Workbook for Spring Cloud Contract

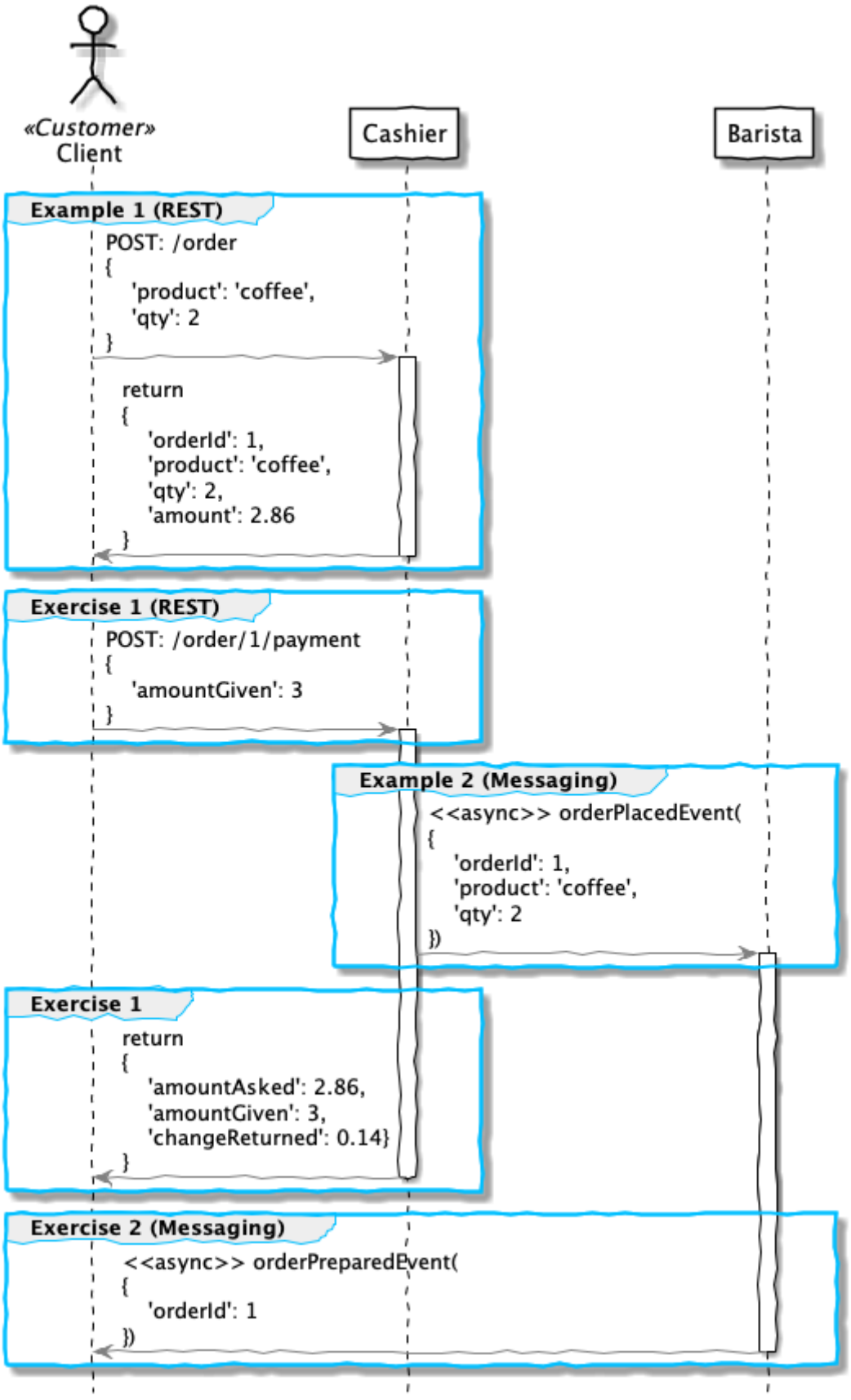
Introduction

The example handles ordering a coffee as probably everyone knows it from fastfood and coffeeshop chains. The goal is to provide a simple example which participants should be able to understand without further explanation or specialised knowledge. The focus is on the communication and the contracts describing this communication and not the business logic itself. Application design comes after simplicity for the sake of the example.

