message);
    }

    @Override
    public void onClose(Status status, Metadata
trailers) {
        // TODO: This must be fixed somehow either in
Jetty (WireMock) or somewhere else
        responselister.onClose(Status.OK, trailers);
    }

    @Override
    public void onReady() {
        responselister.onReady();
    }
};
call.start(listener, headers);
}

@Override
public void request(int numMessages) {
    call.request(numMessages);
}

@Override
public void cancel(@Nullable String message, @Nullable
Throwable cause) {
    call.cancel(message, cause);
}

@Override
public void halfClose() {
    call.halfClose();
}

@Override
public void sendMessage(ReqT message) {
    call.sendMessage(message);
}
};
}
};
}
}

```

```
}  
}
```

3.4. Messaging

Spring Cloud Contract lets you verify applications that use messaging as a means of communication. All of the integrations shown in this document work with Spring, but you can also create one of your own and use that.

3.4.1. Messaging DSL Top-level Elements

The DSL for messaging looks a little bit different than the one that focuses on HTTP. The following sections explain the differences:

- [Output Triggered by a Method](#)
- [Consumer/Producer](#)
- [Common](#)

Output Triggered by a Method

The output message can be triggered by calling a method (such as a `Scheduler` when a contract was started and when a message was sent), as shown in the following example:

Groovy

```
def dsl = Contract.make {
    // Human readable description
    description 'Some description'
    // Label by means of which the output message can be triggered
    label 'some_label'
    // input to the contract
    input {
        // the contract will be triggered by a method
        triggeredBy('bookReturnedTriggered()')
    }
    // output message of the contract
    outputMessage {
        // destination to which the output message will be sent
        sentTo('output')
        // the body of the output message
        body('{ "bookName" : "foo" }')
        // the headers of the output message
        headers {
            header('BOOK-NAME', 'foo')
        }
    }
}
```

YAML

```
# Human readable description
description: Some description
# Label by means of which the output message can be triggered
label: some_label
input:
  # the contract will be triggered by a method
  triggeredBy: bookReturnedTriggered()
# output message of the contract
outputMessage:
  # destination to which the output message will be sent
  sentTo: output
  # the body of the output message
  body:
    bookName: foo
  # the headers of the output message
  headers:
    BOOK-NAME: foo
```

In the previous example case, the output message is sent to `output` if a method called `bookReturnedTriggered` is invoked. On the message publisher's side, we generate a test that calls that method to trigger the message. On the consumer side, you can use `some_label` to trigger the

message.

Consumer/Producer



This section is valid only for the Groovy DSL.

In HTTP, you have a notion of `client/stub` and ``server/test` notation. You can also use those paradigms in messaging. In addition, Spring Cloud Contract Verifier also provides the `consumer` and `producer` methods (note that you can use either `$` or `value` methods to provide `consumer` and `producer` parts).

Common

In the `input` or `outputMessage` section, you can call `assertThat` with the name of a `method` (for example, `assertThatMessageIsOnTheQueue()`) that you have defined in the base class or in a static import. Spring Cloud Contract runs that method in the generated test.

3.4.2. Integrations

You can use one of the following integration configurations:

- Apache Camel
- Spring Integration
- Spring Cloud Stream
- Spring JMS

Since we use Spring Boot, if you have added one of these libraries to the classpath, all the messaging configuration is automatically set up.



Remember to put `@AutoConfigureMessageVerifier` on the base class of your generated tests. Otherwise, the messaging part of Spring Cloud Contract does not work.

If you want to use Spring Cloud Stream, remember to add a test dependency on `org.springframework.cloud:spring-cloud-stream`, as follows:

Maven

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-stream</artifactId>
  <type>test-jar</type>
  <scope>test</scope>
  <classifier>test-binder</classifier>
</dependency>
```

Gradle

```
testImplementation(group: 'org.springframework.cloud', name:
'spring-cloud-stream', classifier: 'test-binder')
```

Manual Integration Testing

The main interface used by the tests is `org.springframework.cloud.contract.verifier.messaging.MessageVerifierSender` and `org.springframework.cloud.contract.verifier.messaging.MessageVerifierReceiver`. It defines how to send and receive messages.

In a test, you can inject a `ContractVerifierMessageExchange` to send and receive messages that follow the contract. Then add `@AutoConfigureMessageVerifier` to your test. The following example shows how to do so:

```
@RunWith(SpringTestRunner.class)
@SpringBootTest
@AutoConfigureMessageVerifier
public static class MessagingContractTests {

    @Autowired
    private MessageVerifier verifier;
    ...
}
```



If your tests require stubs as well, then `@AutoConfigureStubRunner` includes the messaging configuration, so you only need the one annotation.

3.4.3. Producer Side Messaging Test Generation

Having the `input` or `outputMessage` sections in your DSL results in creation of tests on the publisher's side. By default, JUnit 4 tests are created. However, there is also a possibility to create JUnit 5, TestNG, or Spock tests.



The destination passed to `messageFrom` or `sentTo` can have different meanings for different messaging implementations. For Stream and Integration, it is first resolved as a `destination` of a channel. Then, if there is no such `destination`, it is resolved as a channel name. For Camel, that's a certain component (for example, `jms`).

Consider the following contract:

Groovy

```
def contractDsl = Contract.make {
  name "foo"
  label 'some_label'
  input {
    triggeredBy('bookReturnedTriggered()')
  }
  outputMessage {
    sentTo('activemq:output')
    body('{ "bookName" : "foo" }')
    headers {
      header('BOOK-NAME', 'foo')
      messagingContentType(applicationJson())
    }
  }
}
```

YAML

```
label: some_label
input:
  triggeredBy: bookReturnedTriggered
outputMessage:
  sentTo: activemq:output
  body:
    bookName: foo
  headers:
    BOOK-NAME: foo
  contentType: application/json
```

For the preceding example, the following test would be created:

JUnit

```
package com.example;

import com.jayway.jsonpath.DocumentContext;
import com.jayway.jsonpath.JsonPath;
import org.junit.Test;
import org.junit.Rule;
import javax.inject.Inject;
import
org.springframework.cloud.contract.verifier.messaging.internal.ContractVerifierObj
ectMapper;
import
org.springframework.cloud.contract.verifier.messaging.internal.ContractVerifierMes
sage;
import
org.springframework.cloud.contract.verifier.messaging.internal.ContractVerifierMes
saging;

import static
org.springframework.cloud.contract.verifier.assertion.SpringCloudContractAssertion
s.assertThat;
import static
org.springframework.cloud.contract.verifier.util.ContractVerifierUtil.*;
import static com.toomuchcoding.jsonassert.JsonAssertion.assertThatJson;
import static
org.springframework.cloud.contract.verifier.messaging.util.ContractVerifierMessagi
ngUtil.headers;
import static
org.springframework.cloud.contract.verifier.util.ContractVerifierUtil.fileToBytes;

@SuppressWarnings("rawtypes")
public class FooTest {
    @Inject ContractVerifierMessaging contractVerifierMessaging;
    @Inject ContractVerifierObjectMapper contractVerifierObjectMapper;

    @Test
    public void validate_foo() throws Exception {
        // when:
        bookReturnedTriggered();

        // then:
        ContractVerifierMessage response =
contractVerifierMessaging.receive("activemq:output",
contract(this, "foo.yml"));
assertThat(response).isNotNull();

        // and:
assertThat(response.getHeader("BOOK-NAME")).isNotNull();
assertThat(response.getHeader("BOOK-
```

```

NAME").toString()).isEqualTo("foo");
        assertThat(response.getHeader("contentType")).isNotNull();

assertThat(response.getHeader("contentType").toString()).isEqualTo("application/js
on");

        // and:
        DocumentContext parsedJson =
JsonPath.parse(contractVerifierObjectMapper.writeValueAsString(response.getPayload
()));
        assertThatJson(parsedJson).field("[ 'bookName' ]").isEqualTo("foo");
    }
}

```

Spock

```

package com.example

import com.jayway.jsonpath.DocumentContext
import com.jayway.jsonpath.JsonPath
import spock.lang.Specification
import javax.inject.Inject
import
org.springframework.cloud.contract.verifier.messaging.internal.ContractVerifierObj
ectMapper
import
org.springframework.cloud.contract.verifier.messaging.internal.ContractVerifierMes
sage
import
org.springframework.cloud.contract.verifier.messaging.internal.ContractVerifierMes
saging

import static
org.springframework.cloud.contract.verifier.assertion.SpringCloudContractAssertion
s.assertThat
import static
org.springframework.cloud.contract.verifier.util.ContractVerifierUtil.*
import static com.toomuchcoding.jsonassert.JsonAssertion.assertThatJson
import static
org.springframework.cloud.contract.verifier.messaging.util.ContractVerifierMessagi
ngUtil.headers
import static
org.springframework.cloud.contract.verifier.util.ContractVerifierUtil.fileToBytes

@SuppressWarnings("rawtypes")
class FooSpec extends Specification {
    @Inject ContractVerifierMessaging contractVerifierMessaging
    @Inject ContractVerifierObjectMapper contractVerifierObjectMapper

    def validate_foo() throws Exception {

```

```

when:
    bookReturnedTriggered()

then:
    ContractVerifierMessage response =
contractVerifierMessaging.receive("activemq:output",
    contract(this, "foo.yml"))
    response != null

and:
    response.getHeader("BOOK-NAME") != null
    response.getHeader("BOOK-NAME").toString() == 'foo'
    response.getHeader("contentType") != null
    response.getHeader("contentType").toString() == 'application/json'

and:
    DocumentContext parsedJson =
JsonPath.parse(contractVerifierObjectMapper.writeValueAsString(response.getPayload
()))
        assertThatJson(parsedJson).field("[ 'bookName' ]").isEqualTo("foo")
    }
}

```

3.4.4. Consumer Stub Generation

Unlike in the HTTP part, in messaging, we need to publish the contract definition inside the JAR with a stub. Then it is parsed on the consumer side, and proper stubbed routes are created.



If you have multiple frameworks on the classpath, Stub Runner needs to define which one should be used. Assume that you have AMQP, Spring Cloud Stream, and Spring Integration on the classpath and that you want to use Spring AMQP. Then you need to set `stubrunner.stream.enabled=false` and `stubrunner.integration.enabled=false`. That way, the only remaining framework is Spring AMQP.

Stub triggering

To trigger a message, use the `StubTrigger` interface, as the following example shows:

```

package org.springframework.cloud.contract.stubrunner;

import java.util.Collection;
import java.util.Map;

/**
 * Contract for triggering stub messages.
 *
 * @author Marcin Grzejszczak
 */
public interface StubTrigger {

    /**
     * Triggers an event by a given label for a given {@code groupid:artifactid}
     notation.
     * You can use only {@code artifactId} too.
     *
     * Feature related to messaging.
     * @param ivyNotation ivy notation of a stub
     * @param labelName name of the label to trigger
     * @return true - if managed to run a trigger
     */
    boolean trigger(String ivyNotation, String labelName);

    /**
     * Triggers an event by a given label.
     *
     * Feature related to messaging.
     * @param labelName name of the label to trigger
     * @return true - if managed to run a trigger
     */
    boolean trigger(String labelName);

    /**
     * Triggers all possible events.
     *
     * Feature related to messaging.
     * @return true - if managed to run a trigger
     */
    boolean trigger();

    /**
     * Feature related to messaging.
     * @return a mapping of ivy notation of a dependency to all the labels it has.
     */
    Map<String, Collection<String>> labels();
}

```

For convenience, the `StubFinder` interface extends `StubTrigger`, so you need only one or the other in your tests.

`StubTrigger` gives you the following options to trigger a message:

- [Trigger by Label](#)
- [Trigger by Group and Artifact IDs](#)
- [Trigger by Artifact IDs](#)
- [Trigger All Messages](#)

Trigger by Label

The following example shows how to trigger a message with a label:

```
stubFinder.trigger('return_book_1')
```

Trigger by Group and Artifact IDs

The following example shows how to trigger a message by group and artifact IDs:

```
stubFinder.trigger('org.springframework.cloud.contract.verifier.stubs:streamService', 'return_book_1')
```

Trigger by Artifact IDs

The following example shows how to trigger a message from artifact IDs:

```
stubFinder.trigger('streamService', 'return_book_1')
```

Trigger All Messages

The following example shows how to trigger all messages:

```
stubFinder.trigger()
```

3.4.5. Consumer Side Messaging With Apache Camel

Spring Cloud Contract Stub Runner's messaging module gives you an easy way to integrate with Apache Camel. For the provided artifacts, it automatically downloads the stubs and registers the required routes.

Adding Apache Camel to the Project

You can have both Apache Camel and Spring Cloud Contract Stub Runner on the classpath. Remember to annotate your test class with `@AutoConfigureStubRunner`.

Disabling the Functionality

If you need to disable this functionality, set the `stubrunner.camel.enabled=false` property.

Examples

Assume that we have the following Maven repository with deployed stubs for the `camelService` application:

```
├── .m2
│   ├── repository
│   │   ├── io
│   │   │   ├── codearte
│   │   │   │   ├── accurest
│   │   │   │   │   ├── stubs
│   │   │   │   │   │   ├── camelService
│   │   │   │   │   │   │   ├── 0.0.1-SNAPSHOT
│   │   │   │   │   │   │   │   ├── camelService-0.0.1-SNAPSHOT.pom
│   │   │   │   │   │   │   │   ├── camelService-0.0.1-SNAPSHOT-stubs.jar
│   │   │   │   │   │   │   │   ├── maven-metadata-local.xml
│   │   │   │   │   │   │   └── maven-metadata-local.xml
```

Further, assume that the stubs contain the following structure:

```
├── META-INF
│   └── MANIFEST.MF
├── repository
│   ├── accurest
│   │   └── bookReturned1.groovy
│   └── mappings
```

Now consider the following contract:

```
Contract.make {
  label 'return_book_1'
  input {
    triggeredBy('bookReturnedTriggered()')
  }
  outputMessage {
    sentTo('rabbitmq:output?queue=output')
    body(''{ "bookName" : "foo" }'')
    headers {
      header('BOOK-NAME', 'foo')
    }
  }
}
```

To trigger a message from the `return_book_1` label, we use the `StubTrigger` interface, as follows:

```
stubFinder.trigger("return_book_1")
```

That will send out a message to the destination described in the output message of the contract.

3.4.6. Consumer Side Messaging with Spring Integration

Spring Cloud Contract Stub Runner's messaging module gives you an easy way to integrate with Spring Integration. For the provided artifacts, it automatically downloads the stubs and registers the required routes.

Adding the Runner to the Project

You can have both Spring Integration and Spring Cloud Contract Stub Runner on the classpath. Remember to annotate your test class with `@AutoConfigureStubRunner`.

Disabling the Functionality

If you need to disable this functionality, set the `stubrunner.integration.enabled=false` property.

Examples

Assume that you have the following Maven repository with deployed stubs for the `integrationService` application:

```

└── .m2
    ├── repository
    │   ├── io
    │   │   ├── codearte
    │   │   │   ├── accurest
    │   │   │   │   ├── stubs
    │   │   │   │   │   ├── integrationService
    │   │   │   │   │   │   ├── 0.0.1-SNAPSHOT
    │   │   │   │   │   │   │   ├── integrationService-0.0.1-SNAPSHOT.pom
    │   │   │   │   │   │   │   ├── integrationService-0.0.1-SNAPSHOT-
    │   │   │   │   │   │   │   └── maven-metadata-local.xml
    │   │   │   │   │   └── maven-metadata-local.xml
    │   │   └── stubs.jar
    └── stubs.jar

```

Further assume the stubs contain the following structure:

```

└── META-INF
    ├── MANIFEST.MF
    ├── repository
    │   ├── accurest
    │   │   ├── bookReturned1.groovy
    │   └── mappings

```

Consider the following contract:

```

Contract.make {
    label 'return_book_1'
    input {
        triggeredBy('bookReturnedTriggered()')
    }
    outputMessage {
        sentTo('output')
        body('''{ "bookName" : "foo" }''')
        headers {
            header('BOOK-NAME', 'foo')
        }
    }
}

```

Now consider the following Spring Integration Route:


```

<?xml version="1.0" encoding="UTF-8"?>
<beans:beans xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:beans="http://www.springframework.org/schema/beans"
  xmlns="http://www.springframework.org/schema/integration"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
    https://www.springframework.org/schema/beans/spring-beans.xsd
    http://www.springframework.org/schema/integration
    http://www.springframework.org/schema/integration/spring-
integration.xsd">

  <!-- REQUIRED FOR TESTING -->
  <bridge input-channel="output"
    output-channel="outputTest"/>

  <channel id="outputTest">
    <queue/>
  </channel>

</beans:beans>

```

To trigger a message from the `return_book_1` label, use the `StubTrigger` interface, as follows:

```
stubFinder.trigger('return_book_1')
```

That will send out a message to the destination described in the output message of the contract.

3.4.7. Consumer Side Messaging With Spring Cloud Stream

Spring Cloud Contract Stub Runner's messaging module gives you an easy way to integrate with Spring Stream. For the provided artifacts, it automatically downloads the stubs and registers the required routes.



If Stub Runner's integration with the Stream `messageFrom` or `sentTo` strings are resolved first as the `destination` of a channel and no such `destination` exists, the destination is resolved as a channel name.

If you want to use Spring Cloud Stream, remember to add a dependency on `org.springframework.cloud:spring-cloud-stream-test-support`, as follows:



Maven

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-stream-test-binder</artifactId>
  <scope>test</scope>
</dependency>
```

Gradle

```
testImplementation('org.springframework.cloud:spring-cloud-stream-
test-binder')
```

Adding the Runner to the Project

You can have both Spring Cloud Stream and Spring Cloud Contract Stub Runner on the classpath. Remember to annotate your test class with `@AutoConfigureStubRunner`.

Disabling the Functionality

If you need to disable this functionality, set the `stubrunner.stream.enabled=false` property.

Examples

Assume that you have the following Maven repository with deployed stubs for the `streamService` application:

```
.m2
├── repository
│   ├── io
│   │   ├── codearte
│   │   │   ├── accurest
│   │   │   │   ├── stubs
│   │   │   │   │   ├── streamService
│   │   │   │   │   │   ├── 0.0.1-SNAPSHOT
│   │   │   │   │   │   │   ├── streamService-0.0.1-SNAPSHOT.pom
│   │   │   │   │   │   │   ├── streamService-0.0.1-SNAPSHOT-stubs.jar
│   │   │   │   │   │   │   ├── maven-metadata-local.xml
│   │   │   │   │   │   └── maven-metadata-local.xml
```

Further assume the stubs contain the following structure:

```

├── META-INF
│   └── MANIFEST.MF
└── repository
    ├── accurest
    │   └── bookReturned1.groovy
    └── mappings

```

Consider the following contract:

```

Contract.make {
    label 'return_book_1'
    input { triggeredBy('bookReturnedTriggered()') }
    outputMessage {
        sentTo('returnBook')
        body(''{ "bookName" : "foo" }''')
        headers { header('BOOK-NAME', 'foo') }
    }
}

```

Now consider the following Spring Cloud Stream function configuration:

```

@ImportAutoConfiguration(TestChannelBinderConfiguration.class)
@Configuration(proxyBeanMethods = true)
@EnableAutoConfiguration
protected static class Config {

    @Bean
    Function<String, String> test1() {
        return (input) -> {
            println "Test 1 [${input}]"
            return input
        }
    }
}

```

Now consider the following Spring configuration:

```
stubrunner.repositoryRoot: classpath:m2repo/repository/
stubrunner.ids:
org.springframework.cloud.contract.verifier.stubs:streamService:0.0.1-
SNAPSHOT:stubs
stubrunner.stubs-mode: remote
spring:
  cloud:
    stream:
      bindings:
        test1-in-0:
          destination: returnBook
        test1-out-0:
          destination: outputToAssertBook
      function:
        definition: test1

server:
  port: 0

debug: true
```

To trigger a message from the `return_book_1` label, use the `StubTrigger` interface as follows:

```
stubFinder.trigger('return_book_1')
```

That will send out a message to the destination described in the output message of the contract.

3.4.8. Consumer Side Messaging With Spring JMS

Spring Cloud Contract Stub Runner's messaging module provides an easy way to integrate with Spring JMS.

The integration assumes that you have a running instance of a JMS broker.

Adding the Runner to the Project

You need to have both Spring JMS and Spring Cloud Contract Stub Runner on the classpath. Remember to annotate your test class with `@AutoConfigureStubRunner`.

Examples

Assume that the stub structure looks as follows:

```
├── stubs
│   └── bookReturned1.groovy
```

Further assume the following test configuration:

```
stubrunner:
  repository-root: stubs:classpath:/stubs/
  ids: my:stubs
  stubs-mode: remote
spring:
  activemq:
    send-timeout: 1000
  jms:
    template:
      receive-timeout: 1000
```

Now consider the following contract:

```
Contract.make {
  label 'return_book_1'
  input {
    triggeredBy('bookReturnedTriggered()')
  }
  outputMessage {
    sentTo('output')
    body(''{ "bookName" : "foo" }''')
    headers {
      header('BOOKNAME', 'foo')
    }
  }
}
```

To trigger a message from the `return_book_1` label, we use the `StubTrigger` interface, as follows:

```
stubFinder.trigger('return_book_1')
```

That will send out a message to the destination described in the output message of the contract.

3.5. Spring Cloud Contract Stub Runner

One of the issues that you might encounter while using Spring Cloud Contract Verifier is passing the generated WireMock JSON stubs from the server side to the client side (or to various clients). The same takes place in terms of client-side generation for messaging.

Copying the JSON files and setting the client side for messaging manually is out of the question. That is why we introduced Spring Cloud Contract Stub Runner. It can automatically download and run the stubs for you.

3.5.1. Snapshot Versions

You can add the additional snapshot repository to your build file to use snapshot versions, which are automatically uploaded after every successful build, as follows:

Maven

```
<repositories>
  <repository>
    <id>spring-snapshots</id>
    <name>Spring Snapshots</name>
    <url>https://repo.spring.io/snapshot</url>
    <snapshots>
      <enabled>true</enabled>
    </snapshots>
  </repository>
  <repository>
    <id>spring-milestones</id>
    <name>Spring Milestones</name>
    <url>https://repo.spring.io/milestone</url>
    <snapshots>
      <enabled>>false</enabled>
    </snapshots>
  </repository>
  <repository>
    <id>spring-releases</id>
    <name>Spring Releases</name>
    <url>https://repo.spring.io/release</url>
    <snapshots>
      <enabled>>false</enabled>
    </snapshots>
  </repository>
</repositories>
<pluginRepositories>
  <pluginRepository>
    <id>spring-snapshots</id>
    <name>Spring Snapshots</name>
    <url>https://repo.spring.io/snapshot</url>
    <snapshots>
      <enabled>true</enabled>
    </snapshots>
  </pluginRepository>
  <pluginRepository>
    <id>spring-milestones</id>
    <name>Spring Milestones</name>
    <url>https://repo.spring.io/milestone</url>
    <snapshots>
      <enabled>>false</enabled>
    </snapshots>
  </pluginRepository>
  <pluginRepository>
    <id>spring-releases</id>
    <name>Spring Releases</name>
    <url>https://repo.spring.io/release</url>
    <snapshots>
```

```
        <enabled>false</enabled>
    </snapshots>
</pluginRepository>
</pluginRepositories>
```

Gradle ([settings.xml](#))

```
pluginManagement {
    repositories {
        mavenLocal()
        maven { url "https://repo.spring.io/snapshot" }
        maven { url "https://repo.spring.io/milestone" }
        maven { url "https://repo.spring.io/release" }
        gradlePluginPortal()
    }
}
```

3.5.2. Publishing Stubs as JARs

The easiest approach to publishing stubs as jars is to centralize the way stubs are kept. For example, you can keep them as jars in a Maven repository.



For both Maven and Gradle, the setup comes ready to work. However, you can customize it if you want to.

The following example shows how to publish stubs as jars:

Maven

```
<!-- First disable the default jar setup in the properties section -->
<!-- we don't want the verifier to do a jar for us -->
<spring.cloud.contract.verifier.skip>true</spring.cloud.contract.verifier.skip>
```

```
<!-- Next add the assembly plugin to your build -->
<!-- we want the assembly plugin to generate the JAR -->
<plugin>
```

```
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-assembly-plugin</artifactId>
  <executions>
    <execution>
      <id>stub</id>
      <phase>prepare-package</phase>
      <goals>
        <goal>single</goal>
      </goals>
      <inherited>>false</inherited>
      <configuration>
        <attach>true</attach>
        <descriptors>
          $/tmp/releaser-1691636970130-0/spring-cloud-
contract/docs/src/assembly/stub.xml
        </descriptors>
      </configuration>
    </execution>
  </executions>
</plugin>
```

```
<!-- Finally setup your assembly. Below you can find the contents of
src/main/assembly/stub.xml -->
```

```
<assembly
  xmlns="http://maven.apache.org/plugins/maven-assembly-plugin/assembly/1.1.3"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/plugins/maven-assembly-
plugin/assembly/1.1.3 https://maven.apache.org/xsd/assembly-1.1.3.xsd">
  <id>stubs</id>
  <formats>
    <format>jar</format>
  </formats>
  <includeBaseDirectory>>false</includeBaseDirectory>
  <fileSets>
    <fileSet>
      <directory>src/main/java</directory>
      <outputDirectory>../outputDirectory</outputDirectory>
      <includes>
        <include>**com/example/model/*.*</include>
      </includes>
    </fileSet>
```

```

<fileSet>
  <directory>${project.build.directory}/classes</directory>
  <outputDirectory></outputDirectory>
  <includes>
    <include>**com/example/model/*. *</include>
  </includes>
</fileSet>
<fileSet>
  <directory>${project.build.directory}/snippets/stubs</directory>
  <outputDirectory>META-
INF/${project.groupId}/${project.artifactId}/${project.version}/mappings</outputDi
rectory>
  <includes>
    <include>**/*</include>
  </includes>
</fileSet>
<fileSet>
  <directory>$/tmp/releaser-1691636970130-0/spring-cloud-
contract/docs/src/test/resources/contracts</directory>
  <outputDirectory>META-
INF/${project.groupId}/${project.artifactId}/${project.version}/contracts</outputD
irectory>
  <includes>
    <include>**/*.groovy</include>
  </includes>
</fileSet>
</fileSets>
</assembly>

```

Gradle

```

ext {
  contractsDir = file("mappings")
  stubsOutputDirRoot = file("${project.buildDir}/production/${project.name}-
stubs/")
}

// Automatically added by plugin:
// copyContracts - copies contracts to the output folder from which JAR will be
created
// verifierStubsJar - JAR with a provided stub suffix

publishing {
  publications {
    stubs(MavenPublication) {
      artifactId "${project.name}-stubs"
      artifact verifierStubsJar
    }
  }
}
}

```

3.5.3. Stub Runner Core

The stub runner core runs stubs for service collaborators. Treating stubs as contracts of services lets you use stub-runner as an implementation of [Consumer-driven Contracts](#).

Stub Runner lets you automatically download the stubs of the provided dependencies (or pick those from the classpath), start WireMock servers for them, and feed them with proper stub definitions. For messaging, special stub routes are defined.

Retrieving stubs

You can pick from the following options of acquiring stubs:

- Aether-based solution that downloads JARs with stubs from Artifactory or Nexus
- Classpath-scanning solution that searches the classpath with a pattern to retrieve stubs
- Writing your own implementation of the `org.springframework.cloud.contract.stubrunner.StubDownloaderBuilder` for full customization

The latter example is described in the [Custom Stub Runner](#) section.

Downloading Stubs

You can control the downloading of stubs with the `stubsMode` switch. It picks value from the `StubRunnerProperties.StubsMode` enumeration. You can use the following options:

- `StubRunnerProperties.StubsMode.CLASSPATH` (default value): Picks stubs from the classpath
- `StubRunnerProperties.StubsMode.LOCAL`: Picks stubs from a local storage (for example, `.m2`)
- `StubRunnerProperties.StubsMode.REMOTE`: Picks stubs from a remote location

The following example picks stubs from a local location:

```
@AutoConfigureStubRunner(repositoryRoot="https://foo.bar", ids =
"com.example:beer-api-producer:+:stubs:8095", stubsMode =
StubRunnerProperties.StubsMode.LOCAL)
```

Classpath scanning

If you set the `stubsMode` property to `StubRunnerProperties.StubsMode.CLASSPATH` (or set nothing since `CLASSPATH` is the default value), the classpath is scanned. Consider the following example:

```
@AutoConfigureStubRunner(ids = {
    "com.example:beer-api-producer:+:stubs:8095",
    "com.example.foo:bar:1.0.0:superstubs:8096"
})
```

You can add the dependencies to your classpath, as follows:

Maven

