Workbook for Spring Cloud Contract

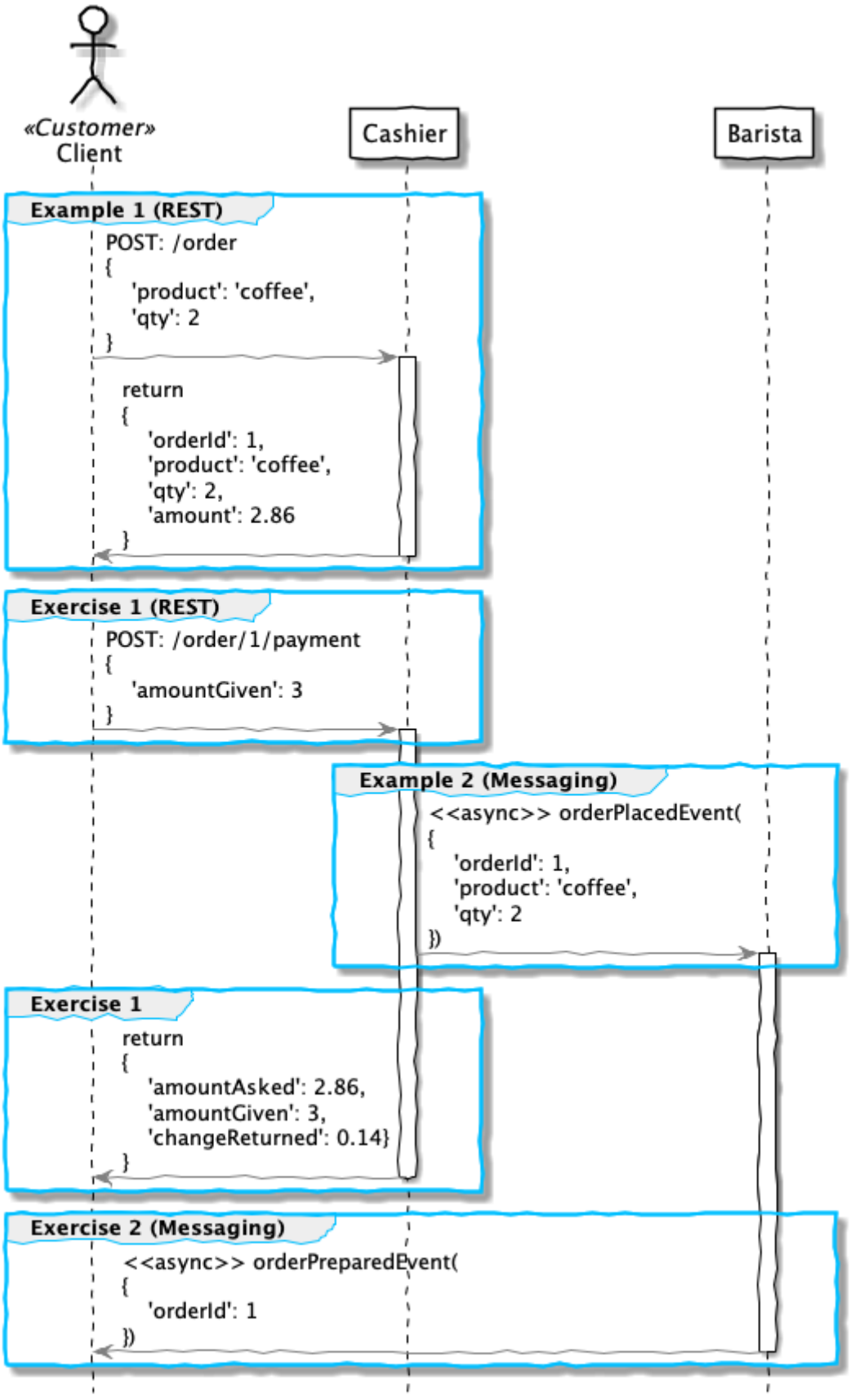
Introduction

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



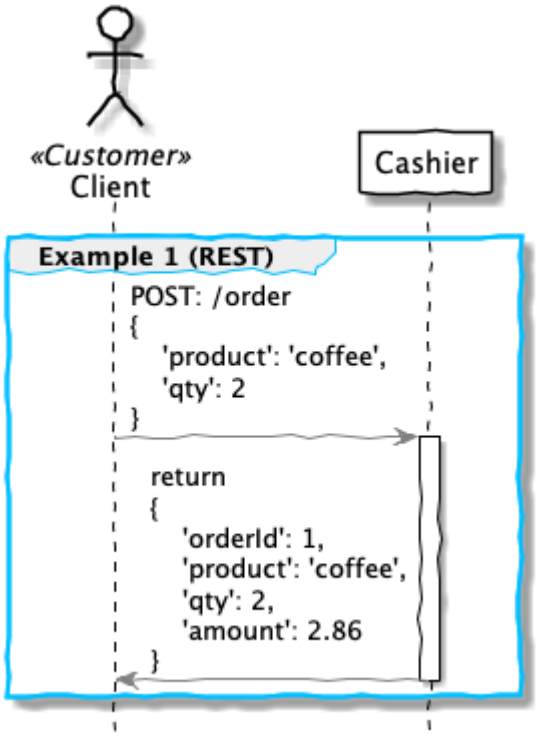
The Workflow

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

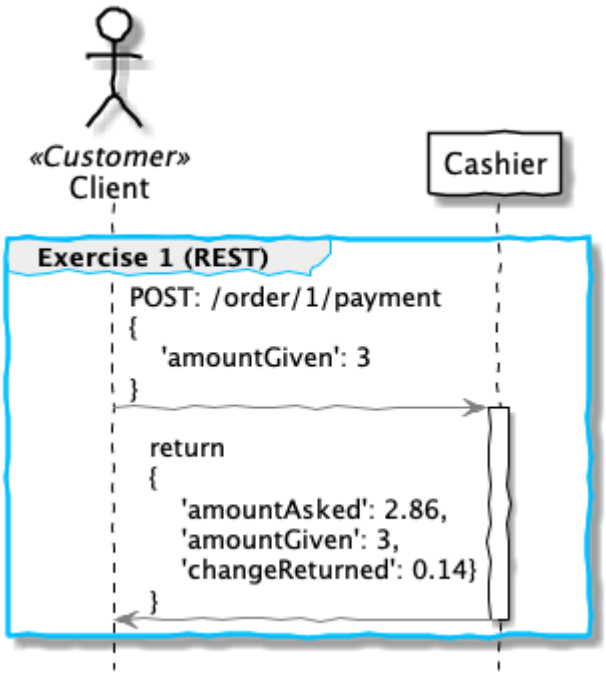
- The Spring Cloud Contract plugin generates a test for each contract which calls the producer Api to verify the contract.
- Therefor Spring Cloud Contract needs a BaseClass provided by you which provides the test setup to run the tests. The generated test will extend this BaseClass.
- The BaseClass 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 code can be found in branch `exercise-2`
- Step by step description can be found here



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.

- The `customer` (consumer) wants to provide a contract that describes the required API provided by the `cashier` (producer).
- Create the contract which verifies the correct path, request and response.
- 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.
- Run 'mvn clean install -DskipTests' to generate the stubs
- Create a new test on consumer side annotated with `@AutoConfigureStubRunner` which uses the wiremock stub and verfies the usage of the API by the `customer` (consumer).
- Verify that the test succeeds and the stibs work as expected

Producer

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

- In the `cashier` module create an abstract base class `de.fabiankrueger.scc.cashier.PaymentTestBase` in `src/test/java/`
- Annotate the base class with `@WebMvcTest(CashierController.class)` to initialize the Controller for integration test.
- Inject a MockMvc instance into the test. You can get it with

```
@Autowired
MockMvc mockMvc;
```

- 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.
- Record the expected behaviour to the `cashierService` using Mockito's `when(..).thenReturn(..)` syntax
- 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.
- If everything looks good run the generated tests for the `cashier`, e.g. by running `mvn clean test`
- Have a look at the generated tests and stubs and verify that you understand what happened.