Get started

- check out the repo from GitHub: https://github.com/fabapp/spring-cloud-contract-workshop, `git clone https://github.com/fabapp/spring-cloud-contract-workshop`
- Import as project into your IDE
- switch to the branch `exercise-1`, e.g. by typing `git checkout exercise-1` on your console

A diagram is worth a thousand words



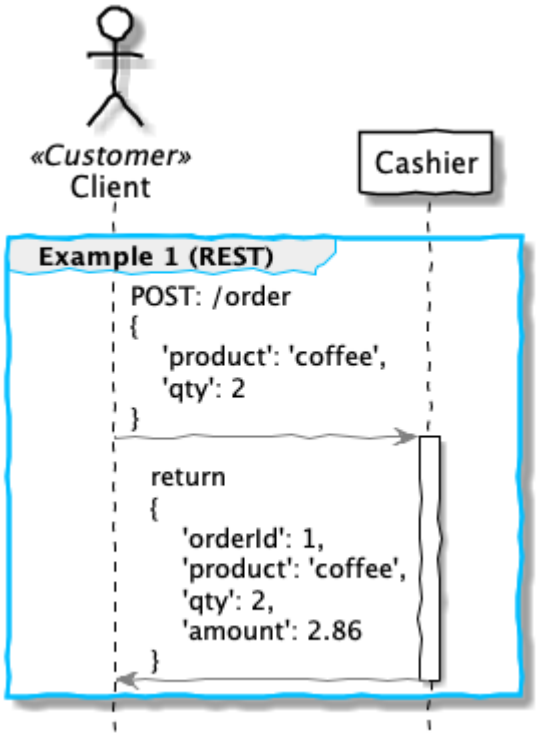
The Workflow

The workshop uses the most simple workflow where the contracts will be added by the consumer into the producer's project (module). The consumer then creates the stubs locally and uses these to implement the tests against the API provided by the producer. Then the producer implements the

contract.

...and we have consumer driven contracts

Example 1: Customer places an order (REST)



The `customer` consumes the `cashier`'s REST Api to order two coffee. The `cashier` takes the order and stores it in a database. She calculates the price and returns the order with all information.

The consumer defines a contract

- The `customer` (consumer) consumes the `cashier`'s (producer) REST API and provides a contract to describe the required REST Api of the `cashier`
- The contract is placed under `src/test/resources/contracts/⋯`
- The `cashier`s (producer) pom.xml requires dependencies to `spring-cloud-starter-contract-verifier` and the Spring Cloud Contract plugin `spring-cloud-contract-maven-plugin`
- When you run `mvn clean install -DskipTests` the plugin generates the stubs which will be used to provide a Wiremock server to the consumer which behaves as defined in the contract. The generated stub definition can be found in target/stubs/META-INF/de.fabiankrueger.scc/cashier/1.0-SNAPSHOT/mappings/rest/order/cashier-accepts-order.json of the `cashier` module.
- These stubs will be provided to the `customer` (consumer) through the generated jar `cashier-1.0-SNAPSHOT-stubs.jar` previously installed to the local Maven repository
- The consumer can then write tests and use `@AutoConfigureStubRunner(ids =`

`"<groupId>:<artifactId>:<version>:stubs", stubsMode = StubRunnerProperties.StubsMode.LOCAL)` annotation on class level of the tests to start the Wiremock server with the stubs as done in the CustomerPlacesOrderTest

- The `costumer` (consumer) requires only the `spring-cloud-starter-contract-stub-runner` dependency.