```
<dependency>
  <groupId>com.example</groupId>
  <artifactId>beer-api-producer-restdocs</artifactId>
  <classifier>stubs</classifier>
  <version>0.0.1-SNAPSHOT</version>
  <scope>test</scope>
  <exclusions>
    <exclusion>
      <groupId>*</groupId>
      <artifactId>*</artifactId>
    </exclusion>
  </exclusions>
</dependency>
<dependency>
  <groupId>com.example.thing1</groupId>
  <artifactId>thing2</artifactId>
  <classifier>superstubs</classifier>
  <version>1.0.0</version>
  <scope>test</scope>
  <exclusions>
    <exclusion>
      <groupId>*</groupId>
      <artifactId>*</artifactId>
    </exclusion>
  </exclusions>
</dependency>
```

Gradle

```
testCompile("com.example:beer-api-producer-restdocs:0.0.1-SNAPSHOT:stubs") {
  transitive = false
}
testCompile("com.example.thing1:thing2:1.0.0:superstubs") {
  transitive = false
}
```

Then the specified locations on your classpath get scanned. For `com.example:beer-api-producer-restdocs`, the following locations are scanned:

- `/META-INF/com.example/beer-api-producer-restdocs/*/*`
- `/contracts/com.example/beer-api-producer-restdocs/*/*`
- `/mappings/com.example/beer-api-producer-restdocs/*/*`

For `com.example.thing1:thing2`, the following locations are scanned:

- `/META-INF/com.example.thing1/thing2/*/*`
- `/contracts/com.example.thing1/thing2/*/*`
- `/mappings/com.example.thing1/thing2/*/*`



You have to explicitly provide the group and artifact IDs when you package the producer stubs.

To achieve proper stub packaging, the producer would set up the contracts as follows:

```
├── src
│   └── test
│       ├── resources
│       │   ├── contracts
│       │   │   ├── com.example
│       │   │   │   ├── beer-api-producer-restdocs
│       │   │   │   │   ├── nested
│       │   │   │   │   └── contract3.groovy
```

By using the [Maven assembly plugin](#) or the [Gradle Jar](#) task, you have to create the following structure in your stubs jar:

```
├── META-INF
│   └── com.example
│       ├── beer-api-producer-restdocs
│       │   ├── 2.0.0
│       │   │   ├── contracts
│       │   │   │   ├── nested
│       │   │   │   └── contract2.groovy
│       │   ├── mappings
│       │   └── mapping.json
```

By maintaining this structure, the classpath gets scanned and you can profit from the messaging or HTTP stubs without the need to download artifacts.

Configuring HTTP Server Stubs

Stub Runner has a notion of a `HttpServerStub` that abstracts the underlying concrete implementation of the HTTP server (for example, WireMock is one of the implementations). Sometimes, you need to perform some additional tuning (which is concrete for the given implementation) of the stub servers. To do that, Stub Runner gives you the `httpServerStubConfigurer` property that is available in the annotation and the JUnit rule and is accessible through system properties, where you can provide your implementation of the

`org.springframework.cloud.contract.stubrunner.HttpServerStubConfigurer` interface. The implementations can alter the configuration files for the given HTTP server stub.

Spring Cloud Contract Stub Runner comes with an implementation that you can extend for WireMock:

`org.springframework.cloud.contract.stubrunner.provider.wiremock.WireMockHttpServerStubConfigurer`. In the `configure` method, you can provide your own custom configuration for the given stub. The use case might be starting WireMock for the given artifact ID, on an HTTPS port. The following example shows how to do so:

Example 4. WireMockHttpServerStubConfigurer implementation

```
@CompileStatic
static class HttpsForFraudDetection extends WireMockHttpServerStubConfigurer {

    private static final Log log = LogFactory.getLog(HttpsForFraudDetection)

    @Override
    WireMockConfiguration configure(WireMockConfiguration httpStubConfiguration,
    HttpServerStubConfiguration httpServerStubConfiguration) {
        if (httpServerStubConfiguration.stubConfiguration.artifactId ==
"fraudDetectionServer") {
            int httpsPort = TestSocketUtils.findAvailableTcpPort()
            log.info("Will set HTTPS port [" + httpsPort + "] for fraud detection
server")
            return httpStubConfiguration
                .httpsPort(httpsPort)
        }
        return httpStubConfiguration
    }
}
```

You can then reuse it with the `@AutoConfigureStubRunner` annotation, as follows:

```
@AutoConfigureStubRunner(mappingsOutputFolder = "target/outputmappings/",
    httpServerStubConfigurer = HttpsForFraudDetection)
```

Whenever an HTTPS port is found, it takes precedence over the HTTP port.

Running stubs

This section describes how to run stubs. It contains the following topics:

- [HTTP Stubs](#)
- [Viewing Registered Mappings](#)

- [Messaging Stubs](#)

HTTP Stubs

Stubs are defined in JSON documents, whose syntax is defined in the [WireMock documentation](#).

The following example defines a stub in JSON:

```
{
  "request": {
    "method": "GET",
    "url": "/ping"
  },
  "response": {
    "status": 200,
    "body": "pong",
    "headers": {
      "Content-Type": "text/plain"
    }
  }
}
```

Viewing Registered Mappings

Every stubbed collaborator exposes a list of defined mappings under the `__/admin/` endpoint.

You can also use the `mappingsOutputFolder` property to dump the mappings to files. For the annotation-based approach, it would resemble the following example:

```
@AutoConfigureStubRunner(ids="a.b.c:loanIssuance,a.b.c:fraudDetectionServer",
mappingsOutputFolder = "target/outputmappings/")
```

For the JUnit approach, it resembles the following example:

```
@ClassRule @Shared StubRunnerRule rule = new StubRunnerRule()
    .repoRoot("https://some_url")
    .downloadStub("a.b.c", "loanIssuance")
    .downloadStub("a.b.c:fraudDetectionServer")
    .withMappingsOutputFolder("target/outputmappings")
```

Then, if you check out the `target/outputmappings` folder, you would see the following structure;

```
.
├── fraudDetectionServer_13705
└── loanIssuance_12255
```

That means that there were two stubs registered. `fraudDetectionServer` was registered at port `13705` and `loanIssuance` at port `12255`. If we take a look at one of the files, we would see (for WireMock) the mappings available for the given server:

```
[{
  "id" : "f9152eb9-bf77-4c38-8289-90be7d10d0d7",
  "request" : {
    "url" : "/name",
    "method" : "GET"
  },
  "response" : {
    "status" : 200,
    "body" : "fraudDetectionServer"
  },
  "uuid" : "f9152eb9-bf77-4c38-8289-90be7d10d0d7"
},
...
]
```

Messaging Stubs

Depending on the provided Stub Runner dependency and the DSL, the messaging routes are automatically set up.

3.5.4. Stub Runner JUnit Rule and Stub Runner JUnit5 Extension

Stub Runner comes with a JUnit rule that lets you can download and run stubs for a given group and artifact ID, as the following example shows:


```

@ClassRule
public static StubRunnerRule rule = new StubRunnerRule().repoRoot(repoRoot())
    .stubsMode(StubRunnerProperties.StubsMode.REMOTE)
    .downloadStub("org.springframework.cloud.contract.verifier.stubs",
"loanIssuance")

    .downloadStub("org.springframework.cloud.contract.verifier.stubs:fraudDetectionSer
ver");

@BeforeClass
@AfterClass
public static void setupProps() {
    System.clearProperty("stubrunner.repository.root");
    System.clearProperty("stubrunner.classifier");
}

```

A `StubRunnerExtension` is also available for JUnit 5. `StubRunnerRule` and `StubRunnerExtension` work in a very similar fashion. After the rule or extension is called, Stub Runner connects to your Maven repository and, for the given list of dependencies, tries to:

- Download them
- Cache them locally
- Unzip them to a temporary folder
- Start a WireMock server for each Maven dependency on a random port from the provided range of ports or the provided port
- Feed the WireMock server with all JSON files that are valid WireMock definitions
- Send messages (remember to pass an implementation of `MessageVerifierSender` interface)

Stub Runner uses the [Eclipse Aether](#) mechanism to download the Maven dependencies. Check their [docs](#) for more information.

Since the `StubRunnerRule` and `StubRunnerExtension` implement the `StubFinder`, they let you find the started stubs, as the following example shows:

```

package org.springframework.cloud.contract.stubrunner;

import java.net.URL;
import java.util.Collection;
import java.util.Map;

import org.springframework.cloud.contract.spec.Contract;

/**
 * Contract for finding registered stubs.
 *
 * @author Marcin Grzejszczak
 */
public interface StubFinder extends StubTrigger {

    /**
     * For the given groupId and artifactId tries to find the matching URL of the
     running
     * stub.
     * @param groupId - might be null. In that case a search only via artifactId
     takes
     * place
     * @param artifactId - artifact id of the stub
     * @return URL of a running stub or throws exception if not found
     * @throws StubNotFoundException in case of not finding a stub
     */
    URL findStubUrl(String groupId, String artifactId) throws
    StubNotFoundException;

    /**
     * For the given Ivy notation {@code
    [groupId]:artifactId:[version]:[classifier]}
     * tries to find the matching URL of the running stub. You can also pass only
     * {@code artifactId}.
     * @param ivyNotation - Ivy representation of the Maven artifact
     * @return URL of a running stub or throws exception if not found
     * @throws StubNotFoundException in case of not finding a stub
     */
    URL findStubUrl(String ivyNotation) throws StubNotFoundException;

    /**
     * @return all running stubs
     */
    RunningStubs findAllRunningStubs();

    /**
     * @return the list of Contracts
     */
    Map<StubConfiguration, Collection<Contract>> getContracts();

```

```
}
```

The following examples provide more detail about using Stub Runner:

Spock

```
@ClassRule
@Shared
StubRunnerRule rule = new StubRunnerRule()
    .stubsMode(StubRunnerProperties.StubsMode.REMOTE)

.repoRoot(StubRunnerRuleSpec.getResource("/m2repo/repository").toURI().toString())
    .downloadStub("org.springframework.cloud.contract.verifier.stubs",
"loanIssuance")

.downloadStub("org.springframework.cloud.contract.verifier.stubs:fraudDetectionServer")
    .withMappingsOutputFolder("target/outputmappingsforrule")

def 'should start WireMock servers'() {
    expect: 'WireMocks are running'
        rule.findStubUrl('org.springframework.cloud.contract.verifier.stubs',
'loanIssuance') != null
        rule.findStubUrl('loanIssuance') != null
        rule.findStubUrl('loanIssuance') ==
rule.findStubUrl('org.springframework.cloud.contract.verifier.stubs',
'loanIssuance')

rule.findStubUrl('org.springframework.cloud.contract.verifier.stubs:fraudDetection
Server') != null
    and:
        rule.findAllRunningStubs().isPresent('loanIssuance')

rule.findAllRunningStubs().isPresent('org.springframework.cloud.contract.verifier.
stubs', 'fraudDetectionServer')

rule.findAllRunningStubs().isPresent('org.springframework.cloud.contract.verifier.
stubs:fraudDetectionServer')
    and: 'Stubs were registered'
        "${rule.findStubUrl('loanIssuance').toString()}/name".toURL().text ==
'loanIssuance'
        "${rule.findStubUrl('fraudDetectionServer').toString()}/name".toURL().text
== 'fraudDetectionServer'
}

def 'should output mappings to output folder'() {
    when:
        def url = rule.findStubUrl('fraudDetectionServer')
    then:
        new File("target/outputmappingsforrule",
"fraudDetectionServer_${url.port}").exists()
}
```

JUnit 4

```
@Test
public void should_start_wiremock_servers() throws Exception {
    // expect: 'WireMocks are running'
    then(rule.findStubUrl("org.springframework.cloud.contract.verifier.stubs",
"loanIssuance")).isNotNull();
    then(rule.findStubUrl("loanIssuance")).isNotNull();
    then(rule.findStubUrl("loanIssuance"))

.isEqualTo(rule.findStubUrl("org.springframework.cloud.contract.verifier.stubs",
"loanIssuance"));

then(rule.findStubUrl("org.springframework.cloud.contract.verifier.stubs:fraudDete
ctionServer")).isNotNull();
    // and:
    then(rule.findAllRunningStubs().isPresent("loanIssuance")).isTrue();

then(rule.findAllRunningStubs().isPresent("org.springframework.cloud.contract.veri
fier.stubs",
    "fraudDetectionServer")).isTrue();
    then(rule.findAllRunningStubs()

.isPresent("org.springframework.cloud.contract.verifier.stubs:fraudDetectionServer
")).isTrue();
    // and: 'Stubs were registered'
    then(httpGet(rule.findStubUrl("loanIssuance").toString() +
"/name")).isEqualTo("loanIssuance");
    then(httpGet(rule.findStubUrl("fraudDetectionServer").toString() +
"/name")).isEqualTo("fraudDetectionServer");
}
```

JUnit 5

```
// Visible for JUnit
@RegisterExtension
static StubRunnerExtension stubRunnerExtension = new
StubRunnerExtension().repoRoot(repoRoot())
    .stubsMode(StubRunnerProperties.StubsMode.REMOTE)
    .downloadStub("org.springframework.cloud.contract.verifier.stubs",
"loanIssuance")

.downloadStub("org.springframework.cloud.contract.verifier.stubs:fraudDetectionSer
ver")
    .withMappingsOutputFolder("target/outputmappingsforrule");

@BeforeAll
@AfterAll
static void setupProps() {
    System.clearProperty("stubrunner.repository.root");
    System.clearProperty("stubrunner.classifier");
}

private static String repoRoot() {
    try {
        return
StubRunnerRuleJUnitTest.class.getResource("/m2repo/repository/").toURI().toString(
);
    }
    catch (Exception e) {
        return "";
    }
}
```

See the [Common Properties for JUnit and Spring](#) for more information on how to apply global configuration of Stub Runner.



To use the JUnit rule or JUnit 5 extension together with messaging, you have to provide an implementation of the `MessageVerifierSender` and `MessageVerifierReceiver` interface to the rule builder (for example, `rule.messageVerifierSender(new MyMessageVerifierSender())`). If you do not do this, then, whenever you try to send a message, an exception is thrown.

Maven Settings

The stub downloader honors Maven settings for a different local repository folder. Authentication details for repositories and profiles are currently not taken into account, so you need to specify it by using the properties mentioned above.

Providing Fixed Ports

You can also run your stubs on fixed ports. You can do it in two different ways. One is to pass it in the properties, and the other is to use the fluent API of JUnit rule.

Fluent API

When using the `StubRunnerRule` or `StubRunnerExtension`, you can add a stub to download and then pass the port for the last downloaded stub. The following example shows how to do so:

```
@ClassRule
public static StubRunnerRule rule = new StubRunnerRule().repoRoot(repoRoot())
    .stubsMode(StubRunnerProperties.StubsMode.REMOTE)
    .downloadStub("org.springframework.cloud.contract.verifier.stubs",
"loanIssuance").withPort(35465)

.downloadStub("org.springframework.cloud.contract.verifier.stubs:fraudDetectionServer:35466");

@BeforeClass
@AfterClass
public static void setupProps() {
    System.clearProperty("stubrunner.repository.root");
    System.clearProperty("stubrunner.classifier");
}
```

For the preceding example, the following test is valid:

```
then(rule.findStubUrl("loanIssuance")).isEqualTo(URI.create("http://localhost:35465").toURL());
then(rule.findStubUrl("fraudDetectionServer")).isEqualTo(URI.create("http://localhost:35466").toURL());
```

Stub Runner with Spring

Stub Runner with Spring sets up Spring configuration of the Stub Runner project.

By providing a list of stubs inside your configuration file, Stub Runner automatically downloads and registers in WireMock the selected stubs.

If you want to find the URL of your stubbed dependency, you can autowire the `StubFinder` interface and use its methods, as follows:

```

@SpringBootTest(classes = Config, properties = [" stubrunner.cloud.enabled=false",
    'foo=${stubrunner.runningstubs.fraudDetectionServer.port}',

    'fooWithGroup=${stubrunner.runningstubs.org.springframework.cloud.contract.verifier.stubs.fraudDetectionServer.port}'])
@AutoConfigureStubRunner(mappingsOutputFolder = "target/outputmappings/",
    httpServerStubConfigurer = HttpsForFraudDetection)
@ActiveProfiles("test")
class StubRunnerConfigurationSpec {

    @Autowired
    StubFinder stubFinder
    @Autowired
    Environment environment
    @StubRunnerPort("fraudDetectionServer")
    int fraudDetectionServerPort

    @StubRunnerPort("org.springframework.cloud.contract.verifier.stubs:fraudDetectionServer")
    int fraudDetectionServerPortWithGroupId
    @Value('${foo}')
    Integer foo

    @BeforeAll
    static void setupSpec() {
        System.clearProperty("stubrunner.repository.root")
        System.clearProperty("stubrunner.classifier")
        WireMockHttpServerStubAccessor.clear()
    }

    @AfterAll
    static void cleanupSpec() {
        setupSpec()
    }

    @Test
    void 'should mark all ports as random'() {
        expect:
        WireMockHttpServerStubAccessor.everyPortRandom()
    }

    @Test
    void 'should start WireMock servers'() {
        expect: 'WireMocks are running'
        assert
        stubFinder.findStubUrl('org.springframework.cloud.contract.verifier.stubs',
            'loanIssuance') != null
        assert stubFinder.findStubUrl('loanIssuance') != null
        assert stubFinder.findStubUrl('loanIssuance') ==

```



```

stubFinder.findStubUrl('org.springframework.cloud.contract.verifier.stubs',
'loanIssuance')
    assert stubFinder.findStubUrl('loanIssuance') ==
stubFinder.findStubUrl('org.springframework.cloud.contract.verifier.stubs:loanIssuance')
    assert
stubFinder.findStubUrl('org.springframework.cloud.contract.verifier.stubs:loanIssuance:0.0.1-SNAPSHOT') ==
stubFinder.findStubUrl('org.springframework.cloud.contract.verifier.stubs:loanIssuance:0.0.1-SNAPSHOT:stubs')
    assert
stubFinder.findStubUrl('org.springframework.cloud.contract.verifier.stubs:fraudDetectionServer') != null
    and:
    assert stubFinder.findAllRunningStubs().isPresent('loanIssuance')
    assert
stubFinder.findAllRunningStubs().isPresent('org.springframework.cloud.contract.verifier.stubs', 'fraudDetectionServer')
    assert
stubFinder.findAllRunningStubs().isPresent('org.springframework.cloud.contract.verifier.stubs:fraudDetectionServer')
    and: 'Stubs were registered'
    assert
"${stubFinder.findStubUrl('loanIssuance').toString()}/name".toURL().text ==
'loanIssuance'
    assert
"${stubFinder.findStubUrl('fraudDetectionServer').toString()}/name".toURL().text == 'fraudDetectionServer'
    and: 'Fraud Detection is an HTTPS endpoint'
    assert
stubFinder.findStubUrl('fraudDetectionServer').toString().startsWith("https")
}

@Test
void 'should throw an exception when stub is not found'() {
    when:
        BDDAssertions.thenThrownBy(() ->
stubFinder.findStubUrl('nonExistingService')).isInstanceOf(StubNotFoundException)
    when:
        BDDAssertions.thenThrownBy(() ->
stubFinder.findStubUrl('nonExistingGroupId', 'nonExistingArtifactId'))
        .isInstanceOf(StubNotFoundException)
}

@Test
void 'should register started servers as environment variables'() {
    expect:
    assert
environment.getProperty("stubrunner.runningstubs.loanIssuance.port") != null
    assert stubFinder.findAllRunningStubs().getPort("loanIssuance") ==
(environment.getProperty("stubrunner.runningstubs.loanIssuance.port") as Integer)
}

```

```

        and:
        assert
environment.getProperty("stubrunner.runningstubs.fraudDetectionServer.port") !=
null
        assert stubFinder.findAllRunningStubs().getPort("fraudDetectionServer") ==
(environment.getProperty("stubrunner.runningstubs.fraudDetectionServer.port") as
Integer)
        and:
        assert
environment.getProperty("stubrunner.runningstubs.fraudDetectionServer.port") !=
null
        assert stubFinder.findAllRunningStubs().getPort("fraudDetectionServer") ==
(environment.getProperty("stubrunner.runningstubs.org.springframework.cloud.contra
ct.verifier.stubs.fraudDetectionServer.port") as Integer)
    }

    @Test
    void 'should be able to interpolate a running stub in the passed test
property'() {
        given:
        int fraudPort =
stubFinder.findAllRunningStubs().getPort("fraudDetectionServer")
        expect:
        assert fraudPort > 0
        assert environment.getProperty("foo", Integer) == fraudPort
        assert environment.getProperty("fooWithGroup", Integer) == fraudPort
        assert foo == fraudPort
    }

    // @Issue("#573")
    @Test
    void 'should be able to retrieve the port of a running stub via an
annotation'() {
        given:
        int fraudPort =
stubFinder.findAllRunningStubs().getPort("fraudDetectionServer")
        expect:
        assert fraudPort > 0
        assert fraudDetectionServerPort == fraudPort
        assert fraudDetectionServerPortWithGroupId == fraudPort
    }

    @Test
    void 'should dump all mappings to a file'() {
        when:
        def url = stubFinder.findStubUrl("fraudDetectionServer")
        then:
        assert new File("target/outputmappings/",
"fraudDetectionServer_${url.port}").exists()
    }

```

```

@Configuration
@EnableAutoConfiguration
static class Config {}

@CompileStatic
static class HttpsForFraudDetection extends WireMockHttpServerStubConfigurer {

    private static final Log log = LogFactory.getLog(HttpsForFraudDetection)

    @Override
    WireMockConfiguration configure(WireMockConfiguration
httpStubConfiguration, HttpServerStubConfiguration httpServerStubConfiguration) {
        if (httpServerStubConfiguration.stubConfiguration.artifactId ==
"fraudDetectionServer") {
            int httpsPort = TestSocketUtils.findAvailableTcpPort()
            log.info("Will set HTTPS port [" + httpsPort + "] for fraud
detection server")
            return httpStubConfiguration
                .httpsPort(httpsPort)
        }
        return httpStubConfiguration
    }
}
}
}

```

Doing so depends on the following configuration file:

```

stubrunner:
  repositoryRoot: classpath:m2repo/repository/
  ids:
    - org.springframework.cloud.contract.verifier.stubs:loanIssuance
    - org.springframework.cloud.contract.verifier.stubs:fraudDetectionServer
    - org.springframework.cloud.contract.verifier.stubs:bootService
  stubs-mode: remote

```

Instead of using the properties, you can also use the properties inside the `@AutoConfigureStubRunner`. The following example achieves the same result by setting values on the annotation:

```

@AutoConfigureStubRunner(ids =
["org.springframework.cloud.contract.verifier.stubs:loanIssuance",
"org.springframework.cloud.contract.verifier.stubs:fraudDetectionServer",
"org.springframework.cloud.contract.verifier.stubs:bootService"] ,
stubsMode = StubRunnerProperties.StubsMode.REMOTE ,
repositoryRoot = "classpath:m2repo/repository/" )

```

Stub Runner Spring registers environment variables in the following manner for every registered WireMock server. The following example shows Stub Runner IDs for `com.example:thing1` and `com.example:thing2`:

- `stubrunner.runningstubs.thing1.port`
- `stubrunner.runningstubs.com.example.thing1.port`
- `stubrunner.runningstubs.thing2.port`
- `stubrunner.runningstubs.com.example.thing2.port`

You can reference these values in your code.

You can also use the `@StubRunnerPort` annotation to inject the port of a running stub. The value of the annotation can be the `groupid:artifactid` or only the `artifactid`. The following example works shows Stub Runner IDs for `com.example:thing1` and `com.example:thing2`.

```
@StubRunnerPort("thing1")
int thing1Port;
@StubRunnerPort("com.example:thing2")
int thing2Port;
```

3.5.5. Stub Runner Spring Cloud

Stub Runner can integrate with Spring Cloud.

For real life examples, see:

- [The producer application sample](#)
- [The consumer application sample](#)

Stubbing Service Discovery

The most important feature of `Stub Runner Spring Cloud` is the fact that it stubs:

- `DiscoveryClient`
- `ReactorServiceInstanceLoadBalancer`

That means that, regardless of whether you use Zookeeper, Consul, Eureka, or anything else, you do not need that in your tests. We are starting WireMock instances of your dependencies and we are telling your application, whenever you use `Feign`, to load a balanced `RestTemplate` or `DiscoveryClient` directly, to call those stubbed servers instead of calling the real Service Discovery tool.

Test Profiles and Service Discovery

In your integration tests, you typically do not want to call either a discovery service (such as Eureka) or Config Server. That is why you create an additional test configuration in which you want

to disable these features.

Due to certain limitations of `spring-cloud-commons`, to achieve this, you have to disable these properties in a static block such as the following example (for Eureka):

```
//Hack to work around https://github.com/spring-cloud/spring-cloud-  
commons/issues/156  
static {  
    System.setProperty("eureka.client.enabled", "false");  
    System.setProperty("spring.cloud.config.failFast", "false");  
}
```

Additional Configuration

You can match the `artifactId` of the stub with the name of your application by using the `stubrunner.idsToServiceIds`: map.



By default, all service discovery is stubbed. This means that, regardless of whether you have an existing `DiscoveryClient`, its results are ignored. However, if you want to reuse it, you can set `stubrunner.cloud.delegate.enabled` to `true`, and then your existing `DiscoveryClient` results are merged with the stubbed ones.

The default Maven configuration used by Stub Runner can be tweaked either by setting the following system properties or by setting the corresponding environment variables:

- `maven.repo.local`: Path to the custom maven local repository location
- `org.apache.maven.user-settings`: Path to custom maven user settings location
- `org.apache.maven.global-settings`: Path to maven global settings location

3.5.6. Using the Stub Runner Boot Application

Spring Cloud Contract Stub Runner Boot is a Spring Boot application that exposes REST endpoints to trigger the messaging labels and to access WireMock servers.

Stub Runner Boot Security

The Stub Runner Boot application is not secured by design - securing it would require to add security to all stubs even if they don't actually require it. Since this is a testing utility - the server is **not intended** to be used in production environments.



It is expected that **only a trusted client** has access to the Stub Runner Boot server. You should not run this application as a Fat Jar or a [Docker Image](#) in untrusted locations.

Stub Runner Server

To use the Stub Runner Server, add the following dependency:

```
compile "org.springframework.cloud:spring-cloud-starter-stub-runner"
```

Then annotate a class with `@EnableStubRunnerServer`, build a fat jar, and it is ready to work.

For the properties, see the [Stub Runner Spring](#) section.

Stub Runner Server Fat Jar

You can download a standalone JAR from Maven (for example, for version 2.0.1.RELEASE) by running the following commands:

```
$ wget -O stub-runner.jar  
'https://search.maven.org/remotecontent?filepath=org/springframework/cloud/spring-  
cloud-contract-stub-runner-boot/2.0.1.RELEASE/spring-cloud-contract-stub-runner-  
boot-2.0.1.RELEASE.jar'  
$ java -jar stub-runner.jar --stubrunner.ids=... --stubrunner.repositoryRoot=...
```

Spring Cloud CLI

Starting from the 1.4.0.RELEASE version of the [Spring Cloud CLI](#) project, you can start Stub Runner Boot by running `spring cloud stubrunner`.

To pass the configuration, you can create a `stubrunner.yml` file in the current working directory, in a subdirectory called `config`, or in `~/.spring-cloud`. The file could resemble the following example for running stubs installed locally:

Example 5. stubrunner.yml

```
stubrunner:  
  stubsMode: LOCAL  
  ids:  
    - com.example:beer-api-producer:+:9876
```

Then you can call `spring cloud stubrunner` from your terminal window to start the Stub Runner server. It is available at port 8750.

Endpoints

Stub Runner Boot offers two endpoints:

- [HTTP](#)
- [Messaging](#)

HTTP

For HTTP, Stub Runner Boot makes the following endpoints available:

- GET `/stubs`: Returns a list of all running stubs in `ivy:integer` notation
- GET `/stubs/{ivy}`: Returns a port for the given `ivy` notation (when calling the endpoint `ivy` can also be `artifactId` only)

Messaging

For Messaging, Stub Runner Boot makes the following endpoints available:

- GET `/triggers`: Returns a list of all running labels in `ivy : [label1, label2 ...]` notation
- POST `/triggers/{label}`: Runs a trigger with `label`
- POST `/triggers/{ivy}/{label}`: Runs a trigger with a `label` for the given `ivy` notation (when calling the endpoint, `ivy` can also be `artifactId` only)

Example

The following example shows typical usage of Stub Runner Boot:

```
@SpringBootTest(classes = StubRunnerBoot, properties =
"spring.cloud.zookeeper.enabled=false")
@ActiveProfiles("test")
class StubRunnerBootSpec {

    @Autowired
    StubRunning stubRunning

    @BeforeEach
    void setup() {
        RestAssuredMockMvc.standaloneSetup(new HttpStubsController(stubRunning),
            new TriggerController(stubRunning))
    }