Resources

- StubRunner properties

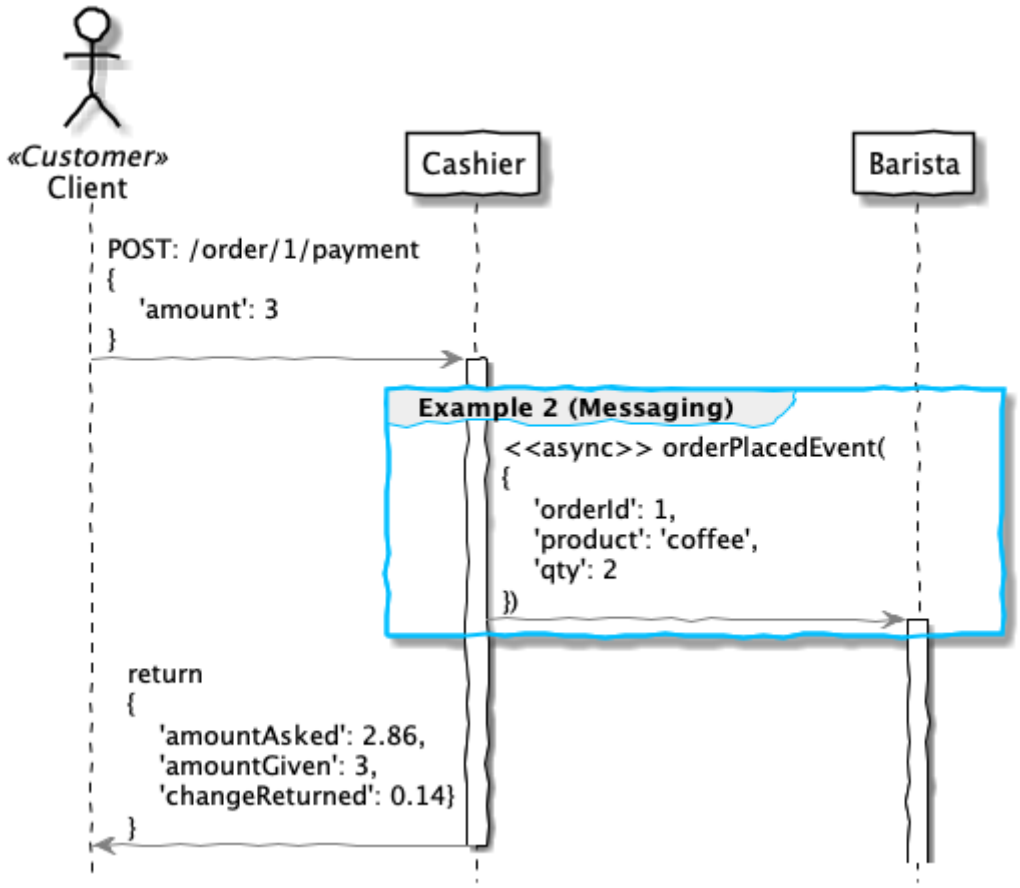
Example 2: Cashier places the Order (async messaging)

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.



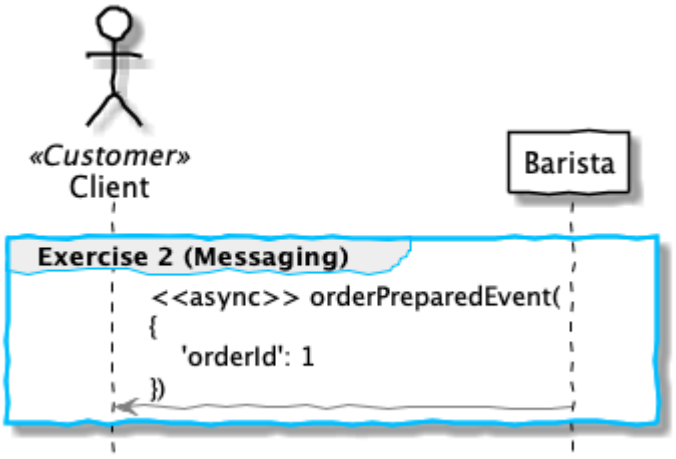
Consumer

- 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`
Step by step description can be found here

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.