The producer implements the contract

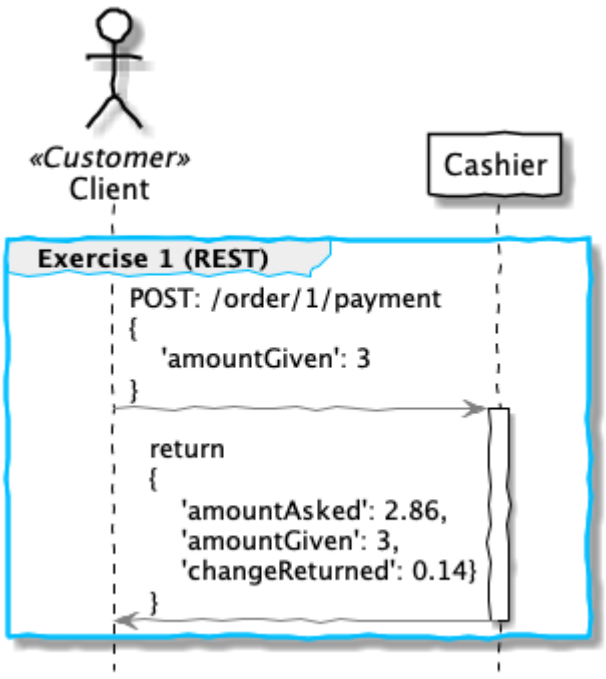
- The Spring Cloud Contract plugin generates a test for each contract which calls the producer Api to verify the contract.
- Therefor Spring Cloud Contract needs a BaseClass provided by you which provides the test setup to run the tests. The generated test will extend this BaseClass.
- The BaseClass has to be defined in the plugin configuration.
- If you run `mvn clean install` on the `cashier` (producer) side, the generated test will be executed and verify that the contract has been implemented.
- You can find the generated test in the `cashier`s target dir under `target/generated-test-sources`.

Exercise 1: Customer pays order (REST)

Now that we've seen how Spring Cloud Contract guarantees the implementation of the Api by the producer as expected by the consumer. Let's get our hands dirty and define a contract for the payment flow.

Setup

- switch to branch `exercise-1`
- Solution can be found in branch `exercise-2`



Consumer

The `customer` receveid the `Order` with an amount to pay. Now the `customer` needs to pay his order and sends a POST request with the amount given to the `cashier`. The `cashier` processes the payment and returns the information about the payment.

In short

The `customer` (consumer) wants to provide a contract that describes the required API provided by the `cashier` (producer). The contract should go here `cashier/src/test/resources/contracts/rest/payment/cashier-accepts-payment.groovy`. After providing the contract the stubs need to be generated to allow the `customer` to write tests against the API. Use `@AutoConfigureStubRunner` to implement a test on consumer side which uses the wiremock stub and verfies the usage of the API by the `customer` (consumer).

Step by step

1. Copy this contract

```
package contracts.rest.payment
org.springframework.cloud.contract.spec.Contract.make {
  description "should accept payment for order and return payment details."
  request {
      url "/order/1/payment"
      method POST()
      headers {
          contentType applicationJson()
      }
      body(amountGiven: 3)
  }
  response {
      status OK()
      headers {
          contentType applicationJson()
      }
      body (
          amountAsked: 2.86,
          amountGiven: 3,
          changeReturned: 0.14
      )
  }
}
```

to `cashier/src/test/resources/contracts/rest/payment/cashier-accepts-payment.groovy`

1. In the `cashier` module run `mvn clean install -DskipTests` to generate the stubs and install them in the local Maven repository.
2. In the `customer` module create a test `de.fabiankrueger.scc.customer.CustomerPaysOrderTest`

3. Annotate the test class with `@AutoConfigureStubRunner` annotation and set the required properties
4. Create a test method and use e.g. Spring's `RestTemplate` or (better [1]) `WebClient` to execute calls against the stubbed payment endpoint of the `cashier`
5. Verify the correct behaviour of the API using assertions
6. Run the test and verify that it passes

Producer

The `cashier` now needs to implement the Api defined by the contract.

In short

Create a BaseClass and configure the SCC Maven plugin in `pom.xml` to use this BaseClass for the generated payment API test. Use the `<baseClassMapping>` approach to do this. Activate the endpoint in the existing `CashierController` and verify that the generated tests succeed.