    @Test
    void 'should return a list of running stub servers in "full ivy port" notation'()
    {
        when:
            String response = RestAssuredMockMvc.get('/stubs').body.asString()
        then:
            def root = new JsonSlurper().parseText(response)
            assert
            root.'org.springframework.cloud.contract.verifier.stubs:bootService:0.0.1-
            SNAPSHOT:stubs' instanceof Integer
    }
}
```

```

@Test
void 'should return a port on which a #stubId stub is running'() {
    given:
        def stubIds =
['org.springframework.cloud.contract.verifier.stubs:bootService:++stubs',
'org.springframework.cloud.contract.verifier.stubs:bootService:0.0.1-SNAPSHOT:stubs',
        'org.springframework.cloud.contract.verifier.stubs:bootService:+',
        'org.springframework.cloud.contract.verifier.stubs:bootService',
        'bootService']
    stubIds.each {
        when:
            def response = RestAssuredMockMvc.get("/stubs/${it}")
        then:
            assert response.statusCode == 200
            assert Integer.valueOf(response.body.asString()) > 0
        }
    }
}

@Test
void 'should return 404 when missing stub was called'() {
    when:
        def response = RestAssuredMockMvc.get("/stubs/a:b:c:d")
    then:
        assert response.statusCode == 404
    }
}

@Test
void 'should return a list of messaging labels that can be triggered when version
and classifier are passed'() {
    when:
        String response = RestAssuredMockMvc.get('/triggers').body.asString()
    then:
        def root = new JsonSlurper().parseText(response)
        assert
root.'org.springframework.cloud.contract.verifier.stubs:bootService:0.0.1-
SNAPSHOT:stubs'?.containsAll(["return_book_1"])
    }
}

@Test
void 'should trigger a messaging label'() {
    given:
        StubRunning stubRunning = Mockito.mock(StubRunning)
        RestAssuredMockMvc.standaloneSetup(new HttpStubsController(stubRunning),
new TriggerController(stubRunning))
    when:
        def response = RestAssuredMockMvc.post("/triggers/delete_book")
    then:
        response.statusCode == 200
    and:

```



```

        Mockito.verify(stubRunning).trigger('delete_book')
    }

    @Test
    void 'should trigger a messaging label for a stub with #stubId ivy notation'() {
        given:
            StubRunning stubRunning = Mockito.mock(StubRunning)
            RestAssuredMockMvc.standaloneSetup(new HttpStubsController(stubRunning),
new TriggerController(stubRunning))
            and:
                def stubIds =
['org.springframework.cloud.contract.verifier.stubs:bootService:stubs',
'org.springframework.cloud.contract.verifier.stubs:bootService', 'bootService']
                stubIds.each {
                    when:
                        def response = RestAssuredMockMvc.post("/triggers/$it/delete_book")
                    then:
                        assert response.statusCode == 200
                    and:
                        Mockito.verify(stubRunning).trigger(it, 'delete_book')
                }
            }
    }

    @Test
    void 'should throw exception when trigger is missing'() {
        when:
            BDDAssertions.thenThrownBy(() ->
RestAssuredMockMvc.post("/triggers/missing_label"))
                .hasMessageContaining("Exception occurred while trying to return
[missing_label] label.")
                .hasMessageContaining("Available labels are")

                .hasMessageContaining("org.springframework.cloud.contract.verifier.stubs:loanIssuance:
0.0.1-SNAPSHOT:stubs=[]")

                .hasMessageContaining("org.springframework.cloud.contract.verifier.stubs:bootService:0
.0.1-SNAPSHOT:stubs=")
            }
    }
}

```

Stub Runner Boot with Service Discovery

One way to use Stub Runner Boot is to use it as a feed of stubs for “smoke tests”. What does that mean? Assume that you do not want to deploy 50 microservices to a test environment in order to see whether your application works. You have already run a suite of tests during the build process, but you would also like to ensure that the packaging of your application works. You can deploy your application to an environment, start it, and run a couple of tests on it to see whether it works. We can call those tests “smoke tests”, because their purpose is to check only a handful of testing

scenarios.

The problem with this approach is that, if you use microservices, you most likely also use a service discovery tool. Stub Runner Boot lets you solve this issue by starting the required stubs and registering them in a service discovery tool.

Now assume that we want to start this application so that the stubs get automatically registered. We can do so by running the application with `java -jar ${SYSTEM_PROPS} stub-runner-boot-eureka-example.jar`, where `${SYSTEM_PROPS}`.

That way, your deployed application can send requests to started WireMock servers through service discovery. Most likely, points 1 through 3 could be set by default in `application.yml`, because they are not likely to change. That way, you can provide only the list of stubs to download whenever you start the Stub Runner Boot.

3.5.7. Consumer-Driven Contracts: Stubs Per Consumer

There are cases in which two consumers of the same endpoint want to have two different responses.



This approach also lets you immediately know which consumer uses which part of your API. You can remove part of a response that your API produces and see which of your autogenerated tests fails. If none fails, you can safely delete that part of the response, because nobody uses it.

Consider the following example of a contract defined for the producer called `producer`, which has two consumers (`foo-consumer` and `bar-consumer`):

Consumer `foo-service`

```
request {
  url '/foo'
  method GET()
}
response {
  status OK()
  body(
    foo: "foo"
  )
}
```

Consumer `bar-service`

```
request {
  url '/bar'
  method GET()
}
response {
  status OK()
  body(
    bar: "bar"
  )
}
```

You cannot produce two different responses for the same request. That is why you can properly package the contracts and then profit from the `stubsPerConsumer` feature.

On the producer side, the consumers can have a folder that contains contracts related only to them. By setting the `stubrunner.stubs-per-consumer` flag to `true`, we no longer register all stubs but only those that correspond to the consumer application's name. In other words, we scan the path of every stub and, if it contains a subfolder with name of the consumer in the path, only then is it registered.

On the `foo` producer side the contracts would look like this

```
.
├── contracts
│   ├── bar-consumer
│   │   ├── bookReturnedForBar.groovy
│   │   └── shouldCallBar.groovy
│   └── foo-consumer
│       ├── bookReturnedForFoo.groovy
│       └── shouldCallFoo.groovy
```

The `bar-consumer` consumer can either set the `spring.application.name` or the `stubrunner.consumer-`

name to `bar-consumer` Alternatively, you can set the test as follows:

```
@SpringBootTest(classes = Config, properties = ["spring.application.name=bar-
consumer"])
@AutoConfigureStubRunner(ids =
"org.springframework.cloud.contract.verifier.stubs:producerWithMultipleConsumers",
    repositoryRoot = "classpath:m2repo/repository/",
    stubsMode = StubRunnerProperties.StubsMode.REMOTE,
    stubsPerConsumer = true)
@ActiveProfiles("streamconsumer")
class StubRunnerStubsPerConsumerSpec {
    ...
}
```

Then only the stubs registered under a path that contains `bar-consumer` in its name (that is, those from the `src/test/resources/contracts/bar-consumer/some/contracts/...` folder) are allowed to be referenced.

You can also set the consumer name explicitly, as follows:

```
@SpringBootTest(classes = Config)
@AutoConfigureStubRunner(ids =
"org.springframework.cloud.contract.verifier.stubs:producerWithMultipleConsumers",
    repositoryRoot = "classpath:m2repo/repository/",
    consumerName = "foo-consumer",
    stubsMode = StubRunnerProperties.StubsMode.REMOTE,
    stubsPerConsumer = true)
@ActiveProfiles("streamconsumer")
class StubRunnerStubsPerConsumerWithConsumerNameSpec {
    ...
}
```

Then only the stubs registered under a path that contains the `foo-consumer` in its name (that is, those from the `src/test/resources/contracts/foo-consumer/some/contracts/...` folder) are allowed to be referenced.

For more information about the reasons behind this change, see [issue 224](#).

3.5.8. Fetching Stubs or Contract Definitions From A Location

Instead of picking the stubs or contract definitions from Artifactory, Nexus, or Git, you can point to a location on a drive or the classpath. Doing so can be especially useful in a multi-module project, where one module wants to reuse stubs or contracts from another module without the need to actually install those in a local maven repository to commit those changes to Git.

In order to achieve this, you can use the `stubs://` protocol when the repository root parameter is set either in Stub Runner or in a Spring Cloud Contract plugin.

In this example, the `producer` project has been successfully built and stubs were generated under the `target/stubs` folder. As a consumer, one can set up the Stub Runner to pick the stubs from that location by using the `stubs://` protocol.

Annotation

```
@AutoConfigureStubRunner(  
    stubsMode = StubRunnerProperties.StubsMode.REMOTE,  
    repositoryRoot = "stubs://file://location/to/the/producer/target/stubs/",  
    ids = "com.example:some-producer")
```

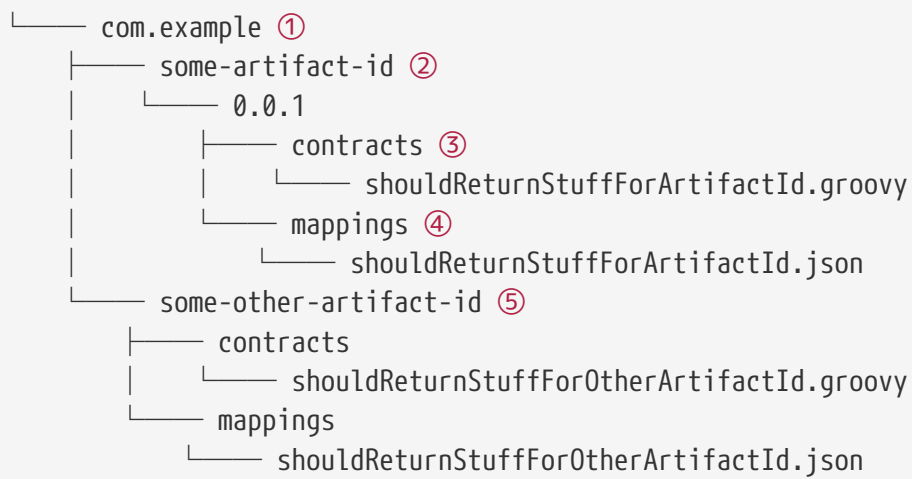
JUnit 4 Rule

```
@Rule  
public StubRunnerRule rule = new StubRunnerRule()  
    .downloadStub("com.example:some-producer")  
    .repoRoot("stubs://file://location/to/the/producer/target/stubs/")  
    .stubsMode(StubRunnerProperties.StubsMode.REMOTE);
```

JUnit 5 Extension

```
@RegisterExtension  
public StubRunnerExtension stubRunnerExtension = new StubRunnerExtension()  
    .downloadStub("com.example:some-producer")  
    .repoRoot("stubs://file://location/to/the/producer/target/stubs/")  
    .stubsMode(StubRunnerProperties.StubsMode.REMOTE);
```

Contracts and stubs may be stored in a location, where each producer has its own, dedicated folder for contracts and stub mappings. Under that folder, each consumer can have its own setup. To make Stub Runner find the dedicated folder from the provided IDs, you can pass the `stubs.find-producer=true` property or the `stubrunner.stubs.find-producer=true` system property. The following listing shows an arrangement of contracts and stubs:



① group ID of the consumers

② consumer with artifact id [some-artifact-id]

③ contracts for the consumer with artifact id [some-artifact-id]

④ mappings for the consumer with artifact id [some-artifact-id]

⑤ consumer with artifact id [some-other-artifact-id]

Annotation

```
@AutoConfigureStubRunner(  
    stubsMode = StubRunnerProperties.StubsMode.REMOTE,  
    repositoryRoot = "stubs://file://location/to/the/contracts/directory",  
    ids = "com.example:some-producer",  
    properties="stubs.find-producer=true")
```

JUnit 4 Rule

```
static Map<String, String> contractProperties() {  
    Map<String, String> map = new HashMap<>();  
    map.put("stubs.find-producer", "true");  
    return map;  
}  
  
@Rule  
public StubRunnerRule rule = new StubRunnerRule()  
    .downloadStub("com.example:some-producer")  
    .repoRoot("stubs://file://location/to/the/contracts/directory")  
    .stubsMode(StubRunnerProperties.StubsMode.REMOTE)  
    .properties(contractProperties());
```

JUnit 5 Extension

```
static Map<String, String> contractProperties() {  
    Map<String, String> map = new HashMap<>();  
    map.put("stubs.find-producer", "true");  
    return map;  
}  
  
@RegisterExtension  
public StubRunnerExtension stubRunnerExtension = new StubRunnerExtension()  
    .downloadStub("com.example:some-producer")  
    .repoRoot("stubs://file://location/to/the/contracts/directory")  
    .stubsMode(StubRunnerProperties.StubsMode.REMOTE)  
    .properties(contractProperties());
```

3.5.9. Generating Stubs at Runtime

As a consumer, you might not want to wait for the producer to finish its implementation and then publish their stubs. A solution to this problem can be generation of stubs at runtime.

As a producer, when a contract is defined, you are required to make the generated tests pass in order for the stubs to be published. There are cases where you would like to unblock the consumers so that they can fetch the stubs before your tests actually pass. In this case, you should set such contracts as in-progress. You can read more about this under the [Contracts in Progress](#) section. That

way, your tests are not generated, but the stubs are generated.

As a consumer, you can toggle a switch to generate stubs at runtime. Stub Runner ignores all the existing stub mappings and generates new ones for all the contract definitions. Another option is to pass the `stubrunner.generate-stubs` system property. The following example shows such a setup:

Annotation

```
@AutoConfigureStubRunner(  
    stubsMode = StubRunnerProperties.StubsMode.REMOTE,  
    repositoryRoot = "stubs://file://location/to/the/contracts",  
    ids = "com.example:some-producer",  
    generateStubs = true)
```

JUnit 4 Rule

```
@Rule  
public StubRunnerRule rule = new StubRunnerRule()  
    .downloadStub("com.example:some-producer")  
    .repoRoot("stubs://file://location/to/the/contracts")  
    .stubsMode(StubRunnerProperties.StubsMode.REMOTE)  
    .withGenerateStubs(true);
```

JUnit 5 Extension

```
@RegisterExtension  
public StubRunnerExtension stubRunnerExtension = new StubRunnerExtension()  
    .downloadStub("com.example:some-producer")  
    .repoRoot("stubs://file://location/to/the/contracts")  
    .stubsMode(StubRunnerProperties.StubsMode.REMOTE)  
    .withGenerateStubs(true);
```

3.5.10. Fail On No Stubs

By default, Stub Runner will fail if no stubs are found. In order to change that behavior, set the `failOnNoStubs` property to `false` in the annotation or call the `withFailOnNoStubs(false)` method on a JUnit Rule or Extension. The following example shows how to do so:

Annotation

```
@AutoConfigureStubRunner(  
    stubsMode = StubRunnerProperties.StubsMode.REMOTE,  
    repositoryRoot = "stubs://file://location/to/the/contracts",  
    ids = "com.example:some-producer",  
    failOnNoStubs = false)
```

JUnit 4 Rule

```
@Rule  
public StubRunnerRule rule = new StubRunnerRule()  
    .downloadStub("com.example:some-producer")  
    .repoRoot("stubs://file://location/to/the/contracts")  
    .stubsMode(StubRunnerProperties.StubsMode.REMOTE)  
    .withFailOnNoStubs(false);
```

JUnit 5 Extension

```
@RegisterExtension  
public StubRunnerExtension stubRunnerExtension = new StubRunnerExtension()  
    .downloadStub("com.example:some-producer")  
    .repoRoot("stubs://file://location/to/the/contracts")  
    .stubsMode(StubRunnerProperties.StubsMode.REMOTE)  
    .withFailOnNoStubs(false);
```

3.5.11. Common Properties

This section briefly describes common properties, including:

- [Common Properties for JUnit and Spring](#)
- [Stub Runner Stubs IDs](#)

Common Properties for JUnit and Spring

You can set repetitive properties by using system properties or Spring configuration properties. The following table shows their names with their default values:

Property name	Default value	Description
<code>stubrunner.minPort</code>	10000	Minimum value of a port for a started WireMock with stubs.
<code>stubrunner.maxPort</code>	15000	Maximum value of a port for a started WireMock with stubs.
<code>stubrunner.repositoryRoot</code>		Maven repository URL. If blank, then call the local Maven repo.

Property name	Default value	Description
<code>stubrunner.classifier</code>	<code>stubs</code>	Default classifier for the stub artifacts.
<code>stubrunner.stubsMode</code>	<code>CLASSPATH</code>	The way you want to fetch and register the stubs.
<code>stubrunner.ids</code>		Array of Ivy notation stubs to download.
<code>stubrunner.username</code>		Optional username to access the tool that stores the JARs with stubs.
<code>stubrunner.password</code>		Optional password to access the tool that stores the JARs with stubs.
<code>stubrunner.stubsPerConsumer</code>	<code>false</code>	Set to <code>true</code> if you want to use different stubs for each consumer instead of registering all stubs for every consumer.
<code>stubrunner.consumerName</code>		If you want to use a stub for each consumer and want to override the consumer name, change this value.

Stub Runner Stubs IDs

You can set the stubs to download in the `stubrunner.ids` system property. They use the following pattern:

```
groupId:artifactId:version:classifier:port
```

Note that `version`, `classifier`, and `port` are optional.

- If you do not provide the `port`, a random one is picked.
- If you do not provide the `classifier`, the default is used. (Note that you can pass an empty classifier this way: `groupId:artifactId:version:`).
- If you do not provide the `version`, then `+` is passed, and the latest one is downloaded.

`port` means the port of the WireMock server.



Starting with version 1.0.4, you can provide a range of versions that you would like the Stub Runner to take into consideration. You can read more about the Aether versioning ranges [here](#).

3.6. Spring Cloud Contract WireMock

The Spring Cloud Contract WireMock modules let you use [WireMock](#) in a Spring Boot application. For more detail, check out the [samples](#).

If you have a Spring Boot application that uses Tomcat as an embedded server (which is the default with `spring-boot-starter-web`), you can add `spring-cloud-starter-contract-stub-runner` to your classpath and add `@AutoConfigureWireMock` to use Wiremock in your tests. Wiremock runs as a stub server, and you can register stub behavior by using a Java API or by using static JSON declarations as part of your test. The following code shows an example:

```
@SpringBootTest(webEnvironment = WebEnvironment.RANDOM_PORT)
@AutoConfigureWireMock(port = 0)
public class WiremockForDocsTests {

    // A service that calls out over HTTP
    @Autowired
    private Service service;

    @BeforeEach
    public void setup() {
        this.service.setBase("http://localhost:"
            + this.environment.getProperty("wiremock.server.port"));
    }

    // Using the WireMock APIs in the normal way:
    @Test
    public void contextLoads() throws Exception {
        // Stubbing WireMock
        stubFor(get(urlEqualTo("/resource")).willReturn(aResponse()
            .withHeader("Content-Type", "text/plain").withBody("Hello
World!"))));
        // We're asserting if WireMock responded properly
        assertThat(this.service.go()).isEqualTo("Hello World!");
    }
}
```

To start the stub server on a different port, use (for example), `@AutoConfigureWireMock(port=9999)`. For a random port, use a value of `0`. The stub server port can be bound in the test application context with the `wiremock.server.port` property. Using `@AutoConfigureWireMock` adds a bean of type `WiremockConfiguration` to your test application context, where it is cached between methods and classes that have the same context. The same is true for Spring integration tests. Also, you can inject a bean of type `WireMockServer` into your test. The registered WireMock server is reset after each test class. However, if you need to reset it after each test method, set the `wiremock.reset-mappings-after-each-test` property to `true`.

3.6.1. Registering Stubs Automatically

If you use `@AutoConfigureWireMock`, it registers WireMock JSON stubs from the file system or classpath (by default, from `file:src/test/resources/mappings`). You can customize the locations by using the `stubs` attribute in the annotation, which can be an Ant-style resource pattern or a directory. In the case of a directory, `/*.json` is appended. The following code shows an example:

```
@RunWith(SpringRunner.class)
@SpringBootTest
@AutoConfigureWireMock(stubs="classpath:/stubs")
public class WiremockImportApplicationTests {

    @Autowired
    private Service service;

    @Test
    public void contextLoads() throws Exception {
        assertThat(this.service.go()).isEqualTo("Hello World!");
    }
}
```



Actually, WireMock always loads mappings from `src/test/resources/mappings` **as well as** the custom locations in the `stubs` attribute. To change this behavior, you can also specify a file root, as described in the [next section of this document](#).



Also, the mappings in the `stubs` location are not considered part of Wiremock's "default mappings" and calls to `com.github.tomakehurst.wiremock.client.WireMock.resetToDefaultMappings` during a test do not result in the mappings in the `stubs` location being included. However, the `org.springframework.cloud.contract.wiremock.WireMockTestExecutionListener` does reset the mappings (including adding the ones from the stubs location) after every test class and, optionally, after every test method (guarded by the `wiremock.reset-mappings-after-each-test` property).

If you use Spring Cloud Contract's default stub jars, your stubs are stored in the `/META-INF/group-id/artifact-id/versions/mappings/` folder. If you want to register all stubs from that location, from all embedded JARs, you can use the following syntax:

```
@AutoConfigureWireMock(port = 0, stubs = "classpath*/META-INF/**/mappings/**/*.json")
```

3.6.2. Using Files to Specify the Stub Bodies

WireMock can read response bodies from files on the classpath or the file system. In the case of the file system, you can see in the JSON DSL that the response has a `bodyFileName` instead of a (literal) `body`. The files are resolved relative to a root directory (by default, `src/test/resources/__files`). To customize this location, you can set the `files` attribute in the `@AutoConfigureWireMock` annotation to the location of the parent directory (in other words, `__files` is a subdirectory). You can use a Spring resource notation to refer to `file:...` or `classpath:...` locations. Generic URLs are not supported. A list of values can be given—in which case, WireMock resolves the first file that exists when it needs to find a response body.



When you configure the `files` root, it also affects the automatic loading of stubs, because they come from the root location in a subdirectory called `mappings`.



The value of `files` has no effect on the stubs loaded explicitly from the `stubs` attribute.

3.6.3. Alternative: Using JUnit Rules

For a more conventional WireMock experience, you can use JUnit `@Rules` to start and stop the server. To do so, use the `WireMockSpring` convenience class to obtain an `Options` instance, as the following example shows:

```

@SpringBootTest(webEnvironment = WebEnvironment.RANDOM_PORT)
public class WiremockForDocsClassRuleTests {

    // Start WireMock on some dynamic port
    // for some reason `dynamicPort()` is not working properly
    public static WireMockServer wiremock = new
WireMockServer(WireMockSpring.options().dynamicPort());

    @BeforeAll
    static void setupClass() {
        wiremock.start();
    }

    @AfterEach
    void after() {
        wiremock.resetAll();
    }

    @AfterAll
    static void clean() {
        wiremock.shutdown();
    }

    // A service that calls out over HTTP to wiremock's port
    @Autowired
    private Service service;

    @BeforeEach
    public void setup() {
        this.service.setBase("http://localhost:" + wiremock.port());
    }

    // Using the WireMock APIs in the normal way:
    @Test
    public void contextLoads() throws Exception {
        // Stubbing WireMock
        wiremock.stubFor(get(urlEqualTo("/resource")).willReturn(aResponse()
            .withHeader("Content-Type", "text/plain").withBody("Hello
World!"))));
        // We're asserting if WireMock responded properly
        assertThat(this.service.go()).isEqualTo("Hello World!");
    }
}

```

The `@ClassRule` means that the server shuts down after all the methods in this class have been run.

3.6.4. Relaxed SSL Validation for Rest Template

WireMock lets you stub a “secure” server with an `https` URL protocol. If your application wants to contact that stub server in an integration test, it finds that the SSL certificates are not valid (the usual problem with self-installed certificates). The best option is often to re-configure the client to use `http`. If that is not an option, you can ask Spring to configure an HTTP client that ignores SSL validation errors (do so only for tests, of course).

To make this work with minimum fuss, you need to use the Spring Boot `RestTemplateBuilder` in your application, as the following example shows:

```
@Bean
public RestTemplate restTemplate(RestTemplateBuilder builder) {
    return builder.build();
}
```

You need `RestTemplateBuilder` because the builder is passed through callbacks to initialize it, so the SSL validation can be set up in the client at that point. This happens automatically in your test if you use the `@AutoConfigureWireMock` annotation or the stub runner. If you use the JUnit `@Rule` approach, you need to add the `@AutoConfigureHttpClient` annotation as well, as the following example shows:

```
@RunWith(SpringRunner.class)
@SpringBootTest("app.baseUrl=https://localhost:6443")
@AutoConfigureHttpClient
public class WiremockHttpsServerApplicationTests {

    @ClassRule
    public static WireMockClassRule wiremock = new WireMockClassRule(
        WireMockSpring.options().httpsPort(6443));

    ...
}
```

If you use `spring-boot-starter-test`, you have the Apache HTTP client on the classpath, and it is selected by the `RestTemplateBuilder` and configured to ignore SSL errors. If you use the default `java.net` client, you do not need the annotation (but it does no harm). There is currently no support for other clients, but it may be added in future releases.

To disable the custom `RestTemplateBuilder`, set the `wiremock.rest-template-ssl-enabled` property to `false`.

3.6.5. WireMock and Spring MVC Mocks

Spring Cloud Contract provides a convenience class that can load JSON WireMock stubs into a

Spring `MockRestServiceServer`. The following code shows an example:

```
@SpringBootTest(webEnvironment = WebEnvironment.NONE)
public class WiremockForDocsMockServerApplicationTests {

    @Autowired
    private RestTemplate restTemplate;

    @Autowired
    private Service service;

    @Test
    public void contextLoads() throws Exception {
        // will read stubs classpath
        MockRestServiceServer server =
        WireMockRestServiceServer.with(this.restTemplate)

        .baseUrl("https://example.org").stubs("classpath:/stubs/resource.json")
            .build();
        // We're asserting if WireMock responded properly
        assertThat(this.service.go()).isEqualTo("Hello World");
        server.verify();
    }
}
```

The `baseUrl` value is prepended to all mock calls, and the `stubs()` method takes a stub path resource pattern as an argument. In the preceding example, the stub defined at `/stubs/resource.json` is loaded into the mock server. If the `RestTemplate` is asked to visit `example.org/`, it gets the responses as being declared at that URL. More than one stub pattern can be specified, and each one can be a directory (for a recursive list of all `.json`), a fixed filename (as in the preceding example), or an Ant-style pattern. The JSON format is the normal WireMock format, which you can read about at the [WireMock website](#).

Currently, the Spring Cloud Contract Verifier supports Tomcat, Jetty, and Undertow as Spring Boot embedded servers, and Wiremock itself has “native” support for a particular version of Jetty (currently 9.2). To use the native Jetty, you need to add the native Wiremock dependencies and exclude the Spring Boot container (if there is one).

3.7. Build Tools Integration

You can run test generation and stub invocation in various ways. The most common ones are as follows:

- [Maven](#)
- [Gradle](#)

- [Docker](#)

3.8. What to Read Next

If you want to learn more about any of the classes discussed in this section, you can browse the [source code directly](#). If you have specific questions, see the [how-to](#) section.

If you are comfortable with Spring Cloud Contract's core features, you can continue on and read about [Spring Cloud Contract's advanced features](#).

4. Maven Project

To learn how to set up the Maven project for Spring Cloud Contract Verifier, read the following sections:

- [Adding the Maven Plugin](#)
- [Maven and Rest Assured 2.0](#)
- [Using Snapshot and Milestone Versions for Maven](#)
- [Adding stubs](#)
- [Run Plugin](#)
- [Configure plugin](#)
- [Configuration Options](#)
- [Single Base Class for All Tests](#)
- [Using Different Base Classes for Contracts](#)
- [Invoking Generated Tests](#)
- [Pushing Stubs to SCM](#)
- [Maven Plugin and STS](#)

You can also check the plugin's documentation [here](#).

4.1. Adding the Maven Plugin

To add the Spring Cloud Contract BOM, include the following section in your `pom.xml` file:

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

Next, add the **Spring Cloud Contract Verifier** Maven plugin, as follows:

```
<plugin>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-contract-maven-plugin</artifactId>
  <version>${spring-cloud-contract.version}</version>
  <extensions>true</extensions>
  <configuration>

<packageWithBaseClasses>com.example.fraud</packageWithBaseClasses>
<!--           <convertToYaml>true</convertToYaml>-->
    </configuration>
</plugin>
```

You can read more in the [spring-cloud-contract-maven-plugin/index.html](https://docs.spring.io/spring-cloud-contract/maven-plugin/index.html) [Spring Cloud Contract Maven Plugin Documentation].

Sometimes, regardless of the picked IDE, you can see that the **target/generated-test-source** folder is not visible on the IDE's classpath. To ensure that it is always there, you can add the following entry to your **pom.xml**