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.

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. .

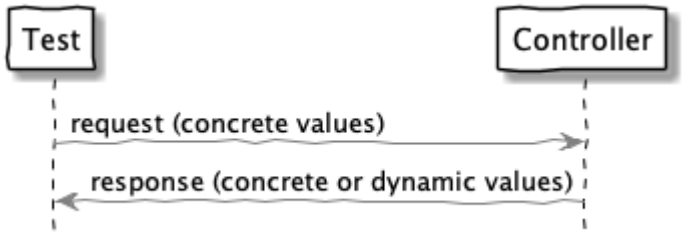
Exercise 4: Use SCC to test the customer as message consumer

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

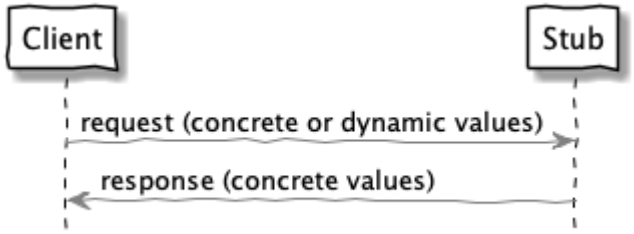
Dynamic properties

There can be situations where a request or response contains values that are dynamic, e.g. a timestamp or a UUID. Spring Cloud Contract can handle these situations.

On **Producer side** values of the **request** need to be **concrete** but values of the **response** can be **dynamic**!



On **Consumer side** values of the **request** can be **dynamic** but values of the **response** need to be **concrete**!



NOTE	You can use `consumer(..)` and `producer(..)` or `c(..)` and `p(..)` or `client(..)` and `server(..)` or `stub(..)` and `test(..)` to configure these settings.

```
...
request {
  body(
    mobileNumber: $(
      // This is the regex for mobileNumber accepted by the stub
      consumer(regex("\\+49 ([1-9]{1}[0-9]{2}) ([0-9]{7})")),
      // This number is sent from the producer to the Controller in generated tests
      producer("+49 160 5563477")
    )
  )
}
...
```

You can do the same for values of the response, but this time the rules are inverted.

```
...
response {
  ...
  body (
    amount: $(
      // The stub will always return 2.86 as price
      consumer(2.86),
      // But the test for the controller will accept any double as amount
      producer(anyDouble())
    )
  )
}
...
```

There can also be situations when a response should contain properties from a given request. This can be done using `fromRequest()` combined with jsonPath expression.

```
...
response {
  ...
  body (
    // Now the returned qty will be the value from the request body
    qty: fromRequest().body('$.qty')
  )
}
...
```

It is also possible to call methods in the base class to generate values for the request or assert values in the response.

```
...
response {
  ...
request {
  url $(
    ...
    server(execute('generateUrl()'))
  )
}

response {
  ...
  body (
    someValue: $(execute('assertSomeValueIsCorrect($it)')),
  )
}
...
```

NOTE | See Dynamic properties

Exercise 5.1: timeOrdered

- The creation time of an `order` should be stored.
  Use a member `Instant timeOrdered` in `Order` to keep the information.
  It can be created by JPA when persisting the `Order` using `@CreationTimestamp`.
  Prevent the value to be read from incoming requests using `@JsonProperty(access = JsonProperty.Access.READ_ONLY)`.
- The test for the `producer` should verify that `cashier`'s `order` API returns a response property `timeOrdered` with a format of ISO 8601.
- The value returned by stubs should always be `2016-12-31T23:30:59Z`.
- Have a look at the generated and test and stubs and how these changes are reflected.
- Adjust client tests accordingly

Hint: To mock the "real" behaviour you will need to add the `Instant` to the object passed to the `CashierService` mock. This is how it can be done with Mockito:

```
when(cashierService.processOrder(any(Order.class))).thenAnswer(o -> {
  Order processedOrder = (Order) o.getArgument(0);
  ...
})
```