Step by step

1. In the `cashier` module create an abstract base class `de.fabiankrueger.scc.cashier.PaymentTestBase` in `src/test/java/`
2. Annotate the BaseClass with `@WebMvcTest(CashierController.class)` to initialize the Controller for integration test.
3. Annotate the BaseClass with `@AutoConfigureMockMvc` so Spring creates an instance of `MockMvc` for you.
4. Add a member of type `MockMvc` and add `@Autowired` to it to make Spring inject the configured `MockMvc` instance into the test.
5. Define a member of type `CashierService` and annotate it with `@MockBean` to make Spring inject a Mockito mock for the `CashierService`.
6. Create a `public void setup()` method and annotate it with `@BeforeEach`
7. In the setup method initialize RestAssured and pass the mockMvc instance to it `` `RestAssuredMockMvc.mockMvc(mockMvc)``. RestAssured will be used in the generated SCC test to call the payment endpoint.
8. Record the expected behaviour to the `cashierService` using Mockito's `when(..).thenReturn(..)` syntax
9. Configure a new `<baseClassMapping>` in the SCC plugin defintion in `pom.xml` that maps the new BaseClass to the contract.
10. Let SCC generate the tests by running `mvn clean install -DskipTests` and have a look at the generated test in the `cashier`s target dir.
11. If everything looks good run the generated tests for the `cashier`, e.g. by running `mvn clean test`

Resources

- StubRunner properties

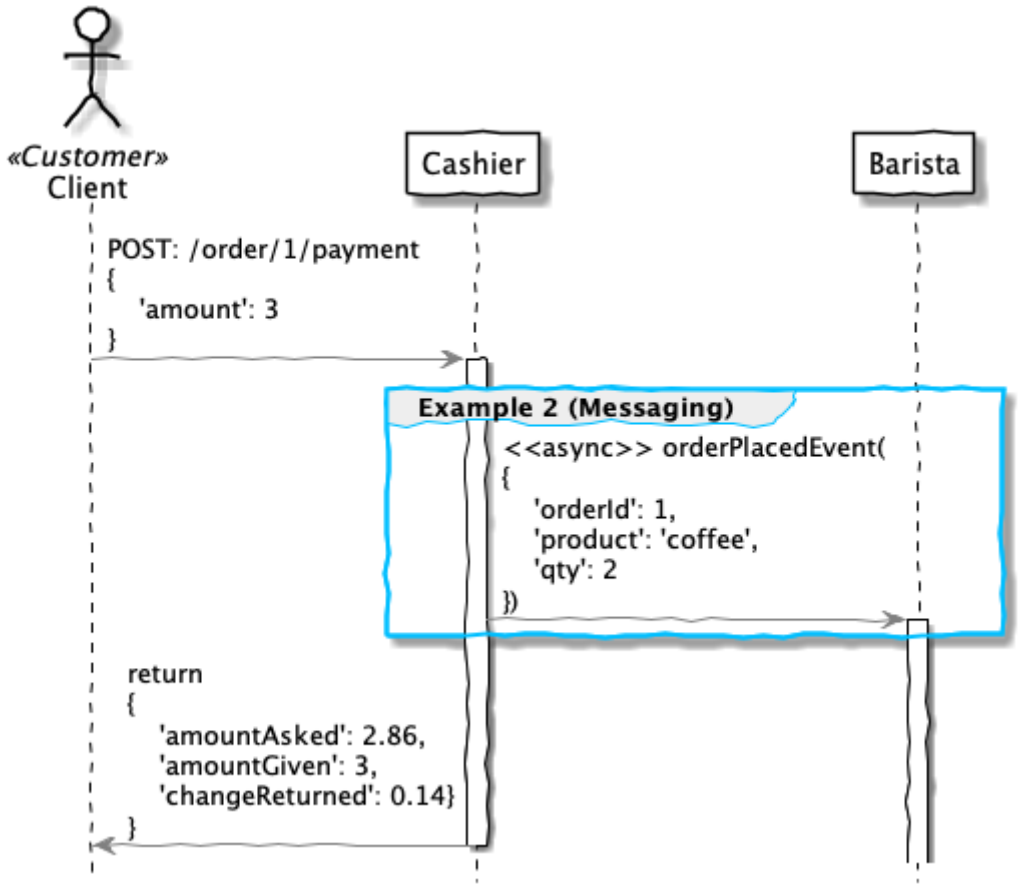
Example 2: Cashier places the Order (async messaging)

The `barista` has to be informed about new orders to prepare but the `cashier` should not wait for the order to be prepared until she can accept a new order. We can solve this situation by using asynchronuous communication using messaging.