```
<plugin>
  <groupId>org.codehaus.mojo</groupId>
  <artifactId>build-helper-maven-plugin</artifactId>
  <executions>
    <execution>
      <id>add-source</id>
      <phase>generate-test-sources</phase>
      <goals>
        <goal>add-test-source</goal>
      </goals>
      <configuration>
        <sources>
          <source>${project.build.directory}/generated-test-
sources/contracts/</source>
        </sources>
      </configuration>
    </execution>
  </executions>
</plugin>
```

4.2. Maven and Rest Assured 2.0

By default, Rest Assured 3.x is added to the classpath. However, you can use Rest Assured 2.x by

adding it to the plugins classpath, as follows:

```
<plugin>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-contract-maven-plugin</artifactId>
  <version>${spring-cloud-contract.version}</version>
  <extensions>>true</extensions>
  <configuration>
    <packageWithBaseClasses>com.example</packageWithBaseClasses>
  </configuration>
  <dependencies>
    <dependency>
      <groupId>org.springframework.cloud</groupId>
      <artifactId>spring-cloud-contract-verifier</artifactId>
      <version>${spring-cloud-contract.version}</version>
    </dependency>
    <dependency>
      <groupId>com.jayway.restassured</groupId>
      <artifactId>rest-assured</artifactId>
      <version>2.5.0</version>
      <scope>compile</scope>
    </dependency>
    <dependency>
      <groupId>com.jayway.restassured</groupId>
      <artifactId>spring-mock-mvc</artifactId>
      <version>2.5.0</version>
      <scope>compile</scope>
    </dependency>
  </dependencies>
</plugin>

<dependencies>
  <!-- all dependencies -->
  <!-- you can exclude rest-assured from spring-cloud-contract-verifier -->
  <dependency>
    <groupId>com.jayway.restassured</groupId>
    <artifactId>rest-assured</artifactId>
    <version>2.5.0</version>
    <scope>test</scope>
  </dependency>
  <dependency>
    <groupId>com.jayway.restassured</groupId>
    <artifactId>spring-mock-mvc</artifactId>
    <version>2.5.0</version>
    <scope>test</scope>
  </dependency>
</dependencies>
```

That way, the plugin automatically sees that Rest Assured 2.x is present on the classpath and modifies the imports accordingly.

4.3. Using Snapshot and Milestone Versions for Maven

To use Snapshot and Milestone versions, you have to add the following section to your `pom.xml`:

```
<repositories>
  <repository>
    <id>spring-snapshots</id>
    <name>Spring Snapshots</name>
    <url>https://repo.spring.io/snapshot</url>
    <snapshots>
      <enabled>>true</enabled>
    </snapshots>
  </repository>
  <repository>
    <id>spring-milestones</id>
    <name>Spring Milestones</name>
    <url>https://repo.spring.io/milestone</url>
    <snapshots>
      <enabled>>false</enabled>
    </snapshots>
  </repository>
  <repository>
    <id>spring-releases</id>
    <name>Spring Releases</name>
    <url>https://repo.spring.io/release</url>
    <snapshots>
      <enabled>>false</enabled>
    </snapshots>
  </repository>
</repositories>
<pluginRepositories>
  <pluginRepository>
    <id>spring-snapshots</id>
    <name>Spring Snapshots</name>
    <url>https://repo.spring.io/snapshot</url>
    <snapshots>
      <enabled>>true</enabled>
    </snapshots>
  </pluginRepository>
  <pluginRepository>
    <id>spring-milestones</id>
    <name>Spring Milestones</name>
    <url>https://repo.spring.io/milestone</url>
    <snapshots>
      <enabled>>false</enabled>
    </snapshots>
  </pluginRepository>
  <pluginRepository>
    <id>spring-releases</id>
    <name>Spring Releases</name>
    <url>https://repo.spring.io/release</url>
    <snapshots>
      <enabled>>false</enabled>
    </snapshots>
  </pluginRepository>
</pluginRepositories>
```

```
</snapshots>
</pluginRepository>
</pluginRepositories>
```

4.4. Adding stubs

By default, Spring Cloud Contract Verifier looks for stubs in the `src/test/resources/contracts` directory. The directory containing stub definitions is treated as a class name, and each stub definition is treated as a single test. We assume that it contains at least one directory to be used as the test class name. If there is more than one level of nested directories, all except the last one is used as the package name. Consider the following structure:

```
src/test/resources/contracts/myservice/shouldCreateUser.groovy
src/test/resources/contracts/myservice/shouldReturnUser.groovy
```

Given that structure, Spring Cloud Contract Verifier creates a test class named `defaultBasePackage.MyService` with two methods:

- `shouldCreateUser()`
- `shouldReturnUser()`

4.5. Run Plugin

The `generateTests` plugin goal is assigned to be invoked in the phase called `generate-test-sources`. If you want it to be part of your build process, you need not do anything. If you want only to generate tests, invoke the `generateTests` goal.

If you want to run stubs from Maven, call the `run` goal with the stubs to run as the `spring.cloud.contract.verifier.stubs` system property as follows:

```
mvn          org.springframework.cloud:spring-cloud-contract-maven-plugin:run \
-Dspring.cloud.contract.verifier.stubs="com.acme:service-name"
```

4.6. Configure plugin

To change the default configuration, you can add a `configuration` section to the plugin definition or the `execution` definition, as follows:

```

<plugin>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-contract-maven-plugin</artifactId>
  <executions>
    <execution>
      <goals>
        <goal>convert</goal>
        <goal>generateStubs</goal>
        <goal>generateTests</goal>
      </goals>
    </execution>
  </executions>
  <configuration>

  <basePackageForTests>org.springframework.cloud.verifier.twitter.place</basePackage
  ForTests>

  <baseClassForTests>org.springframework.cloud.verifier.twitter.place.BaseMockMvcSpe
  c</baseClassForTests>
    </configuration>
</plugin>

```

4.7. Configuration Options

- **testMode**: Defines the mode for acceptance tests. By default, the mode is **MockMvc**, which is based on Spring's MockMvc. You can also change it to **WebTestClient**, **JaxRsClient**, or **Explicit** (for real HTTP calls).
- **basePackageForTests**: Specifies the base package for all generated tests. If not set, the value is picked from the package of **baseClassForTests** and from **packageWithBaseClasses**. If neither of these values are set, the value is set to **org.springframework.cloud.contract.verifier.tests**.
- **ruleClassForTests**: Specifies a rule that should be added to the generated test classes.
- **baseClassForTests**: Creates a base class for all generated tests. By default, if you use Spock classes, the class is **spock.lang.Specification**.
- **contractsDirectory**: Specifies a directory that contains contracts written with the Groovyn DSL. The default directory is **/src/test/resources/contracts**.
- **generatedTestSourcesDir**: Specifies the test source directory where tests generated from the Groovy DSL should be placed. By default, its value is **\$buildDir/generated-test-sources/contracts**.
- **generatedTestResourcesDir**: Specifies the test resource directory for resources used by the generated tests.
- **testFramework**: Specifies the target test framework to be used. Currently, Spock, JUnit 4 (**TestFramework.JUNIT**), and JUnit 5 are supported, with JUnit 4 being the default framework.
- **packageWithBaseClasses**: Defines a package where all the base classes reside. This setting takes

precedence over `baseClassForTests`. The convention is such that, if you have a contract under (for example) `src/test/resources/contract/foo/bar/baz/` and set the value of the `packageWithBaseClasses` property to `com.example.base`, Spring Cloud Contract Verifier assumes that there is a `BarBazBase` class under the `com.example.base` package. In other words, the system takes the last two parts of the package, if they exist, and forms a class with `Base` as a suffix.

- `baseClassMappings`: Specifies a list of base class mappings that provide `contractPackageRegex` (which is checked against the package where the contract is located) and `baseClassFQN` (which maps to the fully qualified name of the base class for the matched contract). For example, if you have a contract under `src/test/resources/contract/foo/bar/baz/` and map the `.* → com.example.base.BaseClass` property, the test class generated from these contracts extends `com.example.base.BaseClass`. This setting takes precedence over `packageWithBaseClasses` and `baseClassForTests`.
- `contractsProperties`: A map that contains properties to be passed to Spring Cloud Contract components. Those properties might be used by (for example) built-in or custom Stub Downloaders.
- `failOnNoContracts`: When enabled, will throw an exception when no contracts were found. Defaults to `true`.
- `failOnInProgress`: If set to `true`, then, if any contracts that are in progress are found, they break the build. On the producer side, you need to be explicit about the fact that you have contracts in progress and take into consideration that you might be causing false positive test results on the consumer side. Defaults to `true`.
- `incrementalContractTests`: When enabled, tests are created only when contracts have changed since last build. Defaults to `true`.
- `incrementalContractStubs`: When enabled, stubs are created only when contracts have changed since last build. Defaults to `true`.
- `incrementalContractStubsJar`: When enabled, stubs jar is created only when stubs have changed since last build. Defaults to `true`. `*httpPort`: HTTP port for the WireMock server that serves stubs. Currently `spring.cloud.contract.verifier.http.port` property works only when serving stubs from the directory. Otherwise, when providing stubs id, port have to be included in the id string. `*skip`: Set this to `true` to bypass the verifier execution. `*skipTestOnly`: Set this to `true` to bypass verifier test generation. `*stubs`: List of stubs to be downloaded and ran in a colon separated Ivy notation. `*minPort`: Specifies the minimal port at which the stub should start. `*maxPort`: Specifies the maximal port at which the stub should start. `*waitForKeyPressed`: Specifies if the plugin should wait for the user to press the key after starting the stubs. `*stubsClassifier`: Specifies the classifier used by stubs artifacts.

If you want to download your contract definitions from a Maven repository, you can use the following options:

- `contractDependency`: The contract dependency that contains all the packaged contracts.
- `contractsPath`: The path to the concrete contracts in the JAR with packaged contracts. Defaults to `groupid/artifactid` where `groupid` is slash separated.
- `contractsMode`: Picks the mode in which stubs are found and registered.
- `deleteStubsAfterTest`: If set to `false`, do not remove any downloaded contracts from temporary

directories.

- `contractsRepositoryUrl`: URL to a repository with the artifacts that have contracts. If it is not provided, use the current Maven ones.
- `contractsRepositoryUsername`: The user name to be used to connect to the repo with contracts.
- `contractsRepositoryPassword`: The password to be used to connect to the repo with contracts.
- `contractsRepositoryProxyHost`: The proxy host to be used to connect to the repo with contracts.
- `contractsRepositoryProxyPort`: The proxy port to be used to connect to the repo with contracts.

We cache only non-snapshot, explicitly provided versions (for example `+` or `1.0.0.BUILD-SNAPSHOT` do not get cached). By default, this feature is turned on.

The following list describes experimental features that you can turn on in the plugin:

- `convertToYaml`: Converts all DSLs to the declarative YAML format. This can be extremely useful when you use external libraries in your Groovy DSLs. By turning this feature on (by setting it to `true`), you need not add the library dependency on the consumer side.
- `assertJsonSize`: You can check the size of JSON arrays in the generated tests. This feature is disabled by default.

4.8. Single Base Class for All Tests

When using Spring Cloud Contract Verifier in the default (`MockMvc`), you need to create a base specification for all generated acceptance tests. In this class, you need to point to an endpoint, which should be verified. The following example shows how to do so:

```
package org.mycompany.tests

import org.mycompany.ExampleSpringController
import com.jayway.restassured.module.mockmvc.RestAssuredMockMvc
import spock.lang.Specification

class MvcSpec extends Specification {
    def setup() {
        RestAssuredMockMvc.standaloneSetup(new ExampleSpringController())
    }
}
```

If necessary, you can also setup the whole context, as the following example shows:

```

import io.restassured.module.mockmvc.RestAssuredMockMvc;
import org.junit.Before;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.test.context.junit4.SpringRunner;
import org.springframework.web.context.WebApplicationContext;

@RunWith(SpringRunner.class)
@SpringBootTest(webEnvironment = WebEnvironment.RANDOM_PORT, classes =
SomeConfig.class, properties="some=property")
public abstract class BaseTestClass {

    @Autowired
    WebApplicationContext context;

    @Before
    public void setup() {
        RestAssuredMockMvc.webAppContextSetup(this.context);
    }
}

```

If you use **EXPLICIT** mode, you can use a base class to initialize the whole tested app, similar to what you might do in regular integration tests. The following example shows how to do so:

```

import io.restassured.RestAssured;
import org.junit.Before;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.boot.web.server.LocalServerPort;
import org.springframework.test.context.junit4.SpringRunner;
import org.springframework.web.context.WebApplicationContext;

@RunWith(SpringRunner.class)
@SpringBootTest(webEnvironment = WebEnvironment.RANDOM_PORT, classes =
SomeConfig.class, properties="some=property")
public abstract class BaseTestClass {

    @LocalServerPort
    int port;

    @Before
    public void setup() {
        RestAssured.baseURI = "http://localhost:" + this.port;
    }
}

```

If you use the **JAXRSCLIENT** mode, this base class should also contain a **protected WebTarget webTarget** field. Right now, the only way to test the JAX-RS API is to start a web server.

4.9. Using Different Base Classes for Contracts

If your base classes differ between contracts, you can tell the Spring Cloud Contract plugin which class should get extended by the autogenerated tests. You have two options:

- Follow a convention by providing a value for **packageWithBaseClasses**
- Provide explicit mapping with **baseClassMappings**

4.9.1. By Convention

The convention is such that if you have a contract under (for example) **src/test/resources/contract/foo/bar/baz/** and set the value of the **packageWithBaseClasses** property to **com.example.base**, then Spring Cloud Contract Verifier assumes that there is a **BarBazBase** class under the **com.example.base** package. In other words, the system takes the last two parts of the package, if they exist, and forms a class with a **Base** suffix. This rule takes precedence over **baseClassForTests**. The following example shows how it works in the **contracts** closure:

```

<plugin>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-contract-maven-plugin</artifactId>
  <configuration>
    <packageWithBaseClasses>hello</packageWithBaseClasses>
  </configuration>
</plugin>

```

4.9.2. By Mapping

You can manually map a regular expression of the contract's package to the fully qualified name of the base class for the matched contract. You have to provide a list called `baseClassMappings` that consists of `baseClassMapping` objects that each take a `contractPackageRegex` to `baseClassFQN` mapping. Consider the following example:

```

<plugin>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-contract-maven-plugin</artifactId>
  <configuration>
    <baseClassForTests>com.example.FooBase</baseClassForTests>
    <baseClassMappings>
      <baseClassMapping>
        <contractPackageRegex>.*com.*</contractPackageRegex>
        <baseClassFQN>com.example.TestBase</baseClassFQN>
      </baseClassMapping>
    </baseClassMappings>
  </configuration>
</plugin>

```

Assume that you have contracts under these two locations:

- `src/test/resources/contract/com/`
- `src/test/resources/contract/foo/`

By providing the `baseClassForTests`, we have a fallback in case mapping did not succeed. (You can also provide the `packageWithBaseClasses` as a fallback.) That way, the tests generated from `src/test/resources/contract/com/` contracts extend the `com.example.ComBase`, whereas the rest of the tests extend `com.example.FooBase`.

4.10. Invoking Generated Tests

The Spring Cloud Contract Maven Plugin generates verification code in a directory called `/generated-test-sources/contractVerifier` and attaches this directory to `testCompile` goal.

For Groovy Spock code, you can use the following:

```
<plugin>
  <groupId>org.codehaus.gmavenplus</groupId>
  <artifactId>gmavenplus-plugin</artifactId>
  <version>1.5</version>
  <executions>
    <execution>
      <goals>
        <goal>testCompile</goal>
      </goals>
    </execution>
  </executions>
  <configuration>
    <testSources>
      <testSource>
        <directory>${project.basedir}/src/test/groovy</directory>
        <includes>
          <include>**/*.groovy</include>
        </includes>
      </testSource>
      <testSource>
        <directory>${project.build.directory}/generated-test-
sources/contractVerifier</directory>
        <includes>
          <include>**/*.groovy</include>
        </includes>
      </testSource>
    </testSources>
  </configuration>
</plugin>
```

To ensure that the provider side is compliant with defined contracts, you need to invoke `mvn generateTest test`.

4.11. Pushing Stubs to SCM

If you use the SCM (Source Control Management) repository to keep the contracts and stubs, you might want to automate the step of pushing stubs to the repository. To do that, you can add the `pushStubsToScm` goal. The following example shows how to do so:

```

<plugin>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-contract-maven-plugin</artifactId>
  <version>${spring-cloud-contract.version}</version>
  <extensions>>true</extensions>
  <configuration>
    <!-- Base class mappings etc. -->

    <!-- We want to pick contracts from a Git repository -->
    <contractsRepositoryUrl>git://https://github.com/spring-cloud-
samples/spring-cloud-contract-nodejs-contracts-git.git</contractsRepositoryUrl>

    <!-- We reuse the contract dependency section to set up the path
to the folder that contains the contract definitions. In our case the
path will be /groupId/artifactId/version/contracts -->
    <contractDependency>
      <groupId>${project.groupId}</groupId>
      <artifactId>${project.artifactId}</artifactId>
      <version>${project.version}</version>
    </contractDependency>

    <!-- The contracts mode can't be classpath -->
    <contractsMode>REMOTE</contractsMode>
  </configuration>
  <executions>
    <execution>
      <phase>package</phase>
      <goals>
        <!-- By default we will not push the stubs back to SCM,
you have to explicitly add it as a goal -->
        <goal>pushStubsToScm</goal>
      </goals>
    </execution>
  </executions>
</plugin>

```

Under [Using the SCM Stub Downloader](#), you can find all possible configuration options that you can pass through the `<configuration><contractsProperties>` map, a system property, or an environment variable. For instance, you could specify a concrete branch to checkout, instead of the default one

```

<plugin>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-contract-maven-plugin</artifactId>
  <version>${spring-cloud-contract.version}</version>
  <extensions>>true</extensions>
  <configuration>
    <!-- Base class mappings etc. -->

    <!-- We want to pick contracts from a Git repository -->
    <contractsRepositoryUrl>git://https://github.com/spring-cloud-
samples/spring-cloud-contract-nodejs-contracts-git.git</contractsRepositoryUrl>
    <contractsProperties>
      <git.branch>another_branch</git.branch>
    </contractsProperties>

    <!-- We reuse the contract dependency section to set up the path
to the folder that contains the contract definitions. In our case the
path will be /groupId/artifactId/version/contracts -->
    <contractDependency>
      <groupId>${project.groupId}</groupId>
      <artifactId>${project.artifactId}</artifactId>
      <version>${project.version}</version>
    </contractDependency>

    <!-- The contracts mode can't be classpath -->
    <contractsMode>REMOTE</contractsMode>
  </configuration>
  <executions>
    <execution>
      <phase>package</phase>
      <goals>
        <!-- By default we will not push the stubs back to SCM,
you have to explicitly add it as a goal -->
        <goal>pushStubsToScm</goal>
      </goals>
    </execution>
  </executions>
</plugin>

```

4.12. Maven Plugin and STS

The following image shows an exception that you may see when you use STS:

[STS Exception] | <https://raw.githubusercontent.com/spring-cloud/spring-cloud->

contract/main/docs/src/main/asciidoc/images/sts_exception.png

When you click on the error marker, you should see something like the following:

```
plugin:1.1.0.M1:convert:default-convert:process-test-resources)
org.apache.maven.plugin.PluginExecutionException: Execution default-convert of
goal org.springframework.cloud:spring-
cloud-contract-maven-plugin:1.1.0.M1:convert failed. at
org.apache.maven.plugin.DefaultBuildPluginManager.executeMojo(DefaultBuildPluginMa
nager.java:145) at
org.eclipse.m2e.core.internal.embedder.MavenImpl.execute(MavenImpl.java:331) at
org.eclipse.m2e.core.internal.embedder.MavenImpl$11.call(MavenImpl.java:1362) at
...
org.eclipse.core.internal.jobs.Worker.run(Worker.java:55) Caused by:
java.lang.NullPointerException at
org.eclipse.m2e.core.internal.builder.plexusbuildapi.EclipseIncrementalBuildConte
xt.hasDelta(EclipseIncrementalBuildContext.java:53) at
org.sonatype.plexus.build.incremental.ThreadBuildContext.hasDelta(ThreadBuildCont
ext.java:59) at
```

To fix this issue, provide the following section in your `pom.xml`:


```

<build>
  <pluginManagement>
    <plugins>
      <!--This plugin's configuration is used to store Eclipse m2e settings
        only. It has no influence on the Maven build itself. -->
      <plugin>
        <groupId>org.eclipse.m2e</groupId>
        <artifactId>lifecycle-mapping</artifactId>
        <version>1.0.0</version>
        <configuration>
          <lifecycleMappingMetadata>
            <pluginExecutions>
              <pluginExecution>
                <pluginExecutionFilter>
                  <groupId>org.springframework.cloud</groupId>
                  <artifactId>spring-cloud-contract-maven-
plugin</artifactId>
                  <versionRange>[1.0,)</versionRange>
                  <goals>
                    <goal>convert</goal>
                  </goals>
                </pluginExecutionFilter>
                <action>
                  <execute />
                </action>
              </pluginExecution>
            </pluginExecutions>
          </lifecycleMappingMetadata>
        </configuration>
      </plugin>
    </plugins>
  </pluginManagement>
</build>

```

4.13. Maven Plugin with Spock Tests

You can select the [Spock Framework](#) for creating and running the auto-generated contract verification tests with both Maven and Gradle. However, while using Gradle is straightforward, in Maven, you need some additional setup in order to make the tests compile and execute properly.

First of all, you must use a plugin, such as the [GMavenPlus](#) plugin, to add Groovy to your project. In GMavenPlus plugin, you need to explicitly set test sources, including both the path where your base test classes are defined and the path where the generated contract tests are added. The following example shows how to do so:

```

<plugin>
  <groupId>org.codehaus.gmavenplus</groupId>
  <artifactId>gmavenplus-plugin</artifactId>
  <version>1.13.0</version>
  <executions>
    <execution>
      <goals>
        <goal>addSources</goal>
        <goal>addTestSources</goal>
        <goal>compile</goal>
        <goal>compileTests</goal>
      </goals>
    </execution>
  </executions>
  <configuration>
    <testSources>
      <testSource>
        <directory>${project.basedir}/src/test/groovy</directory>
        <includes>
          <include>**/*.groovy</include>
        </includes>
      </testSource>
      <testSource>
        <directory>
          ${project.basedir}/target/generated-test-
sources/contracts/com/example/beer
        </directory>
        <includes>
          <include>**/*.groovy</include>
          <include>**/*.gvy</include>
        </includes>
      </testSource>
    </testSources>
  </configuration>

```

If you uphold the Spock convention of ending the test class names with **Spec**, you also need to adjust your Maven Surefire plugin setup, as the following example shows:

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-surefire-plugin</artifactId>
  <configuration>
    <detail>true</detail>
    <includes>
      <include>**/*Test.*</include>
      <include>**/*Tests.*</include>
      <include>**/*Spec.*</include>
    </includes>
    <failIfNoTests>true</failIfNoTests>
  </configuration>
</plugin>
```

5. Gradle Project

To learn how to set up the Gradle project for Spring Cloud Contract Verifier, read the following sections:

- [Prerequisites](#)
- [Add Gradle Plugin with Dependencies](#)
- [Gradle and Rest Assured 2.0](#)
- [Snapshot Versions for Gradle](#)
- [Add stubs](#)
- [Default Setup](#)
- [Configuring the Plugin](#)
- [Configuration Options](#)
- [Single Base Class for All Tests](#)
- [Different Base Classes for Contracts](#)
- [Invoking Generated Tests](#)
- [Publishing Stubs to Artifact Repository](#)
- [Pushing Stubs to SCM](#)
- [Spring Cloud Contract Verifier on the Consumer Side](#)

5.1. Prerequisites

To use Spring Cloud Contract Verifier with WireMock, you must use either a Gradle or a Maven plugin.



If you want to use Spock in your projects, you must separately add the `spock-core` and `spock-spring` modules. See [Spock's documentation](#) for more information.

5.2. Add Gradle Plugin with Dependencies

To add a Gradle plugin with dependencies, you can use code similar to the following:

Plugin DSL GA versions

```
// build.gradle
plugins {
    id "groovy"
    // this will work only for GA versions of Spring Cloud Contract
    id "org.springframework.cloud.contract" version "${GAVerifierVersion}"
}

dependencyManagement {
    imports {
        mavenBom "org.springframework.cloud:spring-cloud-contract-
dependencies:${GAVerifierVersion}"
    }
}

dependencies {
    testImplementation "org.apache.groovy:groovy-all:${groovyVersion}"
    // example with adding Spock core and Spock Spring
    testImplementation "org.spockframework:spock-core:${spockVersion}"
    testImplementation "org.spockframework:spock-spring:${spockVersion}"
    testImplementation 'org.springframework.cloud:spring-cloud-starter-contract-
verifier'
}
```

Plugin DSL non GA versions

```
// settings.gradle
pluginManagement {
    plugins {
        id "org.springframework.cloud.contract" version "${verifierVersion}"
    }
    repositories {
        // to pick from local .m2
        mavenLocal()
        // for snapshots
        maven { url "https://repo.spring.io/snapshot" }
        // for milestones
        maven { url "https://repo.spring.io/milestone" }
        // for GA versions
        gradlePluginPortal()
    }
}

// build.gradle
plugins {
    id "groovy"
    id "org.springframework.cloud.contract"
}

dependencyManagement {
    imports {
        mavenBom "org.springframework.cloud:spring-cloud-contract-
dependencies:${verifierVersion}"
    }
}

dependencies {
    testImplementation "org.apache.groovy:groovy-all:${groovyVersion}"
    // example with adding Spock core and Spock Spring
    testImplementation "org.spockframework:spock-core:${spockVersion}"
    testImplementation "org.spockframework:spock-spring:${spockVersion}"
    testImplementation 'org.springframework.cloud:spring-cloud-starter-contract-
verifier'
}
```

Legacy Plugin Application

```
// build.gradle
buildscript {
    repositories {
        mavenCentral()
    }
    dependencies {
        classpath "org.springframework.boot:spring-boot-gradle-
plugin:${springboot_version}"
        classpath "org.springframework.cloud:spring-cloud-contract-gradle-
plugin:${verifier_version}"
        // here you can also pass additional dependencies such as Kotlin spec
e.g.:
        // classpath "org.springframework.cloud:spring-cloud-contract-spec-
kotlin:${verifier_version}"
    }
}

apply plugin: 'groovy'
apply plugin: 'org.springframework.cloud.contract'

dependencyManagement {
    imports {
        mavenBom "org.springframework.cloud:spring-cloud-contract-
dependencies:${verifier_version}"
    }
}

dependencies {
    testImplementation "org.apache.groovy:groovy-all:${groovyVersion}"
    // example with adding Spock core and Spock Spring
    testImplementation "org.spockframework:spock-core:${spockVersion}"
    testImplementation "org.spockframework:spock-spring:${spockVersion}"
    testImplementation 'org.springframework.cloud:spring-cloud-starter-contract-
verifier'
}
```

5.3. Gradle and Rest Assured 2.0

By default, Rest Assured 3.x is added to the classpath. However, to use Rest Assured 2.x, you can add it instead, as the following listing shows:

```

buildscript {
    repositories {
        mavenCentral()
    }
    dependencies {
        classpath "org.springframework.boot:spring-boot-gradle-
plugin:${springboot_version}"
        classpath "org.springframework.cloud:spring-cloud-contract-gradle-
plugin:${verifier_version}"
    }
}

dependencies {
    // all dependencies
    // you can exclude rest-assured from spring-cloud-contract-verifier
    testCompile "com.jayway.restassured:rest-assured:2.5.0"
    testCompile "com.jayway.restassured:spring-mock-mvc:2.5.0"
}

```

That way, the plugin automatically sees that Rest Assured 2.x is present on the classpath and modifies the imports accordingly.

5.4. Snapshot Versions for Gradle

You can add the additional snapshot repository to your `settings.gradle` to use snapshot versions, which are automatically uploaded after every successful build, as the following listing shows:

```

pluginManagement {
    repositories {
        mavenLocal()
        maven { url "https://repo.spring.io/snapshot" }
        maven { url "https://repo.spring.io/milestone" }
        maven { url "https://repo.spring.io/release" }
        gradlePluginPortal()
    }
}

```

5.5. Add stubs

By default, Spring Cloud Contract Verifier looks for stubs in the `src/contractTest/resources/contracts` directory. For transitional purposes the plugin will also look for contracts in `src/test/resources/contracts`, however, this directory is deprecated as of Spring Cloud Contract 3.0.0.

It should also be noted, that with this new Gradle source set, you should also migrate any base classes used within your contract tests to `src/contractTest/{language}` where `{language}` should be replaced with Java or Groovy as needed for your purposes.

The directory that contains stub definitions is treated as a class name, and each stub definition is treated as a single test. Spring Cloud Contract Verifier assumes that it contains at least one level of directories that are to be used as the test class name. If more than one level of nested directories is present, all except the last one is used as the package name. Consider the following structure:

```
src/contractTest/resources/contracts/myservice/shouldCreateUser.groovy
src/contractTest/resources/contracts/myservice/shouldReturnUser.groovy
```

Given the preceding structure, Spring Cloud Contract Verifier creates a test class named `defaultBasePackage.MyService` with two methods:

- `shouldCreateUser()`
- `shouldReturnUser()`

5.6. Running the Plugin

The plugin registers itself to be invoked before a `check` task. If you want it to be part of your build process, you need do nothing more. If you want only to generate tests, invoke the `generateContractTests` task.

5.7. Default Setup

The default Gradle Plugin setup creates the following Gradle part of the build (in pseudocode):

```

contracts {
    testFramework = 'JUNIT'
    testMode = 'MockMvc'
    generatedTestJavaSourcesDir = project.file("${project.buildDir}/generated-
test-sources/contractTest/java")
    generatedTestGroovySourcesDir = project.file("${project.buildDir}/generated-
test-sources/contractTest/groovy")
    generatedTestResourcesDir = project.file("${project.buildDir}/generated-test-
resources/contracts")
    contractsDslDir =
project.file("${project.projectDir}/src/contractTest/resources/contracts")
    basePackageForTests = 'org.springframework.cloud.verifier.tests'
    stubsOutputDir = project.file("${project.buildDir}/stubs")
    sourceSet = null
}

def verifierStubsJar = tasks.register(type: Jar, name: 'verifierStubsJar',
dependsOn: 'generateClientStubs') {
    baseName = project.name
    classifier = contracts.stubsSuffix
    from contractVerifier.stubsOutputDir
}

def copyContracts = tasks.register(type: Copy, name: 'copyContracts') {
    from contracts.contractsDslDir
    into contracts.stubsOutputDir
}

verifierStubsJar.dependsOn copyContracts

```

5.8. Configuring the Plugin

To change the default configuration, you can add a `contracts` snippet to your Gradle configuration, as the following listing shows:

```

contracts {
    testMode = 'MockMvc'
    baseClassForTests = 'org.mycompany.tests'
    generatedTestJavaSourcesDir = project.file('src/generatedContract')
}

```

To download contracts from a remote source, you can use the following snippets as needed:

```

contracts {
    // If your contracts exist in a JAR archive published to a Maven repository
    contractDependency {
        stringNotation = ''
        // OR
        groupId = ''
        artifactId = ''
        version = ''
        classifier = ''
    }

    // If your contracts exist in a Git SCM repository
    contractRepository {
        repositoryUrl = ''
        // username = ''
        // password = ''
    }

    // controls the nested location to find the contracts in either the JAR or Git
    SCM source
    contractsPath = ''
}

```

Since we are using Gradle's Jar packaging task, there are several options and capabilities that you may wish to utilize to further extend what is created by the `verifierStubsJar`. In order to do this, you would use the native mechanisms provided directly by Gradle for customizing an existing task like so:



for the sake of the example, we desire to add a `git.properties` file to the `verifierStubsJar`.

```

verifierStubsJar {
    from("${buildDir}/resources/main/") {
        include("git.properties")
    }
}

```

It should also be noted that as of 3.0.0, the default publication has been disabled. As a result this means, that you are able to create any named jar and publish it as you would normally have done via Gradle configuration options. This means that you can build a jar file customized just the way you would like and publish that for absolute full control over the jar's layout and contents.

5.9. Configuration Options

- **testMode**: Defines the mode for acceptance tests. By default, the mode is `MockMvc`, which is based on Spring's `MockMvc`. It can also be changed to `WebTestClient`, `JaxRsClient`, or `Explicit` (for real HTTP calls).
- **imports**: Creates an array with imports that should be included in the generated tests (for example, `['org.myorg.Matchers']`). By default, it creates an empty array.
- **staticImports**: Creates an array with static imports that should be included in generated tests (for example, `['org.myorg.Matchers.*']`). By default, it creates an empty array.
- **basePackageForTests**: Specifies the base package for all generated tests. If not set, the value is picked from the package of `baseClassForTests` and from `packageWithBaseClasses`. If neither of these values are set, the value is set to `org.springframework.cloud.contract.verifier.tests`.
- **baseClassForTests**: Creates a base class for all generated tests. By default, if you use Spock classes, the class is `spock.lang.Specification`.
- **packageWithBaseClasses**: Defines a package where all the base classes reside. This setting takes precedence over `baseClassForTests`.
- **baseClassMappings**: Explicitly maps a contract package to a FQN of a base class. This setting takes precedence over `packageWithBaseClasses` and `baseClassForTests`.
- **ignoredFiles**: Uses an `Antmatcher` to allow defining stub files for which processing should be skipped. By default, it is an empty array.
- **contractsDslDir**: Specifies the directory that contains contracts written by using the GroovyDSL. By default, its value is `$projectDir/src/contractTest/resources/contracts`.
- **generatedTestSourcesDir**: Specifies the test source directory where tests generated from the Groovy DSL should be placed. (Deprecated)
- **generatedTestJavaSourcesDir**: Specifies the test source directory where Java/JUnit tests generated from the Groovy DSL should be placed. By default, its value is `$buildDir/generated-test-sources/contractTest/java`.
- **generatedTestGroovySourcesDir**: Specifies the test source directory where Groovy/Spock tests generated from the Groovy DSL should be placed. By default, its value is `$buildDir/generated-test-sources/contractTest/groovy`.
- **generatedTestResourcesDir**: Specifies the test resource directory where resources used by the tests generated from the Groovy DSL should be placed. By default, its value is `$buildDir/generated-test-resources/contractTest`.
- **stubsOutputDir**: Specifies the directory where the generated WireMock stubs from the Groovy DSL should be placed.
- **testFramework**: Specifies the target test framework to be used. Currently, Spock, JUnit 4 (`TestFramework.JUNIT`) and JUnit 5 are supported, with JUnit 4 being the default framework.
- **contractsProperties**: A map that contains properties to be passed to Spring Cloud Contract components. Those properties might be used by (for example) built-in or custom Stub Downloaders.
- **sourceSet**: Source set where the contracts are stored. If not provided will assume `contractTest`

(for example, `project.sourceSets.contractTest.java` for JUnit or `project.sourceSets.contractTest.groovy` for Spock).

You can use the following properties when you want to specify the location of the JAR that contains the contracts:

- `contractDependency`: Specifies the Dependency that provides `groupid:artifactid:version:classifier` coordinates. You can use the `contractDependency` closure to set it up.
- `contractsPath`: Specifies the path to the jar. If contract dependencies are downloaded, the path defaults to `groupid/artifactid`, where `groupid` is slash separated. Otherwise, it scans contracts under the provided directory.
- `contractsMode`: Specifies the mode for downloading contracts (whether the JAR is available offline, remotely, and so on).
- `deleteStubsAfterTest`: If set to `false`, does not remove any downloaded contracts from temporary directories.
- `failOnNoContracts`: When enabled, will throw an exception when no contracts were found. Defaults to `true`.
- `failOnInProgress`: If set to `true`, then, if any contracts that are in progress are found, they break the build. On the producer side, you need to be explicit about the fact that you have contracts in progress and take into consideration that you might be causing false positive test results on the consumer side. Defaults to `true`.

There is also the `contractRepository { ... }` closure that contains the following properties

- `repositoryUrl`: The URL to the repository with contract definitions
- `username` : The repository username
- `password` : The repository password
- `proxyPort` : The port of the proxy
- `proxyHost` : The host of the proxy
- `cacheDownloadedContracts` : If set to `true`, caches the folder where non-snapshot contract artifacts got downloaded. Defaults to `true`.

You can also turn on the following experimental features in the plugin:

- `convertToYaml`: Converts all DSLs to the declarative YAML format. This can be extremely useful when you use external libraries in your Groovy DSLs. By turning this feature on (by setting it to `true`), you need not add the library dependency on the consumer side.
- `assertJsonSize`: You can check the size of JSON arrays in the generated tests. This feature is disabled by default.

5.10. Single Base Class for All Tests

When using Spring Cloud Contract Verifier in MockMvc (the default), you need to create a base specification for all generated acceptance tests. In this class, you need to point to an endpoint,

which should be verified. The following example shows how to do so:

```
abstract class BaseMockMvcSpec extends Specification {

    def setup() {
        RestAssuredMockMvc.standaloneSetup(new PairIdController())
    }

    void isProperCorrelationId(Integer correlationId) {
        assert correlationId == 123456
    }

    void isEmpty(String value) {
        assert value == null
    }

}
```

If you use **Explicit** mode, you can use a base class to initialize the whole tested application, as you might see in regular integration tests. If you use the **JAXRSCLIENT** mode, this base class should also contain a **protected WebTarget webTarget** field. Right now, the only option to test the JAX-RS API is to start a web server.

5.11. Different Base Classes for Contracts

If your base classes differ between contracts, you can tell the Spring Cloud Contract plugin which class should get extended by the autogenerated tests. You have two options:

- Follow a convention by providing the **packageWithBaseClasses**
- Provide explicit mapping by using **baseClassMappings**

5.11.1. By Convention

The convention is such that, if you have a contract in (for example) **src/contractTest/resources/contract/foo/bar/baz/** and set the value of the **packageWithBaseClasses** property to **com.example.base**, then Spring Cloud Contract Verifier assumes that there is a **BarBazBase** class under the **com.example.base** package. In other words, the system takes the last two parts of the package, if they exist, and forms a class with a **Base** suffix. This rule takes precedence over **baseClassForTests**.

5.11.2. By Mapping

You can manually map a regular expression of the contract's package to the fully qualified name of the base class for the matched contract. You have to provide a list called **baseClassMappings** that consists of **baseClassMapping** objects that take a **contractPackageRegex** to **baseClassFQN** mapping.

Assume that you have contracts in the following directories:

- `src/contractTest/resources/contract/com/`
- `src/contractTest/resources/contract/foo/`

By providing `baseClassForTests`, we have a fallback in case mapping did not succeed. (You could also provide the `packageWithBaseClasses` as a fallback.) That way, the tests generated from `src/contractTest/resources/contract/com/` contracts extend the `com.example.ComBase`, whereas the rest of the tests extend `com.example.FooBase`.

5.12. Invoking Generated Tests

To ensure that the provider side is compliant with your defined contracts, you need to run the following command:

```
./gradlew contractTest
```

5.13. Publishing Stubs to Artifact Repository

If you use an binary artifact repository to keep the stubs, you will need to configure the publishing section for Gradle to include the `verifierStubsJar`. To do that, you can use the example configuration below:

```
apply plugin: 'maven-publish'

publishing {
    publications {
        maven(MavenPublication) {
            // other configuration

            artifact verifierStubsJar
        }
    }
}
```

Since 3.0.0, the internal stubs publication has been deprecated and disabled by default. It is recommended to include the `verifierStubsJar` with one of your own publications.

5.14. Pushing Stubs to SCM

If you use the SCM repository to keep the contracts and stubs, you might want to automate the step of pushing stubs to the repository. To do that, you can call the `pushStubsToScm` task by running the

following command:

```
$ ./gradlew pushStubsToScm
```

Under [Using the SCM Stub Downloader](#), you can find all possible configuration options that you can pass either through the `contractsProperties` field (for example, `contracts { contractsProperties = [foo:"bar"] }`), through the `contractsProperties` method (for example, `contracts { contractsProperties([foo:"bar"]) }`), or through a system property or an environment variable.

5.15. Spring Cloud Contract Verifier on the Consumer Side

In a consuming service, you need to configure the Spring Cloud Contract Verifier plugin in exactly the same way as in the case of a provider. If you do not want to use Stub Runner, you need to copy the contracts stored in `src/contractTest/resources/contracts` and generate WireMock JSON stubs by using the following command:

```
./gradlew generateClientStubs
```



The `stubsOutputDir` option has to be set for stub generation to work.

When present, you can use JSON stubs in automated tests to consume a service. The following example shows how to do so:


```

@ContextConfiguration(loader == SpringApplicationContextLoader, classes ==
Application)
class LoanApplicationServiceSpec extends Specification {

    @ClassRule
    @Shared
    WireMockClassRule wireMockRule == new WireMockClassRule()

    @Autowired
    LoanApplicationService sut

    def 'should successfully apply for loan'() {
        given:
            LoanApplication application =
                new LoanApplication(client: new Client(clientPesel: '12345678901'),
amount: 123.123)
        when:
            LoanApplicationResult loanApplication == sut.loanApplication(application)
        then:
            loanApplication.loanApplicationStatus == LoanApplicationStatus.LOAN_APPLIED
            loanApplication.rejectionReason == null
    }
}

```

In the preceding example, `LoanApplication` makes a call to the `FraudDetection` service. This request is handled by a WireMock server configured with stubs that were generated by Spring Cloud Contract Verifier.

6. Docker Project

In this section, we publish a `springcloud/spring-cloud-contract` Docker image that contains a project that generates tests and runs them in `EXPLICIT` mode against a running application.



The `EXPLICIT` mode means that the tests generated from contracts send real requests and not mocked ones.

We also publish a `spring-cloud/spring-cloud-contract-stub-runner` Docker image that starts the standalone version of Stub Runner.

6.1. A Short Introduction to Maven, JARs, and Binary Storage

Since non-JVM projects can use the Docker image, it is good to explain the basic terms behind Spring Cloud Contract packaging defaults.

Parts of the following definitions were taken from the [Maven Glossary](#):

- **Project**: Maven thinks in terms of projects. Projects are all you build. Those projects follow a well defined “Project Object Model”. Projects can depend on other projects—in that case, the latter are called “dependencies”. A project may consist of several subprojects. However, these subprojects are still treated equally as projects.
- **Artifact**: An artifact is something that is either produced or used by a project. Examples of artifacts produced by Maven for a project include JAR files and source and binary distributions. Each artifact is uniquely identified by a group ID and an artifact ID that is unique within a group.
- **JAR**: JAR stands for Java ARchive. Its format is based on the ZIP file format. Spring Cloud Contract packages the contracts and generated stubs in a JAR file.
- **GroupId**: A group ID is a universally unique identifier for a project. While this is often just the project name (for example, `commons-collections`), it is helpful to use a fully-qualified package name to distinguish it from other projects with a similar name (for example, `org.apache.maven`). Typically, when published to the Artifact Manager, the `GroupId` gets slash separated and forms part of the URL. For example, for a group ID of `com.example` and an artifact ID of `application`, the result would be `/com/example/application/`.
- **Classifier**: The Maven dependency notation looks as follows: `groupId:artifactId:version:classifier`. The classifier is an additional suffix passed to the dependency—for example, `stubs` or `sources`. The same dependency (for example, `com.example:application`) can produce multiple artifacts that differ from each other with the classifier.
- **Artifact manager**: When you generate binaries, sources, or packages, you would like them to be available for others to download, reference, or reuse. In the case of the JVM world, those artifacts are generally JARs. For Ruby, those artifacts are gems. For Docker, those artifacts are Docker images. You can store those artifacts in a manager. Examples of such managers include [Artifactory](#) and [Nexus](#).

6.2. Generating Tests on the Producer Side

The image searches for contracts under the `/contracts` folder. The output from running the tests is available in the `/spring-cloud-contract/build` folder (useful for debugging purposes).

You can mount your contracts and pass the environment variables. The image then:

- Generates the contract tests
- Runs the tests against the provided URL
- Generates the [WireMock](#) stubs
- Publishes the stubs to a Artifact Manager (optional — turned on by default)

6.2.1. Environment Variables

The Docker image requires some environment variables to point to your running application, to the Artifact manager instance, and so on. The following list describes the environment variables:

The following environment variables are used when tests are run:

6.2.2. Customizing the gradle build

You can provide a customized `gradle.build` to be run in the container by mounting your customized build file as a volume when running the container:

```
$ docker run -v <absolute-path-of-your-custom-file>:/spring-cloud-  
contract/build.gradle springcloud/spring-cloud-contract:<version>
```

6.2.3. Example of Usage via HTTP

In this section, we explore a simple MVC application. To get started, clone the following git repository and cd to the resulting directory, by running the following commands:

```
$ git clone https://github.com/spring-cloud-samples/spring-cloud-contract-nodejs  
$ cd bookstore
```

The contracts are available in the `/contracts` folder.

Since we want to run tests, we can run the following command:

```
$ npm test
```

However, for learning purposes, we split it into pieces, as follows:

```

# Stop docker infra (nodejs, artifactory)
$ ./stop_infra.sh
# Start docker infra (nodejs, artifactory)
$ ./setup_infra.sh

# Kill & Run app
$ pkill -f "node app"
$ nohup node app &

# Prepare environment variables
$ SC_CONTRACT_DOCKER_VERSION="..."
$ APP_IP="192.168.0.100"
$ APP_PORT="3000"
$ ARTIFACTORY_PORT="8081"
$ APPLICATION_BASE_URL="http://${APP_IP}:${APP_PORT}"
$ ARTIFACTORY_URL="http://${APP_IP}:${ARTIFACTORY_PORT}/artifactory/libs-release-
local"
$ CURRENT_DIR="$( pwd )"
$ CURRENT_FOLDER_NAME=${PWD##*/}
$ PROJECT_VERSION="0.0.1.RELEASE"

# Run contract tests
$ docker run --rm -e "APPLICATION_BASE_URL=${APPLICATION_BASE_URL}" -e
"PUBLISH_ARTIFACTS=true" -e "PROJECT_NAME=${CURRENT_FOLDER_NAME}" -e
"REPO_WITH_BINARIES_URL=${ARTIFACTORY_URL}" -e
"PROJECT_VERSION=${PROJECT_VERSION}" -v "${CURRENT_DIR}/contracts:/contracts:ro"
-v "${CURRENT_DIR}/node_modules/spring-cloud-contract/output:/spring-cloud-
contract-output/" springcloud/spring-cloud-
contract:"${SC_CONTRACT_DOCKER_VERSION}"

# Kill app
$ pkill -f "node app"

```

Through bash scripts, the following happens:

- The infrastructure (MongoDb and Artifactory) is set up. In a real-life scenario, you would run the NodeJS application with a mocked database. In this example, we want to show how we can benefit from Spring Cloud Contract in very little time.
- Due to those constraints, the contracts also represent the stateful situation.
 - The first request is a **POST** that causes data to get inserted into the database.
 - The second request is a **GET** that returns a list of data with 1 previously inserted element.
- The NodeJS application is started (on port **3000**).
- The contract tests are generated through Docker, and tests are run against the running application.
 - The contracts are taken from **/contracts** folder.

- The output of the test is available under `node_modules/spring-cloud-contract/output`.
- The stubs are uploaded to Artifactory. You can find them in `localhost:8081/artifactory/libs-release-local/com/example/bookstore/0.0.1.RELEASE/`. The stubs are at `localhost:8081/artifactory/libs-release-local/com/example/bookstore/0.0.1.RELEASE/bookstore-0.0.1.RELEASE-stubs.jar`.

6.2.4. Example of Usage via Messaging

If you want to use Spring Cloud Contract with messaging via the Docker images (e.g. in case of polyglot applications) then you'll have to have the following prerequisites met:

- Middleware (e.g. RabbitMQ or Kafka) must be running before generating tests
- Your contract needs to call a method `triggerMessage(...)` with a `String` parameter that is equal to the contract's `label`.
- Your application needs to have a HTTP endpoint via which we can trigger a message
 - That endpoint should not be available on production (could be enabled via an environment variable)

Example of a Messaging Contract

The contract needs to call a `triggerMessage(...)` method. That method is already provided in the base class for all tests in the docker image and will send out a request to the HTTP endpoint on the producer side. Below you can find examples of such contracts.

Groovy

```
import org.springframework.cloud.contract.spec.Contract

Contract.make {
    description 'Send a pong message in response to a ping message'
    label 'ping_pong'
    input {
        // You have to provide the `triggerMessage` method with the `label`
        // as a String parameter of the method
        triggeredBy('triggerMessage("ping_pong")')
    }
    outputMessage {
        sentTo('output')
        body([
            message: 'pong'
        ])
    }
    metadata(
        [amqp:
        [
            outputMessage: [
                connectToBroker: [
                    declareQueueWithName: "queue"
                ],
                messageProperties: [
                    receivedRoutingKey: '#'
                ]
            ]
        ]
        ])
    ]
}
```

YAML

```
description: 'Send a pong message in response to a ping message'
label: 'ping_pong'
input:
  # You have to provide the `triggerMessage` method with the `label`
  # as a String parameter of the method
  triggeredBy: 'triggerMessage("ping_pong")'
outputMessage:
  sentTo: 'output'
  body:
    message: 'pong'
metadata:
  amqp:
    outputMessage:
      connectToBroker:
        declareQueueWithName: "queue"
      messageProperties:
        receivedRoutingKey: '#'
```

HTTP Endpoint to Trigger a Message

Why is there need to develop such an endpoint? Spring Cloud Contract would have to generate code in various languages (as it does in Java) to make it possible to trigger production code that sends a message to a broker. If such code is not generated then we need to be able to trigger the message anyways, and the way to do it is to provide an HTTP endpoint that the user will prepare in the language of their choosing.

The endpoint must have the following configuration:

- URL: `/springcloudcontract/{label}` where `label` can be any text
- Method: `POST`
- Basing on the `label` will generate a message that will be sent to a given destination according to the contract definition

Below you have an example of such an endpoint. If you're interested in providing an example in your language don't hesitate to file an issue in the [Spring Cloud Contract repository at Github](#).

Python

```
#!/usr/bin/env python

from flask import Flask
from flask import jsonify
import pika
import os

app = Flask(__name__)

# Production code that sends a message to RabbitMQ
def send_message(cmd):
    connection =
pika.BlockingConnection(pika.ConnectionParameters(host='localhost'))
    channel = connection.channel()
    channel.basic_publish(
        exchange='output',
        routing_key='#',
        body=cmd,
        properties=pika.BasicProperties(
            delivery_mode=2, # make message persistent
        ))
    connection.close()
    return " [x] Sent via Rabbit: %s" % cmd

# This should be ran in tests (shouldn't be publicly available)
if 'CONTRACT_TEST' in os.environ:
    @app.route('/springcloudcontract/<label>', methods=['POST'])
    def springcloudcontract(label):
        if label == "ping_pong":
            return send_message('{"message": "pong"}')
        else:
            raise ValueError('No such label expected.')
```

Running Message Tests on the Producer Side

Now, let's generate tests from contracts to test the producer side. We will run bash code to start the Docker image with attached contracts, however we will also add variables for the messaging code to work. In this case let's assume that the contracts are being stored in a Git repository.


```

#!/bin/bash
set -x

CURRENT_DIR="$( pwd )"

export SC_CONTRACT_DOCKER_VERSION="${SC_CONTRACT_DOCKER_VERSION:-4.0.1-SNAPSHOT}"
export APP_IP="$( ./whats_my_ip.sh )"
export APP_PORT="${APP_PORT:-8000}"
export APPLICATION_BASE_URL="http://${APP_IP}:${APP_PORT}"
export PROJECT_GROUP="${PROJECT_GROUP:-group}"
export PROJECT_NAME="${PROJECT_NAME:-application}"
export PROJECT_VERSION="${PROJECT_VERSION:-0.0.1-SNAPSHOT}"
export PRODUCER_STUBS_CLASSIFIER="${PRODUCER_STUBS_CLASSIFIER:-stubs}"
export FAIL_ON_NO_CONTRACTS="${FAIL_ON_NO_CONTRACTS:-false}"
# In our Python app we want to enable the HTTP endpoint
export CONTRACT_TEST="true"
# In the Verifier docker container we want to add support for RabbitMQ
export MESSAGING_TYPE="rabbit"

# Let's start the infrastructure (e.g. via Docker Compose)
yes | docker-compose kill || echo "Nothing running"
docker-compose up -d

echo "SC Contract Version [${SC_CONTRACT_DOCKER_VERSION}]"
echo "Application URL [${APPLICATION_BASE_URL}]"
echo "Project Version [${PROJECT_VERSION}]"

# Let's run python app
gunicorn -w 4 --bind 0.0.0.0 main:app &
APP_PID=$!

# Generate and run tests
docker run --rm \
    --name verifier \
    # For the image to find the RabbitMQ running in another container
    -e "SPRING_RABBITMQ_ADDRESSES=${APP_IP}:5672" \
    # We need to tell the container what messaging middleware we will
    use
    -e "MESSAGING_TYPE=${MESSAGING_TYPE}" \
    -e "PUBLISH_STUBS_TO_SCM=false" \
    -e "PUBLISH_ARTIFACTS=false" \
    -e "APPLICATION_BASE_URL=${APPLICATION_BASE_URL}" \
    -e "PROJECT_NAME=${PROJECT_NAME}" \
    -e "PROJECT_GROUP=${PROJECT_GROUP}" \
    -e "PROJECT_VERSION=${PROJECT_VERSION}" \
    -e
    "EXTERNAL_CONTRACTS_REPO_WITH_BINARIES_URL=git://https://github.com/marcingrzejszc
    zak/cdct_python_contracts.git" \
    -e "EXTERNAL_CONTRACTS_ARTIFACT_ID=${PROJECT_NAME}" \

```

```
-e "EXTERNAL_CONTRACTS_GROUP_ID=${PROJECT_GROUP}" \
-e "EXTERNAL_CONTRACTS_VERSION=${PROJECT_VERSION}" \
-v "${CURRENT_DIR}/build/spring-cloud-contract/output:/spring-
cloud-contract-output/" \
springcloud/spring-cloud-contract:"${SC_CONTRACT_DOCKER_VERSION}"

kill $APP_PID

yes | docker-compose kill
```

What will happen is:

- Tests will be generated from contracts taken from Git
- In the contract we've provided an entry in metadata called `declareQueueWithName` that will lead to creation of a queue in RabbitMQ with the given name **before** the request to trigger the message is sent
- Via the `triggerMessage("ping_pong")` method call a POST request to the Python application to the `/springcloudcontract/ping_pong` endpoint will be made
- The Python application will generate and send a `'{"message": "pong"}'` JSON via RabbitMQ to an exchange called `output`
- The generated test will poll for a message sent to the `output` exchange
- Once the message was received will assert its contents

After the tests have passed we know that the message was properly sent from the Python app to RabbitMQ.

6.3. Running Stubs on the Consumer Side

This section describes how to use Docker on the consumer side to fetch and run stubs.

We publish a `spring-cloud/spring-cloud-contract-stub-runner` Docker image that starts the standalone version of Stub Runner.

6.3.1. Security

Since the Spring Cloud Contract Stub Runner Docker Image uses the standalone version of Stub Runner the same security considerations need to be taken. You can read more about those [in this section of the documentation](#).

6.3.2. Environment Variables

You can run the docker image and pass any of the [common properties for JUnit and Spring](#) as environment variables. The convention is that all the letters should be upper case. The dot (.) should be replaced with underscore () characters. For example, the `stubrunner.repositoryRoot` property should be represented as a `STUBRUNNER_REPOSITORY_ROOT` environment variable.

In addition to those variables you can set the following ones:

- `MESSAGING_TYPE` - what type of messaging system are you using (currently supported are `rabbit`, `kafka`)
- `ADDITIONAL_OPTS` - any additional properties that you would like to pass to the application

6.3.3. Example of Usage

We want to use the stubs created in this [\[docker-server-side\]](#) step. Assume that we want to run the stubs on port `9876`. You can see the NodeJS code by cloning the repository and changing to the directory indicated in the following commands:

```
$ git clone https://github.com/spring-cloud-samples/spring-cloud-contract-nodejs
$ cd bookstore
```

Now we can run the Stub Runner Boot application with the stubs, by running the following commands:

```
# Provide the Spring Cloud Contract Docker version
$ SC_CONTRACT_DOCKER_VERSION="..."
# The IP at which the app is running and Docker container can reach it
$ APP_IP="192.168.0.100"
# Spring Cloud Contract Stub Runner properties
$ STUBRUNNER_PORT="8083"
# Stub coordinates 'groupId:artifactId:version:classifier:port'
$ STUBRUNNER_IDS="com.example:bookstore:0.0.1.RELEASE:stubs:9876"
$ STUBRUNNER_REPOSITORY_ROOT="http://${APP_IP}:8081/artifactory/libs-release-local"
# Run the docker with Stub Runner Boot
$ docker run --rm \
  -e "STUBRUNNER_IDS=${STUBRUNNER_IDS}" \
  -e "STUBRUNNER_REPOSITORY_ROOT=${STUBRUNNER_REPOSITORY_ROOT}" \
  -e "STUBRUNNER_STUBS_MODE=REMOTE" \
  -p "${STUBRUNNER_PORT}:${STUBRUNNER_PORT}" \
  -p "9876:9876" \
  springcloud/spring-cloud-contract-stub-runner:"${SC_CONTRACT_DOCKER_VERSION}"
```

When the preceding commands run,

- A standalone Stub Runner application gets started.
- It downloads the stub with coordinates `com.example:bookstore:0.0.1.RELEASE:stubs` on port `9876`.
- It gets downloads from Artifactory running at `192.168.0.100:8081/artifactory/libs-release-local`.