Spring cloud Contract can use different messaging abstractions:

- Apache Camel
- Spring Integration
- Spring Cloud Stream
- Spring AMQP
- Spring JMS (requires embedded broker)
- Spring Kafka (requires embedded broker)

We use Spring Cloud Stream with Kafka in this example.



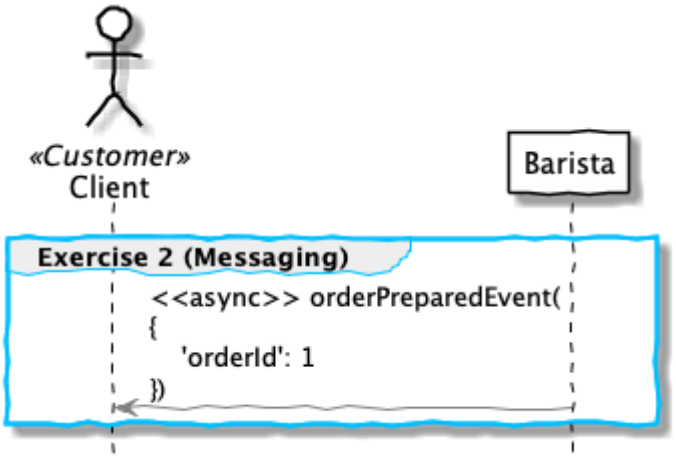
Consumer

- Again the consumer (`barista`) defines the required contract
- The contract describes the message and which label to use to trigger the message sending.
- After defining the contract we can generate the stubs and use them in the test on consumer side (`barista`).
- The sending of messages is done by a `StubFinder` provided by SCC and injected with `@Autowired` into the test.

Producer

- To trigger the sending of the message to Kafka (actually the mocked binder provided by Spring Cloud Streams) we define a method in the BaseClass for this test
- With SCC you need to annotate the BaseClass with `@AutoConfigureMessageVerifier` annotation
- We need no web endpoint, so we can disable the webEnvironment `@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.NONE)`
- The method uses the `OrderPlacedEventOutboundAdapter` to send a message to the mocked Binder
- The BaseClass needs to be mapped in the SCC plugin configuration in pom.xml

Exercise 2: Barista prepared the Order (async messaging)



Setup

You can checkout the branch `exercise-2` to start
You find the solution in branch `master`

Consumer

`customer` (consumer) wants to be informed if the order has been prepared. The `customer` listens for

`OrderPreparedEvent` messages on the Kafka topic `order-prepared`.

In short

After preparing the order the `barista` will publish the `OrderPreparedEvent` message on the topic `order-prepared`. Define a contract that verifies that a message with payload

```
{
  "orderId": 1
}
```

and header

```
"barista": "Jane Doe"
```

is published to the correct topic and provide the contract to `barista` (producer). Configure the Spring Cloud Contract plugin in the `barista`'s `pom.xml`. Then create the stubs and write a test for the `customer` against the created stub.

Step by step

- Create a contract `src/test/resources/contracts/order/prepared/publish-order-prepared-event.groovy` in the `barista` module

```
package contracts.order.prepared
org.springframework.cloud.contract.spec.Contract.make {
    description 'Barista publishes OrderPreparedEvent'
    label 'orderPreparedEvent'
    input {
        triggeredBy('publishOrderPreparedEvent()')
    }
    outputMessage {
        sentTo('orders-prepared')
        body('''{ "orderId" : "1" }''')
        headers {
            header('barista', 'Jane Doe')
        }
    }
}
```

- In the `barista` module run `mvn clean install -DskipTests` to generate the stubs and install them to your local maven repository
- Create a test `de.fabiankrueger.scc.customer.CustomerReceivesPreparedOrderTest` in the `src/test/java` dir of the `customer` module
- Annotate the test class with `@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.NONE)`