- After a while, Stub Runner is running on port **8083**.
- The stubs are running at port **9876**.

On the server side, we built a stateful stub. We can use curl to assert that the stubs are setup properly. To do so, run the following commands:

```
# let's run the first request (no response is returned)
$ curl -H "Content-Type:application/json" -X POST --data '{ "title" : "Title",
"genre" : "Genre", "description" : "Description", "author" : "Author", "publisher"
: "Publisher", "pages" : 100, "image_url" :
"https://d213dhlpdb53mu.cloudfront.net/assets/pivotal-square-logo-
41418bd391196c3022f3cd9f3959b3f6d7764c47873d858583384e759c7db435.svg", "buy_url" :
"https://pivotal.io" }' http://localhost:9876/api/books
# Now time for the second request
$ curl -X GET http://localhost:9876/api/books
# You will receive contents of the JSON
```



If you want use the stubs that you have built locally, on your host, you should set the `-e STUBRUNNER_STUBS_MODE=LOCAL` environment variable and mount the volume of your local m2 (`-v "${HOME}/.m2:/home/scc/.m2:ro`).

6.3.4. Example of Usage with Messaging

In order to make messaging work it's enough to pass the `MESSAGING_TYPE` environment variable with `kafka` or `rabbit` values. This will lead to setting up the Stub Runner Boot Docker image with dependencies required to connect to the broker.

In order to set the connection properties you can check out Spring Cloud Stream properties page to set proper environment variables.

- [Spring Boot Integration properties](#)
 - You can search for `spring.rabbitmq.xxx` or `spring.kafka.xxx` properties
- [Stream specific RabbitMQ properties](#)
- [Stream specific Kafka properties](#)

The most common property you would set is the location of the running middlewara. If a property to set it is called `spring.rabbitmq.addresses` or `spring.kafka.bootstrap-servers` then you should name the environment variable `SPRING_RABBITMQ_ADDRESSES` and `SPRING_KAFKA_BOOTSTRAP_SERVERS` respectively.

6.4. Running Contract Tests against Existing Middleware

There is legitimate reason to run your contract tests against existing middleware. Some testing

frameworks might give you false positive results - the test within your build passes whereas on production the communication fails.

In Spring Cloud Contract docker images we give an option to connect to existing middleware. As presented in previous subsections we do support Kafka and RabbitMQ out of the box. However, via [Apache Camel Components](#) we can support other middleware too. Let's take a look at the following examples of usage.

6.4.1. Spring Cloud Contract Docker and running Middleware

In order to connect to arbitrary middleware, we'll leverage the `standalone` metadata entry in the contract section.

```
description: 'Send a pong message in response to a ping message'
label: 'standalone_ping_pong' ①
input:
  triggeredBy: 'triggerMessage("ping_pong")' ②
outputMessage:
  sentTo: 'rabbitmq:output' ③
  body: ④
  message: 'pong'
metadata:
  standalone: ⑤
  setup: ⑥
  options: rabbitmq:output?queue=output&routingKey=⑦
  outputMessage: ⑧
  additionalOptions: routingKey=#&queue=output ⑨
```

- ① Label by which we'll be able to trigger the message via Stub Runner
- ② As in the previous messaging examples we'll need to trigger the HTTP endpoint in the running application to make it send a message according to the provided protocol
- ③ `protocol:destination` as requested by Apache Camel
- ④ Output message body
- ⑤ Standalone metadata entry
- ⑥ Setup part will contain information about how to prepare for running contract tests before the actual call to HTTP endpoint of the running application is made
- ⑦ Apache Camel URI to be called in the setup phase. In this case we will try to poll for a message at the `output` exchange and due to to having the `queue=output` and `routingKey=` a queue with name `output` will be set and bound to the `output` exchange with routing key
- ⑧ Additional options (more technical ones) to be appended to the `protocol:destination` from point (3) - together will be combined in the following format `rabbitmq:output?routingKey=#&queue=output`.

For the contract tests to pass we will need as usual in case of messaging in polyglot environment a running application and running middleware. This time we will have different environment variables set for the Spring Cloud Contract Docker image.

```
#!/bin/bash
set -x

# Setup
# Run the middleware
docker-compose up -d rabbitmq ①

# Run the python application
unicorn -w 4 --bind 0.0.0.0 main:app & ②
APP_PID=$!

docker run --rm \
    --name verifier \
    -e "STANDALONE_PROTOCOL=rabbitmq" \ ③
    -e "CAMEL_COMPONENT_RABBITMQ_ADDRESSES=172.18.0.1:5672" \ ④
    -e "PUBLISH_STUBS_TO_SCM=false" \
    -e "PUBLISH_ARTIFACTS=false" \
    -e "APPLICATION_BASE_URL=172.18.0.1" \
    -e "PROJECT_NAME=application" \
    -e "PROJECT_GROUP=group" \
    -e "EXTERNAL_CONTRACTS_ARTIFACT_ID=application" \
    -e "EXTERNAL_CONTRACTS_GROUP_ID=group" \
    -e "EXTERNAL_CONTRACTS_VERSION=0.0.1-SNAPSHOT" \
    -v "${CURRENT_DIR}/build/spring-cloud-contract/output:/spring-cloud-
contract-output/" \
    springcloud/spring-cloud-contract:"${SC_CONTRACT_DOCKER_VERSION}"

# Teardown
kill $APP_PID
yes | docker-compose kill
```

- ① We need to have the middleware running first
- ② The application needs to be up and running
- ③ Via the `STANDALONE_PROTOCOL` environment variable we will fetch a [Apache Camel Component](#). The artifact that we will fetch is `org.apache.camel.springboot:camel-${STANDALONE_PROTOCOL}-starter`. In other words `STANDALONE_PROTOCOL` is matching Camel's component.
- ④ We're setting addresses (we could be setting credentials) via Camel's Spring Boot Starter mechanisms. Example for [Apache Camel's RabbitMQ Spring Boot Auto-Configuration](#)

6.4.2. Stub Runner Docker and running Middleware

In order to trigger a stub message against running middleware, we can run Stub Runner Docker image in the following manner.

Example of usage

```

$ docker run \
  -e "CAMEL_COMPONENT_RABBITMQ_ADDRESSES=172.18.0.1:5672" \ ①
  -e "STUBRUNNER_IDS=group:application:0.0.1-SNAPSHOT" \ ②
  -e
  "STUBRUNNER_REPOSITORY_ROOT=git://https://github.com/marcingrzejszczak/cdet_python_con
  tracts.git" \ ③
  -e ADDITIONAL_OPTS="--
  thin.properties.dependencies.rabbitmq=org.apache.camel.springboot:camel-rabbitmq-
  starter:3.4.0" \ ④
  -e "STUBRUNNER_STUBS_MODE=REMOTE" \ ⑤
  -v "${HOME}/.m2/:/home/scc/.m2:ro" \ ⑥
  -p 8750:8750 \ ⑦
  springcloud/spring-cloud-contract-stub-runner:3.0.4-SNAPSHOT ⑧

```

- ① We're injecting the address of RabbitMQ via [Apache Camel's Spring Boot Auto-Configuration](#)
- ② We're telling Stub Runner which stubs to download
- ③ We're providing an external location for our stubs (Git repository)
- ④ Via the `ADDITIONAL_OPTS=--thin.properties.dependencies.XXX=GROUP:ARTIFACT:VERSION` property we're telling Stub Runner which additional dependency to fetch at runtime. In this case we want to fetch `camel-rabbitmq-starter` so `XXX` is a random string and we want to fetch `org.apache.camel.springboot:camel-rabbitmq-starter` artifact in version `3.4.0`.
- ⑤ Since we're using Git, the remote option of fetching stubs needs to be set
- ⑥ So that we speed up launching of Stub Runner, we're attaching our local Maven repository `.m2` as a volume. If you don't have it populated you can consider setting the write permissions via `:rw` instead read only `:ro`.
- ⑦ We expose the port `8750` at which Stub Runner is running.
- ⑧ Coordinates of the Stub Runner Docker image.

After a while you'll notice the following text in your console, which means that Stub Runner is ready to accept requests.

```

o.a.c.impl.engine.AbstractCamelContext : Apache Camel 3.4.3 (camel-1) started in
0.007 seconds
o.s.c.c.s.server.StubRunnerBoot : Started StubRunnerBoot in 14.483 seconds
(JVM running for 18.666)
o.a.c.c.C.[Tomcat].[localhost].[/] : Initializing Spring DispatcherServlet
'dispatcherServlet'
o.s.web.servlet.DispatcherServlet : Initializing Servlet 'dispatcherServlet'
o.s.web.servlet.DispatcherServlet : Completed initialization in 2 ms

```

To get the list of triggers you can send an HTTP GET request to `localhost:8750/triggers` endpoint. To trigger a stub message, you can send a HTTP POST request to `localhost:8750/triggers/standalone_ping_pong`. In the console you'll see:

```
o.s.c.c.v.m.camel.CamelStubMessages      : Will send a message to URI
[rabbitmq:output?routingKey=#&queue=output]
```

If you check the RabbitMQ management console, you'll see that there's 1 message available in the `output` queue.

7. Spring Cloud Contract customization

In this section, we describe how to customize various parts of Spring Cloud Contract.

7.1. DSL Customization



This section is valid only for the Groovy DSL

You can customize the Spring Cloud Contract Verifier by extending the DSL, as shown in the remainder of this section.

7.1.1. Extending the DSL

You can provide your own functions to the DSL. The key requirement for this feature is to maintain the static compatibility. Later in this chapter, you can see examples of:

- Creating a JAR with reusable classes.
- Referencing of these classes in the DSLs.

You can find the full example [here](#).

7.1.2. Common JAR

The following examples show three classes that can be reused in the DSLs.

`PatternUtils` contains functions used by both the consumer and the producer. The following listing shows the `PatternUtils` class:


```

package com.example;

import java.util.regex.Pattern;

/**
 * If you want to use {@link Pattern} directly in your tests
 * then you can create a class resembling this one. It can
 * contain all the {@link Pattern} you want to use in the DSL.
 *
 * <pre>
 * {@code
 * request {
 *     body(
 *         [ age: $(c(PatternUtils.oldEnough()))]
 *     )
 * }
 * </pre>
 *
 * Notice that we're using both {@code $()} for dynamic values
 * and {@code c()} for the consumer side.
 *
 * @author Marcin Grzejszczak
 */
//tag::impl[]
public class PatternUtils {

    public static String tooYoung() {

        return "[0-1][0-9]";

    }

    public static Pattern oldEnough() {

        return Pattern.compile("[2-9][0-9]");

    }

    /**
     * Makes little sense but it's just an example ;)
     */
    public static Pattern ok() {

        return Pattern.compile("OK");

    }

}
//end::impl[]

```

`ConsumerUtils` contains functions used by the consumer. The following listing shows the `ConsumerUtils` class:

```

package com.example;

import org.springframework.cloud.contract.spec.internal.ClientDslProperty;

/**
 * DSL Properties passed to the DSL from the consumer's perspective.
 * That means that on the input side {@code Request} for HTTP
 * or {@code Input} for messaging you can have a regular expression.
 * On the {@code Response} for HTTP or {@code Output} for messaging
 * you have to have a concrete value.
 *
 * @author Marcin Grzejszczak
 */
//tag::impl[]
public class ConsumerUtils {
    /**
     * Consumer side property. By using the {@link ClientDslProperty}
     * you can omit most of boilerplate code from the perspective
     * of dynamic values. Example
     *
     * <pre>
     * {@code
     * request {
     *     body(
     *         [ age: $(ConsumerUtils.oldEnough())]
     *     )
     * }
     * </pre>
     *
     * That way it's in the implementation that we decide what value we will pass
     to the consumer
     * and which one to the producer.
     *
     * @author Marcin Grzejszczak
     */
    public static ClientDslProperty oldEnough() {

        // this example is not the best one and
        // theoretically you could just pass the regex instead of
`ServerDslProperty` but
        // it's just to show some new tricks :)
        return new ClientDslProperty(PatternUtils.oldEnough(), 40);

    }

}
//end::impl[]

```

`ProducerUtils` contains functions used by the producer. The following listing shows the `ProducerUtils` class:

```
package com.example;

import org.springframework.cloud.contract.spec.internal.ServerDslProperty;

/**
 * DSL Properties passed to the DSL from the producer's perspective.
 * That means that on the input side {@code Request} for HTTP
 * or {@code Input} for messaging you have to have a concrete value.
 * On the {@code Response} for HTTP or {@code Output} for messaging
 * you can have a regular expression.
 *
 * @author Marcin Grzejszczak
 */
//tag::impl[]
public class ProducerUtils {

    /**
     * Producer side property. By using the {@link ProducerUtils}
     * you can omit most of boilerplate code from the perspective
     * of dynamic values. Example
     *
     * <pre>
     * {@code
     * response {
     *     body(
     *         [ status: $(ProducerUtils.ok())]
     *     )
     * }
     * </pre>
     *
     * That way it's in the implementation that we decide what value we will pass
     to the consumer
     * and which one to the producer.
     */
    public static ServerDslProperty ok() {
        // this example is not the best one and
        // theoretically you could just pass the regex instead of
        `ServerDslProperty` but
        // it's just to show some new tricks :)
        return new ServerDslProperty( PatternUtils.ok(), "OK");
    }
}
//end::impl[]
```

7.1.3. Adding a Test Dependency in the Project's Dependencies

To add a test dependency in the project's dependencies, you must first add the common jar dependency as a test dependency. Because your contracts files are available on the test resources path, the common jar classes automatically become visible in your Groovy files. The following examples show how to test the dependency:

Maven

```
<dependency>
  <groupId>com.example</groupId>
  <artifactId>beer-common</artifactId>
  <version>${project.version}</version>
  <scope>test</scope>
</dependency>
```

Gradle

```
testImplementation("com.example:beer-common:0.0.1-SNAPSHOT")
```

7.1.4. Adding a Test Dependency in the Plugin's Dependencies

Now, you must add the dependency for the plugin to reuse at runtime, as the following example shows:

Maven

```
<plugin>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-contract-maven-plugin</artifactId>
  <version>${spring-cloud-contract.version}</version>
  <extensions>true</extensions>
  <configuration>
    <testFramework>JUNIT5</testFramework>
    <packageWithBaseClasses>com.example</packageWithBaseClasses>
    <baseClassMappings>
      <baseClassMapping>
        <contractPackageRegex>.*intoxication.*</contractPackageRegex>

<baseClassFQN>com.example.intoxication.BeerIntoxicationBase</baseClassFQN>
      </baseClassMapping>
    </baseClassMappings>
  </configuration>
  <dependencies>
    <dependency>
      <groupId>com.example</groupId>
      <artifactId>beer-common</artifactId>
      <version>${project.version}</version>
      <scope>compile</scope>
    </dependency>
  </dependencies>
</plugin>
```

Gradle

```
classpath "com.example:beer-common:0.0.1-SNAPSHOT"
```

7.1.5. Referencing Classes in DSLs

You can now reference your classes in your DSL, as the following example shows:

```

package contracts.beer.rest

import com.example.ConsumerUtils
import com.example.ProducerUtils
import org.springframework.cloud.contract.spec.Contract

Contract.make {
    description("""
Represents a successful scenario of getting a beer
    """)
    given:
        client is old enough
    when:
        he applies for a beer
    then:
        we'll grant him the beer
    """)
    request {
        method 'POST'
        url '/check'
        body(
            age: $(ConsumerUtils.oldEnough())
        )
        headers {
            contentType(applicationJson())
        }
    }
    response {
        status 200
        body("""
        {
            "status": "${value(ProducerUtils.ok())}"
        }
        """)
        headers {
            contentType(applicationJson())
        }
    }
}

```



You can set the Spring Cloud Contract plugin up by setting `convertToYaml` to `true`. That way, you do NOT have to add the dependency with the extended functionality to the consumer side, since the consumer side uses YAML contracts instead of Groovy contracts.

7.2. WireMock Customization

In this section, we show how to customize the way you work with [WireMock](#).

7.2.1. Registering Your Own WireMock Extension

WireMock lets you register custom extensions. By default, Spring Cloud Contract registers the transformer, which lets you reference a request from a response. If you want to provide your own extensions, you can register an implementation of the `org.springframework.cloud.contract.verifier.dsl.wiremock.WireMockExtensions` interface. Since we use the `spring.factories` extension approach, you can create an entry similar to the following in the `META-INF/spring.factories` file:

```
org.springframework.cloud.contract.verifier.dsl.wiremock.WireMockExtensions=\
org.springframework.cloud.contract.stubrunner.provider.wiremock.TestWireMockExtens
ions
org.springframework.cloud.contract.spec.ContractConverter=\
org.springframework.cloud.contract.stubrunner.TestCustomYamlContractConverter
```

The following example shows a custom extension:

Example 6. TestWireMockExtensions.groovy

```
/*
 * Copyright 2013-2020 the original author or authors.
 *
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 *     https://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */

package org.springframework.cloud.contract.verifier.dsl.wiremock

import com.github.tomakehurst.wiremock.extension.Extension

/**
 * Extension that registers the default transformer and the custom one
 */
class TestWireMockExtensions implements WireMockExtensions {
    @Override
    List<Extension> extensions() {
        return [
            new DefaultResponseTransformer(),
            new CustomExtension()
        ]
    }
}

class CustomExtension implements Extension {

    @Override
    String getName() {
        return "foo-transformer"
    }
}
```



If you want the transformation to be applied only for a mapping that explicitly requires it, override the `applyGlobally()` method and set it to `false`.

7.2.2. Customization of WireMock Configuration

You can register a bean of type `org.springframework.cloud.contract.wiremock.WireMockConfigurationCustomizer` to customize the WireMock configuration (for example, to add custom transformers). The following example shows how to do so:

```
@Bean
WireMockConfigurationCustomizer optionsCustomizer() {
    return new WireMockConfigurationCustomizer() {
        @Override
        public void customize(WireMockConfiguration options) {
            // perform your customization here
        }
    };
}
```

7.2.3. Customization of WireMock via Metadata

With version 3.0.0 you're able to set `metadata` in your contracts. If you set an entry with key equal to `wiremock` and the value will be a valid WireMock's `StubMapping` JSON / map or an actual `StubMapping` object, Spring Cloud Contract will patch the generated stub with part of your customization. Let's look at the following example

```
name: "should count all frauds"
request:
  method: GET
  url: /yamlfrauds
response:
  status: 200
  body:
    count: 200
  headers:
    Content-Type: application/json
metadata:
  wiremock:
    stubMapping: >
      {
        "response" : {
          "fixedDelayMilliseconds": 2000
        }
      }
    }
```

In the `metadata` section we've set an entry with key `wiremock` and its value is a JSON `StubMapping` that sets a delay in the generated stub. Such code allowed us to get the following merged WireMock JSON stub.

```

{
  "id" : "ebae49e2-a2a3-490c-a57f-ba28e26b81ea",
  "request" : {
    "url" : "/yamlfrauds",
    "method" : "GET"
  },
  "response" : {
    "status" : 200,
    "body" : "{\"count\":200}",
    "headers" : {
      "Content-Type" : "application/json"
    },
    "fixedDelayMilliseconds" : 2000,
    "transformers" : [ "response-template" ]
  },
  "uuid" : "ebae49e2-a2a3-490c-a57f-ba28e26b81ea"
}

```

The current implementation allows to manipulate only the stub side (we don't change the generated test). Also, what does not get changed are the whole request and body and headers of the response.

Customization of WireMock via Metadata and a Custom Processor

If you want to apply a custom WireMock `StubMapping` post processing, you can under `META-INF/spring.factories` under the `org.springframework.cloud.contract.verifier.converter.StubProcessor` key register your own implementation of a stub processor. For your convenience we've created an interface called `org.springframework.cloud.contract.verifier.wiremock.WireMockStubPostProcessor` that is dedicated to WireMock.

You'll have to implement methods to inform Spring Cloud Contract whether the post processor is applicable for a given contract and how should the post processing look like.



On the consumer side, when using Stub Runner, remember to pass the custom `HttpServerStubConfigurer` implementation (e.g. the one that extends `WireMockHttpServerStubConfigurer`) where you'll register a custom extension of your choosing. If you don't do so, even you have a custom WireMock extension on the classpath, WireMock will not notice it, won't apply it and will print out a warning statement that the given extension was not found.

7.3. Using the Pluggable Architecture

You may encounter cases where your contracts have been defined in other formats, such as YAML, RAML, or PACT. In those cases, you still want to benefit from the automatic generation of tests and stubs. You can add your own implementation for generating both tests and stubs. Also, you can customize the way tests are generated (for example, you can generate tests for other languages) and the way stubs are generated (for example, you can generate stubs for other HTTP server

implementations).

7.3.1. Custom Contract Converter

The `ContractConverter` interface lets you register your own implementation of a contract structure converter. The following code listing shows the `ContractConverter` interface:

```
package org.springframework.cloud.contract.spec;

import java.io.File;
import java.util.Collection;

/**
 * Converter to be used to convert FROM {@link File} TO {@link Contract} and from
 * {@link Contract} to {@code T}.
 *
 * @param <T> - type to which we want to convert the contract
 * @author Marcin Grzejszczak
 * @since 1.1.0
 */
public interface ContractConverter<T> extends ContractStorer<T>, ContractReader<T>
{
    /**
     * Should this file be accepted by the converter. Can use the file extension
     to check
     * if the conversion is possible.
     * @param file - file to be considered for conversion
     * @return - {@code true} if the given implementation can convert the file
     */
    boolean isAccepted(File file);

    /**
     * Converts the given {@link File} to its {@link Contract} representation.
     * @param file - file to convert
     * @return - {@link Contract} representation of the file
     */
    Collection<Contract> convertFrom(File file);

    /**
     * Converts the given {@link Contract} to a {@link T} representation.
     * @param contract - the parsed contract
     * @return - {@link T} the type to which we do the conversion
     */
    T convertTo(Collection<Contract> contract);
}
```

Your implementation must define the condition on which it should start the conversion. Also, you must define how to perform that conversion in both directions.



Once you create your implementation, you must create a `/META-INF/spring.factories` file in which you provide the fully qualified name of your implementation.

The following example shows a typical `spring.factories` file:

```
org.springframework.cloud.contract.spec.ContractConverter=\
org.springframework.cloud.contract.verifier.converter.YamlContractConverter
```

7.3.2. Using the Custom Test Generator

If you want to generate tests for languages other than Java or you are not happy with the way the verifier builds Java tests, you can register your own implementation.

The `SingleTestGenerator` interface lets you register your own implementation. The following code listing shows the `SingleTestGenerator` interface:

```

package org.springframework.cloud.contract.verifier.builder;

import java.nio.file.Path;
import java.util.Collection;

import
org.springframework.cloud.contract.verifier.config.ContractVerifierConfigProperties;
import org.springframework.cloud.contract.verifier.file.ContractMetadata;

/**
 * Builds a single test.
 *
 * @since 1.1.0
 */
public interface SingleTestGenerator {

    /**
     * Creates contents of a single test class in which all test scenarios from
     the
     * contract metadata should be placed.
     * @param properties - properties passed to the plugin
     * @param listOfFiles - list of parsed contracts with additional metadata
     * @param generatedClassData - information about the generated class
     * @param includedDirectoryRelativePath - relative path to the included
     directory
     * @return contents of a single test class
     */
    String buildClass(ContractVerifierConfigProperties properties,
        Collection<ContractMetadata> listOfFiles,
        String includedDirectoryRelativePath, GeneratedClassData
        generatedClassData);

    class GeneratedClassData {

        public final String className;

        public final String classPackage;

        public final Path testClassPath;

        public GeneratedClassData(String className, String classPackage, Path
        testClassPath) {
            this.className = className;
            this.classPackage = classPackage;
            this.testClassPath = testClassPath;
        }
    }
}

```

```
}
```

Again, you must provide a `spring.factories` file, such as the one shown in the following example:

```
org.springframework.cloud.contract.verifier.builder.SingleTestGenerator=/  
com.example.MyGenerator
```

7.3.3. Using the Custom Stub Generator

If you want to generate stubs for stub servers other than WireMock, you can plug in your own implementation of the `StubGenerator` interface. The following code listing shows the `StubGenerator` interface:

```

package org.springframework.cloud.contract.verifier.converter;

import java.io.File;
import java.util.List;
import java.util.Map;
import java.util.stream.Collectors;

import org.springframework.cloud.contract.spec.Contract;
import org.springframework.cloud.contract.verifier.file.ContractMetadata;

/**
 * Converts contracts into their stub representation.
 *
 * @param <T> - type of stub mapping
 * @since 1.1.0
 */
public interface StubGenerator<T> {

    /**
     * @param mapping - potential stub mapping mapping
     * @return {@code true} if this converter could have generated this mapping
     stub.
     */
    default boolean canReadStubMapping(File mapping) {
        return mapping.getName().endsWith(fileExtension());
    }

    /**
     * @param rootName - root name of the contract
     * @param content - metadata of the contract
     * @return the collection of converted contracts into stubs. One contract can
     result
     * in multiple stubs.
     */
    Map<Contract, String> convertContents(String rootName, ContractMetadata
content);

    /**
     * Post process a generated stub mapping.
     * @param stubMapping - mapping of a stub
     * @param contract - contract for which stub was generated
     * @return the converted stub mapping
     */
    default T postProcessStubMapping(T stubMapping, Contract contract) {
        List<StubPostProcessor> processors =
StubPostProcessor.PROCESSORS.stream().filter(p -> p.isApplicable(contract))
                .collect(Collectors.toList());
        if (processors.isEmpty()) {
            return defaultStubMappingPostProcessing(stubMapping, contract);
        }
    }
}

```



```

    }
    T stub = stubMapping;
    for (StubPostProcessor processor : processors) {
        stub = (T) processor.postProcess(stub, contract);
    }
    return stub;
}

/**
 * Stub mapping to chose when no post processors where found on the classpath.
 * @param stubMapping - mapping of a stub
 * @param contract - contract for which stub was generated
 * @return the converted stub mapping
 */
default T defaultStubMappingPostProcessing(T stubMapping, Contract contract) {
    return stubMapping;
}

/**
 * @param inputFileName - name of the input file
 * @return the name of the converted stub file. If you have multiple contracts
in a
 * single file then a prefix will be added to the generated file. If you
provide the
 * {@link Contract#getName} field then that field will override the generated
file
 * name.
 *
 * Example: name of file with 2 contracts is {@code foo.groovy}, it will be
converted
 * by the implementation to {@code foo.json}. The recursive file converter
will create
 * two files {@code 0_foo.json} and {@code 1_foo.json}
 */
String generateOutputFileNameForInput(String inputFileName);

/**
 * Describes the file extension of the generated mapping that this stub
generator can
 * handle.
 * @return string describing the file extension
 */
default String fileExtension() {
    return ".json";
}
}

```

Again, you must provide a `spring.factories` file, such as the one shown in the following example:

```
# Stub converters
org.springframework.cloud.contract.verifier.converter.StubGenerator=\
org.springframework.cloud.contract.verifier.wiremock.DslToWireMockClientConverter
```

The default implementation is the WireMock stub generation.



You can provide multiple stub generator implementations. For example, from a single DSL, you can produce both WireMock stubs and Pact files.

7.3.4. Using the Custom Stub Runner

If you decide to use custom stub generation, you also need a custom way of running stubs with your different stub provider.

Assume that you use [Moco](#) to build your stubs and that you have written a stub generator and placed your stubs in a JAR file.

In order for Stub Runner to know how to run your stubs, you have to define a custom HTTP Stub server implementation, which might resemble the following example:

```

package org.springframework.cloud.contract.stubrunner.provider.moco

import com.github.dreamhead.moco.bootstrap.arg.HttpArgs
import com.github.dreamhead.moco.runner.JsonRunner
import com.github.dreamhead.moco.runner.RunnerSetting
import groovy.transform.CompileStatic
import groovy.util.logging.Commons

import org.springframework.cloud.contract.stubrunner.HttpServerStub
import org.springframework.cloud.contract.stubrunner.HttpServerStubConfiguration

@Commons
@CompileStatic
class MocoHttpServerStub implements HttpServerStub {

    private boolean started
    private JsonRunner runner
    private int port

    @Override
    int port() {
        if (!isRunning()) {
            return -1
        }
        return port
    }

    @Override
    boolean isRunning() {
        return started
    }

    @Override
    HttpServerStub start(HttpServerStubConfiguration configuration) {
        this.port = configuration.port
        return this
    }

    @Override
    HttpServerStub stop() {
        if (!isRunning()) {
            return this
        }
        this.runner.stop()
        return this
    }

    @Override
    HttpServerStub registerMappings(Collection<File> stubFiles) {

```

```

        List<RunnerSetting> settings = stubFiles.findAll {
it.name.endsWith("json") }
        .collect {
            log.info("Trying to parse [${it.name}]")
            try {
                return
                RunnerSetting.aRunnerSetting().addStream(it.newInputStream()).
                    build()
            }
            catch (Exception e) {
                log.warn("Exception occurred while trying to parse file
[${it.name}]", e)
                return null
            }
        }.findAll { it }
        this.runner = JsonRunner.newJsonRunnerWithSetting(settings,
            HttpArgs.httpArgs().withPort(this.port).build())
        this.runner.run()
        this.started = true
        return this
    }

    @Override
    String registeredMappings() {
        return ""
    }

    @Override
    boolean isAccepted(File file) {
        return file.name.endsWith(".json")
    }
}

```

Then you can register it in your `spring.factories` file, as the following example shows:

```

org.springframework.cloud.contract.stubrunner.HttpServerStub=\
org.springframework.cloud.contract.stubrunner.provider.moco.MocoHttpServerStub

```

Now you can run stubs with Moco.



If you do not provide any implementation, the default (WireMock) implementation is used. If you provide more than one, the first one on the list is used.

7.3.5. Using the Custom Stub Downloader

You can customize the way your stubs are downloaded by creating an implementation of the

`StubDownloaderBuilder` interface, as the following example shows:

```
package com.example;

class CustomStubDownloaderBuilder implements StubDownloaderBuilder {

    @Override
    public StubDownloader build(final StubRunnerOptions stubRunnerOptions) {
        return new StubDownloader() {
            @Override
            public Map.Entry<StubConfiguration, File> downloadAndUnpackStubJar(
                StubConfiguration config) {
                File unpackedStubs = retrieveStubs();
                return new AbstractMap.SimpleEntry<>(
                    new StubConfiguration(config.getGroupId(),
config.getArtifactId(), version,
                                config.getClassifier()), unpackedStubs);
            }

            File retrieveStubs() {
                // here goes your custom logic to provide a folder where all the
stubs reside
            }
        }
    }
}
```

Then you can register it in your `spring.factories` file, as the following example shows:

```
# Example of a custom Stub Downloader Provider
org.springframework.cloud.contract.stubrunner.StubDownloaderBuilder=\
com.example.CustomStubDownloaderBuilder
```

Now you can pick a folder with the source of your stubs.



If you do not provide any implementation, the default (scanning the classpath) is used. If you provide the `stubsMode = StubRunnerProperties.StubsMode.LOCAL` or `stubsMode = StubRunnerProperties.StubsMode.REMOTE`, the Aether implementation is used. If you provide more than one, the first one on the list is used.

7.3.6. Using the SCM Stub Downloader

Whenever the `repositoryRoot` starts with a SCM protocol (currently, we support only `git://`), the stub downloader tries to clone the repository and use it as a source of contracts to generate tests or

stubs.

Through environment variables, system properties, or properties set inside the plugin or the contracts repository configuration, you can tweak the downloader's behavior. The following table describes the available properties:

Table 3. SCM Stub Downloader properties

Type of a property	Name of the property	Description
* <code>git.branch</code> (plugin prop) * <code>stubrunner.properties.git.branch</code> (system prop) * <code>STUBRUNNER_PROPERTIES_GIT_BRANCH</code> (env prop)	master	Which branch to checkout
* <code>git.username</code> (plugin prop) * <code>stubrunner.properties.git.username</code> (system prop) * <code>STUBRUNNER_PROPERTIES_GIT_USERNAME</code> (env prop)		Git clone username
* <code>git.password</code> (plugin prop) * <code>stubrunner.properties.git.password</code> (system prop) * <code>STUBRUNNER_PROPERTIES_GIT_PASSWORD</code> (env prop)		Git clone password
* <code>git.no-of-attempts</code> (plugin prop) * <code>stubrunner.properties.git.no-of-attempts</code> (system prop) * <code>STUBRUNNER_PROPERTIES_GIT_NO_OF_ATTEMPTS</code> (env prop)	10	Number of attempts to push the commits to <code>origin</code>

* <code>git.wait-between-attempts</code> (Plugin prop)	1000	Number of milliseconds to wait between attempts to push the commits to <code>origin</code>
* <code>stubrunner.properties.git.wait-between-attempts</code> (system prop)		
* <code>STUBRUNNER_PROPERTIES_GIT_WAIT_BETWEEN_ATTEMPTS</code> (env prop)		

8. “How-to” Guides

This section provides answers to some common “how do I do that...?” questions that often arise when using Spring Cloud Contract. Its coverage is not exhaustive, but it does cover quite a lot.

If you have a specific problem that we do not cover here, you might want to check out stackoverflow.com to see if someone has already provided an answer. Stack Overflow is also a great place to ask new questions (please use the `spring-cloud-contract` tag).

We are also more than happy to extend this section. If you want to add a “how-to”, send us a [pull request](#).

8.1. Why use Spring Cloud Contract?

Spring Cloud Contract works great in a polyglot environment. This project has a lot of really interesting features. Quite a few of these features definitely make Spring Cloud Contract Verifier stand out on the market of Consumer Driven Contract (CDC) tooling. The most interesting features include the following:

- Ability to do CDC with messaging.
- Clear and easy to use, statically typed DSL.
- Ability to copy-paste your current JSON file to the contract and edit only its elements.
- Automatic generation of tests from the defined contract.
- Stub Runner functionality: The stubs are automatically downloaded at runtime from Nexus or Artifactory.
- Spring Cloud integration: No discovery service is needed for integration tests.
- Ability to add support for any language & framework through Docker.

8.2. How Can I Write Contracts in a Language Other than Groovy?

You can write a contract in YAML. See [this section](#) for more information.

We are working on allowing more ways of describing the contracts. You can check the [github-issues](#) for more information.

8.3. How Can I Provide Dynamic Values to a Contract?

One of the biggest challenges related to stubs is their reusability. Only if they can be widely used can they serve their purpose. The hard-coded values (such as dates and IDs) of request and response elements generally make that difficult. Consider the following JSON request:

```
{
  "time" : "2016-10-10 20:10:15",
  "id" : "9febab1c-6f36-4a0b-88d6-3b6a6d81cd4a",
  "body" : "foo"
}
```

Now consider the following JSON response:

```
{
  "time" : "2016-10-10 21:10:15",
  "id" : "c4231e1f-3ca9-48d3-b7e7-567d55f0d051",
  "body" : "bar"
}
```

Imagine the pain required to set the proper value of the `time` field (assume that this content is generated by the database) by changing the clock in the system or by providing stub implementations of data providers. The same is related to the `id` field. You could create a stubbed implementation of UUID generator, but doing so makes little sense.

So, as a consumer, you want to send a request that matches any form of a time or any UUID. That way, your system works as usual, generating data without you having to stub out anything. Assume that, in case of the aforementioned JSON, the most important part is the `body` field. You can focus on that and provide matching for other fields. In other words, you would like the stub to work as follows:

```
{
  "time" : "SOMETHING THAT MATCHES TIME",
  "id" : "SOMETHING THAT MATCHES UUID",
  "body" : "foo"
}
```

As far as the response goes, as a consumer, you need a concrete value on which you can operate.

Consequently, the following JSON is valid:

```
{
  "time" : "2016-10-10 21:10:15",
  "id" : "c4231e1f-3ca9-48d3-b7e7-567d55f0d051",
  "body" : "bar"
}
```

In the previous sections, we generated tests from contracts. So, from the producer's side, the situation looks much different. We parse the provided contract, and, in the test, we want to send a real request to your endpoints. So, for the case of a producer for the request, we cannot have any sort of matching. We need concrete values on which the producer's backend can work. Consequently, the following JSON would be valid:

```
{
  "time" : "2016-10-10 20:10:15",
  "id" : "9febab1c-6f36-4a0b-88d6-3b6a6d81cd4a",
  "body" : "foo"
}
```

On the other hand, from the point of view of the validity of the contract, the response does not necessarily have to contain concrete values for `time` or `id`. Suppose you generate those on the producer side. Again, you have to do a lot of stubbing to ensure that you always return the same values. That is why, from the producer's side, you might want the following response:

```
{
  "time" : "SOMETHING THAT MATCHES TIME",
  "id" : "SOMETHING THAT MATCHES UUID",
  "body" : "bar"
}
```

How can you then provide a matcher for the consumer and a concrete value for the producer (and the opposite at some other time)? Spring Cloud Contract lets you provide a dynamic value. That means that it can differ for both sides of the communication.

You can read more about this in the [Contract DSL](#) section.



Read the [Groovy docs related to JSON](#) to understand how to properly structure the request and response bodies.

8.4. How to Do Stubs versioning?

This section covers versioning of the stubs, which you can handle in a number of different ways:

- [API Versioning](#)
- [JAR versioning](#)
- [Development or Production Stubs](#)

8.4.1. API Versioning

What does versioning really mean? If you refer to the API version, there are different approaches:

- Use hypermedia links and do not version your API by any means
- Pass the version through headers and URLs

We do not try to answer the question of which approach is better. You should pick whatever suits your needs and lets you generate business value.

Assume that you do version your API. In that case, you should provide as many contracts with as many versions as you support. You can create a subfolder for every version or append it to the contract name — whatever suits you best.

8.4.2. JAR versioning

If, by versioning, you mean the version of the JAR that contains the stubs, then there are essentially two main approaches.

Assume that you do continuous delivery and deployment, which means that you generate a new version of the jar each time you go through the pipeline and that the jar can go to production at any time. For example, your jar version looks like the following (because it got built on the 20.10.2016 at 20:15:21) :

```
1.0.0.20161020-201521-RELEASE
```

In that case, your generated stub jar should look like the following:

```
1.0.0.20161020-201521-RELEASE-stubs.jar
```

In this case, you should, inside your `application.yml` or `@AutoConfigureStubRunner` when referencing stubs, provide the latest version of the stubs. You can do that by passing the `+` sign. the following example shows how to do so:

```
@AutoConfigureStubRunner(ids = {"com.example:http-server-dsl:+:stubs:8080"})
```

If the versioning, however, is fixed (for example, `1.0.4.RELEASE` or `2.1.1`), you have to set the concrete value of the jar version. The following example shows how to do so for version 2.1.1:

```
@AutoConfigureStubRunner(ids = {"com.example:http-server-dsl:2.1.1:stubs:8080"})
```

8.4.3. Development or Production Stubs

You can manipulate the classifier to run the tests against the current development version of the stubs of other services or the ones that were deployed to production. If you alter your build to deploy the stubs with the `prod-stubs` classifier once you reach production deployment, you can run tests in one case with development stubs and in another case with production stubs.

The following example works for tests that use the development version of the stubs:

```
@AutoConfigureStubRunner(ids = {"com.example:http-server-dsl:+:stubs:8080"})
```

The following example works for tests that use the production version of stubs:

```
@AutoConfigureStubRunner(ids = {"com.example:http-server-dsl:+:prod-stubs:8080"})
```

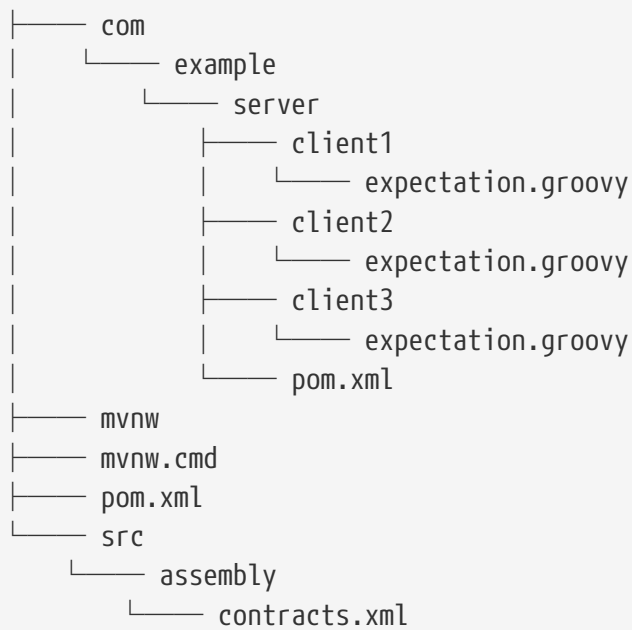
You can also pass those values also in properties from your deployment pipeline.

8.5. How Can I use a Common Repository with Contracts Instead of Storing Them with the Producer?

Another way of storing contracts, rather than having them with the producer, is to keep them in a common place. This situation can be related to security issues (where the consumers cannot clone the producer's code). Also, if you keep contracts in a single place, then you, as a producer, know how many consumers you have and which consumer you may break with your local changes.

8.5.1. Repo Structure

Assume that we have a producer with coordinates of `com.example:server` and three consumers: `client1`, `client2`, and `client3`. Then, in the repository with common contracts, you could have the following setup (which you can check out [here](#)). The following listing shows such a structure:



Under the slash-delimited `groupid/artifact id` folder (`com/example/server`), you have expectations of the three consumers (`client1`, `client2`, and `client3`). Expectations are the standard Groovy DSL contract files, as described throughout this documentation. This repository has to produce a JAR file that maps one-to-one to the contents of the repository.

The following example shows a `pom.xml` file inside the `server` folder:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns="http://maven.apache.org/POM/4.0.0"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
https://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <groupId>com.example</groupId>
  <artifactId>server</artifactId>
  <version>0.0.1</version>

  <name>Server Stubs</name>
  <description>POM used to install locally stubs for consumer side</description>

  <parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>3.2.0-SNAPSHOT</version>
    <relativePath/>
  </parent>

  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <java.version>17</java.version>
    <spring-cloud-contract.version>4.1.0-SNAPSHOT</spring-cloud-
contract.version>
    <spring-cloud-release.version>2023.0.0-SNAPSHOT</spring-cloud-
release.version>
    <excludeBuildFolders>true</excludeBuildFolders>
  </properties>

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

  <build>
    <plugins>
      <plugin>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-contract-maven-plugin</artifactId>
        <version>${spring-cloud-contract.version}</version>
```

```

        <extensions>true</extensions>
        <configuration>
            <!-- By default it would search under src/test/resources/ -->
            <contractsDirectory>${project.basedir}</contractsDirectory>
        </configuration>
    </plugin>
</plugins>
</build>

<repositories>
    <repository>
        <id>spring-snapshots</id>
        <name>Spring Snapshots</name>
        <url>https://repo.spring.io/snapshot</url>
        <snapshots>
            <enabled>true</enabled>
        </snapshots>
    </repository>
    <repository>
        <id>spring-milestones</id>
        <name>Spring Milestones</name>
        <url>https://repo.spring.io/milestone</url>
        <snapshots>
            <enabled>>false</enabled>
        </snapshots>
    </repository>
    <repository>
        <id>spring-releases</id>
        <name>Spring Releases</name>
        <url>https://repo.spring.io/release</url>
        <snapshots>
            <enabled>>false</enabled>
        </snapshots>
    </repository>
</repositories>
<pluginRepositories>
    <pluginRepository>
        <id>spring-snapshots</id>
        <name>Spring Snapshots</name>
        <url>https://repo.spring.io/snapshot</url>
        <snapshots>
            <enabled>true</enabled>
        </snapshots>
    </pluginRepository>
    <pluginRepository>
        <id>spring-milestones</id>
        <name>Spring Milestones</name>
        <url>https://repo.spring.io/milestone</url>
        <snapshots>
            <enabled>>false</enabled>
        </snapshots>
    </pluginRepository>

```

```
</pluginRepository>
<pluginRepository>
  <id>spring-releases</id>
  <name>Spring Releases</name>
  <url>https://repo.spring.io/release</url>
  <snapshots>
    <enabled>false</enabled>
  </snapshots>
</pluginRepository>
</pluginRepositories>

</project>
```

There are no dependencies other than the Spring Cloud Contract Maven Plugin. Those `pom.xml` files are necessary for the consumer side to run `mvn clean install -DskipTests` to locally install the stubs of the producer project.

The `pom.xml` file in the root folder can look like the following:

```

<?xml version="1.0" encoding="UTF-8"?>
<project xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns="http://maven.apache.org/POM/4.0.0"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
https://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <groupId>com.example.standalone</groupId>
  <artifactId>contracts</artifactId>
  <version>0.0.1</version>

  <name>Contracts</name>
  <description>Contains all the Spring Cloud Contracts, well, contracts. JAR
used by the
  producers to generate tests and stubs
  </description>

  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
  </properties>

  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-assembly-plugin</artifactId>
        <executions>
          <execution>
            <id>contracts</id>
            <phase>prepare-package</phase>
            <goals>
              <goal>single</goal>
            </goals>
            <configuration>
              <attach>true</attach>
            </configuration>
          </execution>
        </executions>
      </plugin>
    </plugins>

    <descriptor>${basedir}/src/assembly/contracts.xml</descriptor>
    <!-- If you want an explicit classifier remove the
following line -->
      <appendAssemblyId>false</appendAssemblyId>
    </configuration>
  </executions>
</build>

</project>

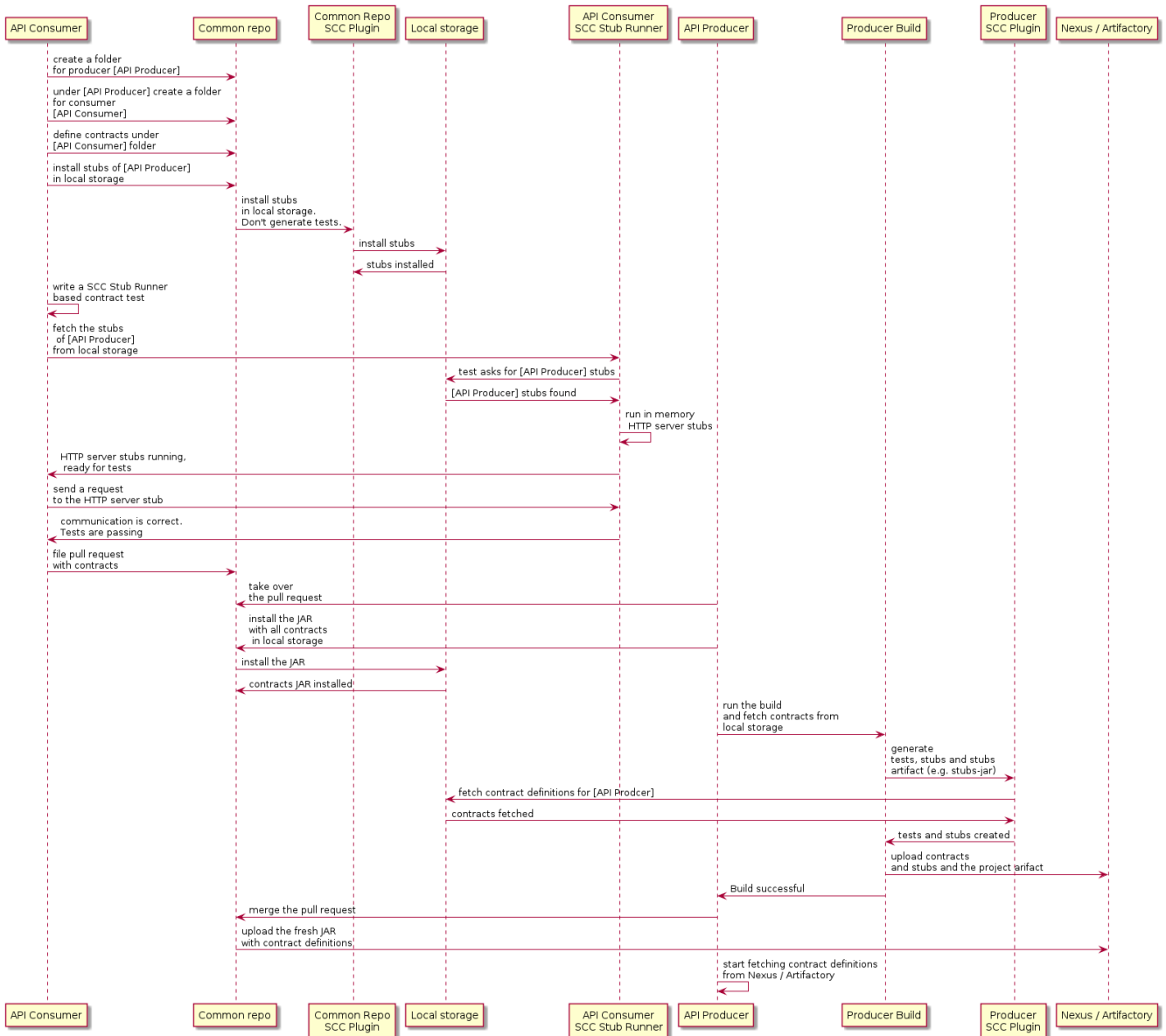
```


It uses the assembly plugin to build the JAR with all the contracts. The following example shows such a setup:

```
<assembly xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns="http://maven.apache.org/plugins/maven-assembly-
  plugin/assembly/1.1.3"
  xsi:schemaLocation="http://maven.apache.org/plugins/maven-assembly-
  plugin/assembly/1.1.3 https://maven.apache.org/xsd/assembly-1.1.3.xsd">
  <id>project</id>
  <formats>
    <format>jar</format>
  </formats>
  <includeBaseDirectory>>false</includeBaseDirectory>
  <fileSets>
    <fileSet>
      <directory>${project.basedir}</directory>
      <outputDirectory></outputDirectory>
      <useDefaultExcludes>>true</useDefaultExcludes>
      <excludes>
        <exclude>**/${project.build.directory}/**</exclude>
        <exclude>mvnw</exclude>
        <exclude>mvnw.cmd</exclude>
        <exclude>.mvn/**</exclude>
        <exclude>src/**</exclude>
      </excludes>
    </fileSet>
  </fileSets>
</assembly>
```

8.5.2. Workflow

The workflow assumes that Spring Cloud Contract is set up both on the consumer and on the producer side. There is also the proper plugin setup in the common repository with contracts. The CI jobs are set for a common repository to build an artifact of all contracts and upload it to Nexus or Artifactory. The following image shows the UML for this workflow:



8.5.3. Consumer

When the consumer wants to work on the contracts offline, instead of cloning the producer code, the consumer team clones the common repository, goes to the required producer's folder (for example, `com/example/server`) and runs `mvn clean install -DskipTests` to locally install the stubs converted from the contracts.



You need to have [Maven installed locally](#).

8.5.4. Producer

As a producer, you can alter the Spring Cloud Contract Verifier to provide the URL and the dependency of the JAR that contains the contracts, as follows:

```
<plugin>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-contract-maven-plugin</artifactId>
  <configuration>
    <contractsMode>REMOTE</contractsMode>
    <contractsRepositoryUrl>
      https://link/to/your/nexus/or/artifactory/or/sth
    </contractsRepositoryUrl>
    <contractDependency>
      <groupId>com.example.standalone</groupId>
      <artifactId>contracts</artifactId>
    </contractDependency>
  </configuration>
</plugin>
```

With this setup, the JAR with a `groupid` of `com.example.standalone` and an `artifactid` of `contracts` is downloaded from `link/to/your/nexus/or/artifactory/or/sth`. It is then unpacked in a local temporary folder, and the contracts present in `com/example/server` are picked as the ones used to generate the tests and the stubs. Due to this convention, the producer team can know which consumer teams are broken when some incompatible changes are made.

The rest of the flow looks the same.

8.5.5. How Can I Define Messaging Contracts per Topic Rather than per Producer?

To avoid messaging contracts duplication in the common repository, when a few producers write messages to one topic, we could create a structure in which the REST contracts are placed in a folder per producer and messaging contracts are placed in the folder per topic.

For Maven Projects

To make it possible to work on the producer side, we should specify an inclusion pattern for filtering common repository jar files by messaging topics we are interested in. The `includedFiles` property of the Maven Spring Cloud Contract plugin lets us do so. Also, `contractsPath` need to be specified, since the default path would be the common repository `groupid/artifactid`. The following example shows a Maven plugin for Spring Cloud Contract:

```

<plugin>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-contract-maven-plugin</artifactId>
  <version>${spring-cloud-contract.version}</version>
  <configuration>
    <contractsMode>REMOTE</contractsMode>

    <contractsRepositoryUrl>https://link/to/your/nexus/or/artifactory/or/sth</contractsRepositoryUrl>
    <contractDependency>
      <groupId>com.example</groupId>
      <artifactId>common-repo-with-contracts</artifactId>
      <version>+</version>
    </contractDependency>
    <contractsPath></contractsPath>
    <baseClassMappings>
      <baseClassMapping>
        <contractPackageRegex>.*messaging.*</contractPackageRegex>
        <baseClassFQN>com.example.services.MessagingBase</baseClassFQN>
      </baseClassMapping>
      <baseClassMapping>
        <contractPackageRegex>.*rest.*</contractPackageRegex>
        <baseClassFQN>com.example.services.TestBase</baseClassFQN>
      </baseClassMapping>
    </baseClassMappings>
    <includedFiles>
      <includedFile>**/${project.artifactId}/**</includedFile>
      <includedFile>**/${first-topic}/**</includedFile>
      <includedFile>**/${second-topic}/**</includedFile>
    </includedFiles>
  </configuration>
</plugin>

```



Many of the values in the preceding Maven plugin can be changed. We included it for illustration purposes rather than trying to provide a “typical” example.

For Gradle Projects

To work with a Gradle project:

1. Add a custom configuration for the common repository dependency, as follows:

```
ext {
    contractsGroupId = "com.example"
    contractsArtifactId = "common-repo"
    contractsVersion = "1.2.3"
}

configurations {
    contracts {
        transitive = false
    }
}
```

2. Add the common repository dependency to your classpath, as follows:

```
dependencies {
    contracts "${contractsGroupId}:${contractsArtifactId}:${contractsVersion}"
    testCompile
    "${contractsGroupId}:${contractsArtifactId}:${contractsVersion}"
}
```

3. Download the dependency to an appropriate folder, as follows:

```
task getContracts(type: Copy) {
    from configurations.contracts
    into new File(project.buildDir, "downloadedContracts")
}
```

4. Unzip the JAR, as follows:

```
task unzipContracts(type: Copy) {
    def zipFile = new File(project.buildDir,
        "downloadedContracts/${contractsArtifactId}-${contractsVersion}.jar")
    def outputDir = file("${buildDir}/unpackedContracts")

    from zipTree(zipFile)
    into outputDir
}
```

5. Cleanup unused contracts, as follows:

```

task deleteUnwantedContracts(type: Delete) {
    delete fileTree(dir: "${buildDir}/unpackedContracts",
        include: "**/*",
        excludes: [
            "**/${project.name}/**",
            "**/${first-topic}/**",
            "**/${second-topic}/**"])
}

```

6. Create task dependencies, as follows:

```

unzipContracts.dependsOn("getContracts")
deleteUnwantedContracts.dependsOn("unzipContracts")
build.dependsOn("deleteUnwantedContracts")

```

7. Configure the plugin by specifying the directory that contains the contracts, by setting the `contractsDslDir` property, as follows:

```

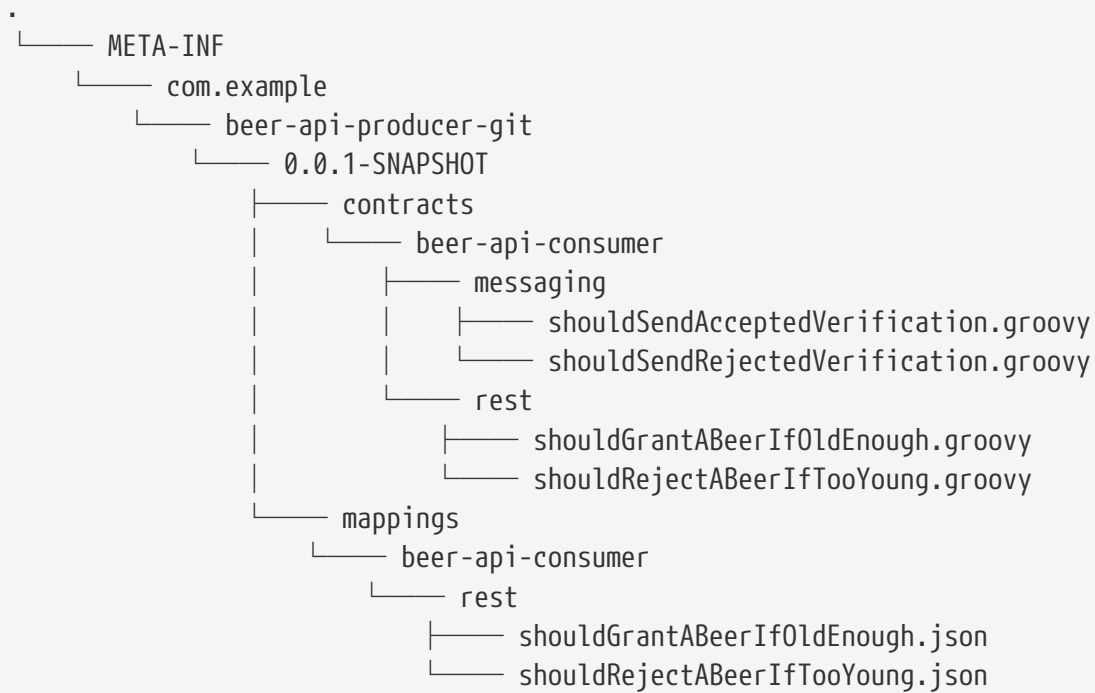
contracts {
    contractsDslDir = new File("${buildDir}/unpackedContracts")
}

```

8.6. How Can I Use Git as the Storage for Contracts and Stubs?

In the polyglot world, there are languages that do not use binary storage, as Artifactory and Nexus do. Starting from Spring Cloud Contract version 2.0.0, we provide mechanisms to store contracts and stubs in a SCM (Source Control Management) repository. Currently, the only supported SCM is Git.

The repository would have to have the following setup (which you can checkout from [here](#)):



Under the **META-INF** folder:

- We group applications by **groupId** (such as **com.example**).
- Each application is represented by its **artifactId** (for example, **beer-api-producer-git**).
- Next, each application is organized by its version (such as **0.0.1-SNAPSHOT**). Starting from Spring Cloud Contract version **2.1.0**, you can specify the versions as follows (assuming that your versions follow semantic versioning):
 - **+** or **latest**: To find the latest version of your stubs (assuming that the snapshots are always the latest artifact for a given revision number). That means:
 - If you have **1.0.0.RELEASE**, **2.0.0.BUILD-SNAPSHOT**, and **2.0.0.RELEASE**, we assume that the latest is **2.0.0.BUILD-SNAPSHOT**.
 - If you have **1.0.0.RELEASE** and **2.0.0.RELEASE**, we assume that the latest is **2.0.0.RELEASE**.
 - If you have a version called **latest** or **+**, we will pick that folder.
 - **release**: To find the latest release version of your stubs. That means:
 - If you have **1.0.0.RELEASE**, **2.0.0.BUILD-SNAPSHOT**, and **2.0.0.RELEASE** we assume that the latest is **2.0.0.RELEASE**.
 - If you have a version called **release**, we pick that folder.

Finally, there are two folders:

- **contracts**: The good practice is to store the contracts required by each consumer in the folder with the consumer name (such as **beer-api-consumer**). That way, you can use the **stubs-per-consumer** feature. Further directory structure is arbitrary.
- **mappings**: The Maven or Gradle Spring Cloud Contract plugins push the stub server mappings in

this folder. On the consumer side, Stub Runner scans this folder to start stub servers with stub definitions. The folder structure is a copy of the one created in the `contracts` subfolder.

8.6.1. Protocol Convention

To control the type and location of the source of contracts (whether binary storage or an SCM repository), you can use the protocol in the URL of the repository. Spring Cloud Contract iterates over registered protocol resolvers and tries to fetch the contracts (by using a plugin) or stubs (from Stub Runner).

For the SCM functionality, currently, we support the Git repository. To use it, in the property where the repository URL needs to be placed, you have to prefix the connection URL with `git://`. The following listing shows some examples:

```
git://file:///foo/bar
git://https://github.com/spring-cloud-samples/spring-cloud-contract-nodejs-
contracts-git.git
git://git@github.com:spring-cloud-samples/spring-cloud-contract-nodejs-
contracts-
git.git
```

8.6.2. Producer

For the producer, to use the SCM (Source Control Management) approach, we can reuse the same mechanism we use for external contracts. We route Spring Cloud Contract to use the SCM implementation from the URL that starts with the `git://` protocol.



You have to manually add the `pushStubsToScm` goal in Maven or use (bind) the `pushStubsToScm` task in Gradle. We do not push stubs to the `origin` of your git repository.

The following listing includes the relevant parts both Maven and Gradle build files:

Maven

```
<plugin>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-contract-maven-plugin</artifactId>
  <version>${spring-cloud-contract.version}</version>
  <extensions>>true</extensions>
  <configuration>
    <!-- Base class mappings etc. -->

    <!-- We want to pick contracts from a Git repository -->
    <contractsRepositoryUrl>git://https://github.com/spring-cloud-
samples/spring-cloud-contract-nodejs-contracts-git.git</contractsRepositoryUrl>

    <!-- We reuse the contract dependency section to set up the path
to the folder that contains the contract definitions. In our case the
path will be /groupId/artifactId/version/contracts -->
    <contractDependency>
      <groupId>${project.groupId}</groupId>
      <artifactId>${project.artifactId}</artifactId>
      <version>${project.version}</version>
    </contractDependency>

    <!-- The contracts mode can't be classpath -->
    <contractsMode>REMOTE</contractsMode>
  </configuration>
  <executions>
    <execution>
      <phase>package</phase>
      <goals>
        <!-- By default we will not push the stubs back to SCM,
you have to explicitly add it as a goal -->
        <goal>pushStubsToScm</goal>
      </goals>
    </execution>
  </executions>
</plugin>
```

Gradle

```
contracts {
    // We want to pick contracts from a Git repository
    contractDependency {
        stringNotation = "${project.group}:${project.name}:${project.version}"
    }
    /*
    We reuse the contract dependency section to set up the path
    to the folder that contains the contract definitions. In our case the
    path will be /groupId/artifactId/version/contracts
    */
    contractRepository {
        repositoryUrl = "git://https://github.com/spring-cloud-samples/spring-
cloud-contract-nodejs-contracts-git.git"
    }
    // The mode can't be classpath
    contractsMode = "REMOTE"
    // Base class mappings etc.
}

/*
In this scenario we want to publish stubs to SCM whenever
the `publish` task is invoked
*/
publish.dependsOn("publishStubsToScm")
```

You can also further customize the `publishStubsToScm` gradle task. In the following example, the task is customized to pick contracts from a local git repository:

gradle

```
publishStubsToScm {
    // We want to modify the default set up of the plugin when publish stubs to
    scm is called
    // We want to pick contracts from a Git repository
    contractDependency {
        stringNotation = "${project.group}:${project.name}:${project.version}"
    }
    /*
    We reuse the contract dependency section to set up the path
    to the folder that contains the contract definitions. In our case the
    path will be /groupId/artifactId/version/contracts
    */
    contractRepository {
        repositoryUrl = "git://file://${new File(project.rootDir,
"../target")}/contract_empty_git/"
    }
    // We set the contracts mode to `LOCAL`
    contractsMode = "LOCAL"
}
```

IMPORTANT

Starting with the **2.3.0.RELEASE**, the `customize{}` closure previously used for the `publishStubsToScm` customization is no longer available. The settings should be applied directly within the `publishStubsToScm` closure, as in the preceding example.

With such a setup:

- A git project is cloned to a temporary directory
- The SCM stub downloader goes to the `META-INF/groupId/artifactId/version/contracts` folder to find contracts. For example, for `com.example:foo:1.0.0`, the path would be `META-INF/com.example/foo/1.0.0/contracts`.
- Tests are generated from the contracts.
- Stubs are created from the contracts.
- Once the tests pass, the stubs are committed in the cloned repository.
- Finally, a push is sent to that repo's `origin`.

8.6.3. Producer with Contracts Stored Locally

Another option to use the SCM as the destination for stubs and contracts is to store the contracts locally, with the producer, and only push the contracts and the stubs to SCM. The following listing shows the setup required to achieve this with Maven and Gradle:

```

<plugin>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-contract-maven-plugin</artifactId>
  <version>${spring-cloud-contract.version}</version>
  <extensions>>true</extensions>
  <!-- In the default configuration, we want to use the contracts stored locally -->
  <configuration>
    <baseClassMappings>
      <baseClassMapping>
        <contractPackageRegex>.*messaging.*</contractPackageRegex>
        <baseClassFQN>com.example.BeerMessagingBase</baseClassFQN>
      </baseClassMapping>
      <baseClassMapping>
        <contractPackageRegex>.*rest.*</contractPackageRegex>
        <baseClassFQN>com.example.BeerRestBase</baseClassFQN>
      </baseClassMapping>
    </baseClassMappings>
    <basePackageForTests>com.example</basePackageForTests>
  </configuration>
  <executions>
    <execution>
      <phase>package</phase>
      <goals>
        <!-- By default we will not push the stubs back to SCM,
              you have to explicitly add it as a goal -->
        <goal>pushStubsToScm</goal>
      </goals>
      <configuration>
        <!-- We want to pick contracts from a Git repository -->
        <contractsRepositoryUrl>git://file://${env.ROOT}/target/contract_empty_git/
          </contractsRepositoryUrl>
        <!-- Example of URL via git protocol -->
        <!--<contractsRepositoryUrl>git://git@github.com:spring-cloud-
samples/spring-cloud-contract-samples.git</contractsRepositoryUrl>-->
        <!-- Example of URL via http protocol -->
        <!--<contractsRepositoryUrl>git://https://github.com/spring-cloud-
samples/spring-cloud-contract-samples.git</contractsRepositoryUrl>-->
        <!-- We reuse the contract dependency section to set up the path
              to the folder that contains the contract definitions. In our case
              the
              path will be /groupId/artifactId/version/contracts -->
        <contractDependency>
          <groupId>${project.groupId}</groupId>
          <artifactId>${project.artifactId}</artifactId>
          <version>${project.version}</version>
        </contractDependency>

```

```

        <!-- The mode can't be classpath -->
        <contractsMode>LOCAL</contractsMode>
    </configuration>
</execution>
</executions>
</plugin>

```

Gradle

```

contracts {
    contractsDslDir = file("src/test/resources/contracts")
    testFramework = "JUNIT5"
    // Base package for generated tests
    basePackageForTests = "com.example"
    baseClassMappings {
        baseClassMapping(".*messaging.*", "com.example.BeerMessagingBase")
        baseClassMapping(".*rest.*", "com.example.BeerRestBase")
    }

    /*
    In this scenario we want to publish stubs to SCM whenever
    the `publish` task is executed
    */
    publishStubsToScm {

        // We want to pick contracts from a Git repository
        contractDependency {
            stringNotation = "${project.group}:${project.name}:${project.version}"
        }

        contractRepository {
            /*
            We reuse the contract dependency section to set up the path
            to the folder that contains the contract definitions. In our case the
            path will be /groupId/artifactId/version/contracts
            */
            repositoryUrl = "git://file://${new File(project.rootDir,
            "../target")}/contract_empty_git/"
        }
    }
}

```

With such a setup:

- Contracts from the default `src/test/resources/contracts` directory are picked.
- Tests are generated from the contracts.
- Stubs are created from the contracts.
- Once the tests pass:

- The git project is cloned to a temporary directory.
- The stubs and contracts are committed in the cloned repository.
- Finally, a push is done to that repository's **origin**.

8.6.4. Keeping Contracts with the Producer and Stubs in an External Repository

You can also keep the contracts in the producer repository but keep the stubs in an external git repository. This is most useful when you want to use the base consumer-producer collaboration flow but cannot use an artifact repository to store the stubs.

To do so, use the usual producer setup and then add the `pushStubsToScm` goal and set `contractsRepositoryUrl` to the repository where you want to keep the stubs.

8.6.5. Consumer

On the consumer side, when passing the `repositoryRoot` parameter, either from the `@AutoConfigureStubRunner` annotation, the JUnit 4 rule, JUnit 5 extension, or properties, you can pass the URL of the SCM repository, prefixed with the `git://` protocol. The following example shows how to do so:

```
@AutoConfigureStubRunner(  
    stubsMode="REMOTE",  
    repositoryRoot="git://https://github.com/spring-cloud-samples/spring-cloud-  
contract-nodejs-contracts-git.git",  
    ids="com.example:bookstore:0.0.1.RELEASE"  
)
```

With such a setup:

- The git project is cloned to a temporary directory.
- The SCM stub downloader goes to the `META-INF/groupId/artifactId/version/` folder to find stub definitions and contracts. For example, for `com.example:foo:1.0.0`, the path would be `META-INF/com.example/foo/1.0.0/`.
- Stub servers are started and fed with mappings.
- Messaging definitions are read and used in the messaging tests.

8.7. How Can I Debug the Request/Response Being Sent by the Generated Tests Client?

The generated tests all boil down to RestAssured in some form or fashion. RestAssured relies on the [Apache HttpClient](#). HttpClient has a facility called [wire logging](#), which logs the entire request and response to HttpClient. Spring Boot has a logging [common application property](#) for doing this sort

of thing. To use it, add it to your application properties, as follows:

```
logging.level.org.apache.http.wire=DEBUG
```

8.8. How Can I Debug the Mapping, Request, or Response Being Sent by WireMock?

Starting from version `1.2.0`, we set WireMock logging to `info` and set the WireMock notifier to being verbose. Now you can exactly know what request was received by the WireMock server and which matching response definition was picked.

To turn off this feature, set WireMock logging to `ERROR`, as follows:

```
logging.level.com.github.tomakehurst.wiremock=ERROR
```

8.9. How Can I See What Got Registered in the HTTP Server Stub?

You can use the `mappingsOutputFolder` property on `@AutoConfigureStubRunner`, `StubRunnerRule`, or `StubRunnerExtension` to dump all mappings for each artifact ID. Also, the port at which the given stub server was started is attached.

8.10. How Can I Reference Text from File?

In version `1.2.0`, we added this ability. You can call a `file(...)` method in the DSL and provide a path relative to where the contract lies. If you use YAML, you can use the `bodyFromFile` property.

8.11. How Can I Generate Pact, YAML, or X files from Spring Cloud Contract Contracts?

Spring Cloud Contract comes with a `ToFileContractsTransformer` class that lets you dump contracts as files for the given `ContractConverter`. It contains a `static void main` method that lets you run the transformer as an executable. It takes the following arguments:

- argument 1 : `FQN`: Fully qualified name of the `ContractConverter` (for example, `PactContractConverter`). **REQUIRED**.
- argument 2 : `path`: Path where the dumped files should be stored. **OPTIONAL**—defaults to `target/converted-contracts`.
- argument 3 : `path`: Path where the contracts should be searched for. **OPTIONAL**—defaults to

`src/test/resources/contracts.`

After calling the transformer, the Spring Cloud Contract files are processed and, depending on the provided FQN of the `ContractTransformer`, the contracts are transformed to the required format and dumped to the provided folder.

The following example shows how to configure Pact integration for both Maven and Gradle:

Maven

```
<plugin>
  <groupId>org.codehaus.mojo</groupId>
  <artifactId>exec-maven-plugin</artifactId>
  <version>1.6.0</version>
  <executions>
    <execution>
      <id>convert-dsl-to-pact</id>
      <phase>process-test-classes</phase>
      <configuration>
        <classpathScope>test</classpathScope>
        <mainClass>
          org.springframework.cloud.contract.verifier.util.ToFileContractsTransformer
        </mainClass>
        <arguments>
          <argument>
            org.springframework.cloud.contract.verifier.spec.pact.PactContractConverter
          </argument>
          <argument>${project.basedir}/target/pacts</argument>
          <argument>
            ${project.basedir}/src/test/resources/contracts
          </argument>
        </arguments>
      </configuration>
      <goals>
        <goal>java</goal>
      </goals>
    </execution>
  </executions>
</plugin>
```

Gradle

```
task convertContracts(type: JavaExec) {
    main =
    "org.springframework.cloud.contract.verifier.util.ToFileContractsTransformer"
    classpath = sourceSets.test.compileClasspath

    args("org.springframework.cloud.contract.verifier.spec.pact.PactContractConverter"
    ,
        "${project.rootDir}/build/pacts",
        "${project.rootDir}/src/test/resources/contracts")
}

test.dependsOn("convertContracts")
```

8.12. How Can I Work with Transitive Dependencies?

The Spring Cloud Contract plugins add the tasks that create the stubs jar for you. One problem that arises is that, when reusing the stubs, you can mistakenly import all of that stub's dependencies. When building a Maven artifact, even though you have a couple of different jars, all of them share one `pom.xml` file, as the following listing shows:

```
|—— producer-0.0.1.BUILD-20160903.075506-1-stubs.jar
|—— producer-0.0.1.BUILD-20160903.075506-1-stubs.jar.sha1
|—— producer-0.0.1.BUILD-20160903.075655-2-stubs.jar
|—— producer-0.0.1.BUILD-20160903.075655-2-stubs.jar.sha1
|—— producer-0.0.1.BUILD-SNAPSHOT.jar
|—— producer-0.0.1.BUILD-SNAPSHOT.pom
|—— producer-0.0.1.BUILD-SNAPSHOT-stubs.jar
|—— ...
|—— ...
```

There are three possibilities of working with those dependencies so as not to have any issues with transitive dependencies:

- Mark all application dependencies as optional
- Create a separate `artifactId` for the stubs
- Exclude dependencies on the consumer side

8.12.1. How Can I Mark All Application Dependencies as Optional?

If, in the `producer` application, you mark all of your dependencies as optional, when you include the `producer` stubs in another application (or when that dependency gets downloaded by Stub Runner), then, since all of the dependencies are optional, they do not get downloaded.

8.12.2. How can I Create a Separate `artifactId` for the Stubs?

If you create a separate `artifactId`, you can set it up in whatever way you wish. For example, you might decide to have no dependencies at all.

8.12.3. How can I Exclude Dependencies on the Consumer Side?

As a consumer, if you add the stub dependency to your classpath, you can explicitly exclude the unwanted dependencies.

8.13. How Can I Generate Spring REST Docs Snippets from the Contracts?

When you want to include the requests and responses of your API by using Spring REST Docs, you only need to make some minor changes to your setup if you are using `MockMvc` and

RestAssuredMockMvc. To do so, include the following dependencies (if you have not already done so):

maven

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-contract-verifier</artifactId>
  <scope>test</scope>
</dependency>
<dependency>
  <groupId>org.springframework.restdocs</groupId>
  <artifactId>spring-restdocs-mockmvc</artifactId>
  <optional>true</optional>
</dependency>
```

gradle

```
testImplementation 'org.springframework.cloud:spring-cloud-starter-contract-
verifier'
testImplementation 'org.springframework.restdocs:spring-restdocs-mockmvc'
```

Next, you need to make some changes to your base class. The following examples use [WebApplicationContext](#) and the standalone option with RestAssured:

WebApplicationContext

```
package com.example.fraud;

import io.restassured.module.mockmvc.RestAssuredMockMvc;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.TestInfo;
import org.junit.jupiter.api.extension.ExtendWith;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.restdocs.RestDocumentationContextProvider;
import org.springframework.restdocs.RestDocumentationExtension;
import org.springframework.test.web.servlet.setup.MockMvcBuilders;
import org.springframework.web.context.WebApplicationContext;

import static
org.springframework.restdocs.mockmvc.MockMvcRestDocumentation.document;
import static
org.springframework.restdocs.mockmvc.MockMvcRestDocumentation.documentationConfigu
ration;

@ExtendWith(RestDocumentationExtension.class)
@SpringBootTest(classes = Application.class)
public abstract class FraudBaseWithWebAppSetup {

    @Autowired
    private WebApplicationContext context;

    @BeforeEach
    public void setup(TestInfo info, RestDocumentationContextProvider
restDocumentation) {

RestAssuredMockMvc.mockMvc(MockMvcBuilders.webAppContextSetup(this.context)
    .apply(documentationConfiguration(restDocumentation))
    .alwaysDo(document(
        getClass().getSimpleName() + "_" + info.getDisplayName()))
    .build());
    }

    protected void assertThatRejectionReasonIsNull(Object rejectionReason) {
        assert rejectionReason == null;
    }
}
```

Standalone

```
package com.example.fraud;

import io.restassured.module.mockmvc.RestAssuredMockMvc;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.TestInfo;
import org.junit.jupiter.api.extension.ExtendWith;

import org.springframework.restdocs.RestDocumentationContextProvider;
import org.springframework.restdocs.RestDocumentationExtension;
import org.springframework.test.web.servlet.setup.MockMvcBuilders;

import static
org.springframework.restdocs.mockmvc.MockMvcRestDocumentation.document;
import static
org.springframework.restdocs.mockmvc.MockMvcRestDocumentation.documentationConfigu
ration;

@ExtendWith(RestDocumentationExtension.class)
public abstract class FraudBaseWithStandaloneSetup {

    @BeforeEach
    public void setup(TestInfo info, RestDocumentationContextProvider
restDocumentation) {
        RestAssuredMockMvc.standaloneSetup(MockMvcBuilders
            .standaloneSetup(new FraudDetectionController())
            .apply(documentationConfiguration(restDocumentation))
            .alwaysDo(document(
                getClass().getSimpleName() + "_" +
info.getDisplayName())));
    }
}
```



You need not specify the output directory for the generated snippets (since version 1.2.0.RELEASE of Spring REST Docs).

8.14. How Can I Use Stubs from a Location

If you want to fetch contracts or stubs from a given location without cloning a repository or fetching a JAR, use the `stubs://` protocol when providing the repository root argument for Stub Runner or the Spring Cloud Contract plugin. You can read more about this in [this section](#) of the documentation.

8.15. How Can I Generate Stubs at Runtime

If you want to generate stubs at runtime for contracts, switch the `generateStubs` property in the `@AutoConfigureStubRunner` annotation, or call the `withGenerateStubs(true)` method on the JUnit Rule or Extension. You can read more about this in [this section](#) of the documentation.

8.16. How Can I Make The Build Pass if There Are No Contracts or Stubs

If you want Stub Runner not to fail if no stubs were found, switch the `generateStubs` property in the `@AutoConfigureStubRunner` annotation or call the `withFailOnNoStubs(false)` method on the JUnit Rule or Extension. You can read more about this in [this section](#) of the documentation.

If you want the plugins not to fail the build when no contracts were found, you can set the `failOnNoStubs` flag in Maven or call the `contractRepository { failOnNoStubs(false) }` closure in Gradle.

8.17. How Can I Mark that a Contract Is in Progress

If a contract is in progress, it means that the, on the producer side, tests are not generated, but the stub is generated. You can read more about this in [this section](#) of the documentation.

In a CI build, before going to production, you would like to ensure that no in-progress contracts are on the classpath, because they may lead to false positives. For this reason, by default, in the Spring Cloud Contract plugin, we set the value of `failOnInProgress` to `true`. If you want to allow such contracts when tests are to be generated, set the flag to `false`.

Appendix B: Common application properties

Various properties can be specified inside your `application.properties` file, inside your `application.yml` file, or as command line switches. This appendix provides a list of common Spring Cloud Contract properties and references to the underlying classes that consume them.



Property contributions can come from additional jar files on your classpath, so you should not consider this an exhaustive list. Also, you can define your own properties.

8.B.1. Default application properties

Name	Default	Description
<code>stubrunner.amqp.enabled</code>	<code>false</code>	Whether to enable support for Stub Runner and AMQP.
<code>stubrunner.amqp.mockConnection</code>	<code>true</code>	Whether to enable support for Stub Runner and AMQP mocked connection factory.

Name	Default	Description
stubrunner.classifier	stubs	The classifier to use by default in ivy co-ordinates for a stub.
stubrunner.cloud.consul.enabled	true	Whether to enable stubs registration in Consul.
stubrunner.cloud.delegate.enabled	true	Whether to enable DiscoveryClient's Stub Runner implementation.
stubrunner.cloud.enabled	true	Whether to enable Spring Cloud support for Stub Runner.
stubrunner.cloud.eureka.enabled	true	Whether to enable stubs registration in Eureka.
stubrunner.cloud.loadbalancer.enabled	true	Whether to enable Stub Runner's Spring Cloud Load Balancer integration.
stubrunner.cloud.stubbed.discovery.enabled	true	Whether Service Discovery should be stubbed for Stub Runner. If set to false, stubs will get registered in real service discovery.
stubrunner.cloud.zookeeper.enabled	true	Whether to enable stubs registration in Zookeeper.
stubrunner.consumer-name		You can override the default <code>spring.application.name</code> of this field by setting a value to this parameter.
stubrunner.delete-stubs-after-test	true	If set to <code>false</code> will NOT delete stubs from a temporary folder after running tests.
stubrunner.fail-on-no-stubs	true	When enabled, this flag will tell stub runner to throw an exception when no stubs / contracts were found.
stubrunner.generate-stubs	false	When enabled, this flag will tell stub runner to not load the generated stubs, but convert the found contracts at runtime to a stub format and run those stubs.
stubrunner.http-server-stub-configurer		Configuration for an HTTP server stub.

Name	Default	Description
stubrunner.ids	<code>[]</code>	The ids of the stubs to run in "ivy" notation ([groupId]:artifactId:[version]:[classifier]:[port]). {@code groupId}, {@code classifier}, {@code version} and {@code port} can be optional.
stubrunner.ids-to-service-ids		Mapping of Ivy notation based ids to serviceIds inside your application. Example "a:b" → "myService" "artifactId" → "myOtherService"
stubrunner.integration.enabled	<code>true</code>	Whether to enable Stub Runner integration with Spring Integration.
stubrunner.jms.enabled	<code>true</code>	Whether to enable Stub Runner integration with Spring JMS.
stubrunner.kafka.enabled	<code>true</code>	Whether to enable Stub Runner integration with Spring Kafka.
stubrunner.kafka.initializer.enabled	<code>true</code>	Whether to allow Stub Runner to take care of polling for messages instead of the KafkaStubMessages component. The latter should be used only on the producer side.
stubrunner.mappings-output-folder		Dumps the mappings of each HTTP server to the selected folder.
stubrunner.max-port	<code>15000</code>	Max value of a port for the automatically started WireMock server.
stubrunner.min-port	<code>10000</code>	Min value of a port for the automatically started WireMock server.
stubrunner.password		Repository password.
stubrunner.properties		Map of properties that can be passed to custom {@link org.springframework.cloud.contract.stubrunner.StubDownloaderBuilder}.
stubrunner.proxy-host		Repository proxy host.

Name	Default	Description
stubrunner.proxy-port		Repository proxy port.
stubrunner.server-id		
stubrunner.stream.enabled	true	Whether to enable Stub Runner integration with Spring Cloud Stream.
stubrunner.stubs-mode		Pick where the stubs should come from.
stubrunner.stubs-per-consumer	false	Should only stubs for this particular consumer get registered in HTTP server stub.
stubrunner.username		Repository username.
wiremock.placeholder.enabled	true	Flag to indicate that http URLs in generated wiremock stubs should be filtered to add or resolve a placeholder for a dynamic port.
wiremock.reset-mappings-after-each-test	false	
wiremock.rest-template-ssl-enabled	false	
wiremock.server.files	[]	
wiremock.server.https-port	-1	
wiremock.server.https-port-dynamic	false	
wiremock.server.port	8080	
wiremock.server.port-dynamic	false	
wiremock.server.stubs	[]	

8.B.2. Additional application properties



The following properties can be passed as a system property (for example, `stubrunner.properties.git.branch`), as an environment variable (for example, `STUBRUNNER_PROPERTIES_GIT_BRANCH`), or as a property inside stub runner's annotation or a JUnit Rule (JUnit 4) or Extension (JUnit 5). In the latter case, you can pass the `git.branch` property instead of `stubrunner.properties.git.branch`.

Table 4. Stubrunner Properties Options

Name	Default	Description
<code>stubrunner.properties.git.branch</code>		When using the SCM-based approach, you can customize the branch name to check out.
<code>stubrunner.properties.git.commit-message</code>	Updating project [\$project] with stubs	When using the SCM based approach, you can customize the commit message for created stubs. The <code>\$project</code> text is replaced with the project name.
<code>stubrunner.properties.git.no-of-attempts</code>	10	When using the-SCM based approach, you can customize the number of retries to push the stubs to Git.
<code>stubrunner.properties.git.username</code>		When using the SCM-based approach, you can pass the username to connect to the git repository.
<code>stubrunner.properties.git.password</code>		When using the SCM-based approach, you can pass the password to connect to the git repository.
<code>stubrunner.properties.git.wait-between-attempts</code>	1000	When using the SCM-based approach, you can customize waiting time in ms between attempts to push the stubs to git.
<code>stubrunner.properties.git.ensure-git-suffix</code>	true	When using the SCM based approach, you can prevent stubrunner from adding <code>.git</code> to the repository URL by setting this property to <code>false</code> . This adds compatibility with git repositories which do not support such URLs, for example Azure DevOps.
<code>stubrunner.properties.stubs.find-producer</code>	false	When using the <code>stubs</code> protocol, you can toggle this flag to search for contracts in the <code>group id / artifact id</code> instead of taking the stubs directly from the provided folder.