- Annotate the test class with `@AutoConfigureStubRunner(ids = "de.fabiankrueger.scc:barista:+:stubs", stubsMode = StubRunnerProperties.StubsMode.LOCAL)`
- Inject a mocked `CustomerService` bean by defining a member `customerService` of type `CustomerService` and annotate it with `@MockBean`
- Inject a `StubFinder` into the test using `@Autowired`. The `StubFinder` is provided by SCC and is used to trigger the stub to send a message as defined in the contract
- Create a test method which triggers the sending of an inbound `OrderPreparedEvent` as defined in the contract using `stubFinder.trigger("orderPreparedEvent")`
- Verify that the `customerService.onOrderPrepared(OrderPrepapredEvent, String)` method gets called when the message is received. Use Mockito's `ArgumentCaptor` to capture the parameters passed into the method
- Assert that the `orderId` of the `OrderPreparedEvent` passed into the `onOrderPrepared(…)` method matches the value defined in the contract
- Assert that the `barista` header passed into the `onOrderPrepared(…)` method matches the value defined in the contract
- Run the test and verify that it passes

Producer

When the `barista` prepared an order she should send an `OrderPreparedEvent` as defined in the contract to the `order-prepared` topic.

In short

Create a BaseClass and configure Spring Cloud Config to use this BaseClass for the producer tests of `barista`. The `BaristaService` should use the existing `OrderPreparedOutboundAdapter` to send a message that fulfills the given contract. Use Maven to generate and run the test to verify that the `barista` fulfills the contract.

Step by step

- Create an abstract base class `de.fabiankrueger.scc.barista.baseclasses.OrderPreparedBase` in `src/test/java` of the `barista` module.
- Configure the `spring-cloud-contract-maven-plugin` plugin in `barista` module
  - `de.fabiankrueger.scc.barista.baseclasses` should be used as package for base classes. Use `<packageWithBaseClasses>` to achieve this
  - The tests generated by SCC should have `de.fabiankrueger.scc.barista` as base package for tests. Use `<basePackageForTests>` to achieve this
- Annotate the base class with `@AutoConfigureMessageVerifier`
- Annotate the base class with `@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.NONE)`
- Inject existing `de.fabiankrueger.scc.barista.OrderPreparedOutboundAdapter` to the base class using `@Autowired`

- Create a method `publishOrderPreparedEvent` in the base class. This is the method defined in the contract which triggers the sending of the message
- The method should create a `OrderPreparedEvent` as defined in the contract and send it to Kafka using the `OrderPreparedOutboundAdapter.publish(…)` method

Exercise 3: Barista is a processor and not a source

Until now we triggered the sending of a `OrderProcessedEvent` in the `barista` module by directly calling the `publish` method of the `OrderPreparedOutboundAdapter`. If the `publish(..)` would be triggered by e.g. a scheduler and not as a result of an inbound message the `barista` would be a source for these events.
But the preparation of coffees is triggered by an inbound message and the result is sent as an outbound message. This makes the Barista a processor (output message triggered by input message). SCC allows to reflect this in a contract, see the documentation.

Producer

Create a new contract that reflects the `barista`'s nature of a processor by defining an inbound message that triggers the publication of an outbound message. Take a look at the generated test to understand the difference between testing a source and a processor.

Consumer

Write a new test (you can use the existing test class) in `consumer` and use the new contract to trigger sending a message to the `order-prepared` topic. Alternatively just change the label that triggers sending the message to the `order-prepared` topic. Alternatively just change the label that triggers sending the message in the existing test. .

Exercise 4: Use SCC to test the customer as message consumer

See documentation about Messaging with no output message and use SCC to test the `consumer` consuming `OrderPreparedEvent`s

Resources

- Spring Cloud Contract project
- Reference Documentation
- Maven configuration
- Spring Cloud Contract Tutorial on GitHub
- Use StubFinder to trigger sending of messages

- Contract DSL Reference
- Contract DSL YML Schema
- Hands-On Guide to Spring Cloud Contract on O'Reilly (Video)

[1] RestTemplate is in maintenance mode