Exercise 5.2: mobileNumber

- The `cashier`'s order api should expect a `mobileNumber` to be given.
- The mobile number should look like this: `+49 177 1234567` according to this regex `"\\+49 ([1-9]{1}[0-9]{2}) ([0-9]{7})"`.
- Any number following this pattern should be accepted by the generated stubs.
- The generated test should send the number `"+49 160 5563477"` to the producer.
- The given number should be returned in the response as given in the request
- Have a look at the generated and test and stubs and how these changes are reflected.
- Adjust client tests accordingly

Exercise 5.3: id in URL should be dynamic

- The `id` part in the `/order/{id}/payment` URL should be dynamic
- The contract should accept any id in the generated stubs
- The contract should use a method `generateUrl()` to generate the URLs called from generated producer test.
  You can use `Math.abs(new Random().nextInt())` to generate ids.

- Have a look at the generated and test and stubs and how these changes are reflected.
- Adjust client tests accordingly

Hint: You'll need to modify the Mockito mock to accept any Long as order id:

```
  when(cashierService.processPayment(anyLong(), eq(amountGiven))).thenReturn
(processedPayment);
```

NOTE See: Executing Custom Methods on the Server Side

Contracts in Git repository

The consumers provided the contracts to the producer application by checking it out and providing a pull request (in theory). This requires consumers to be able to access to the code of the producer and to build it locally. Additionally this works only well when we aren't in a polyglot environment. The devs of a JavaScript client might have a harder time to follow this workflow which requires a JVM and a Maven or gradle build.

Spring contract allows to provide the contracts and stubs from Git, Repository Manager like Nexus or Artifactory or a file system. When working in polyglot environments it makes things easier if contracts and stubs can be provided in Git.

Let's do it…

Exercise 6.1: Provide contracts for cashiers through git

- Create a git repository on your local machine (could be anywhere though)
- Configure the scc plugin in the `cashier`'s pom.xml to retrieve contracts from this repository
  Keep the `/` in the end when you use a local repo!

```
...
 <contractsMode>REMOTE</contractsMode>
  <contractDependency>
      <groupId>${project.groupId}</groupId>
      <artifactId>${project.artifactId}</artifactId>
      <version>${project.version}</version>
  </contractDependency>
  <contractsRepositoryUrl>git://file:///path-to-your-(local)-
repo/</contractsRepositoryUrl>
</configuration>
<executions>
  <execution>
      <phase>package</phase>
      <goals>
          <goal>pushStubsToScm</goal>
      </goals>
  </execution>
</executions>
...
```

- checkout the repository to another dir (representing your local repo)
- Move(!) everything from `cashier`s `src/test/resources/contracts` to the local repo.
- commit the existing contracts to the repo using this structure:

```
.
└── META-INF
    └── de.fabiankrueger.scc
        └── cashier
            └── 1.0-SNAPSHOT
                └── contracts
                    ├── messaging
                    │   └── order
                    │       └── cashier-publishes-order-processed-event.groovy
                    └── rest
                        ├── order
                        │   └── cashier-accepts-order.groovy
                        └── payment
                            └── cashier-accepts-payment.groovy
```

- Run a build for `cashier`, e.g. `mvn clean package`
- Verify the console output, verify that scc generated and committed the stubs created from the contracts
- Pull the latest changes into your local repo and verify that the stubs committed by scc are there.

Exercise 6.2: Make the consumer(s) retrive the stubs from git

The `customer` and `barista` still pulls the stubs from the local Maven repository. Let's modify the `customer`'s integration test to pull the stubs from your git repo now.

- Adjust the `@AutoConfigureStubRunner` annotation on all tests related to communication with the `cashier` module.
- set the `ids` to the concrete dependency including the version.
- set the `stubsMode` to `StubRunnerProperties.StubsMode.REMOTE`
- and finally provide the `repositoryRoot`
- verify that the tests are still green

Exercise 7.1:

You can use a basic spring boot application annotated with `@EnableStubRunnerServer` to act as server for your stubs.

For sake of speed, we can use a version provided by scc

- Download spring-cloud-contract-stubrunner-boot to a directory of your choice `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'`
- Open the file `scripts/stub-runner-server.sh`
- adjust the path to the downloaded jar
- run the script or command
- Use the file `scripts/order-coffee.http` to send requests against the stubrunner server.

Exercise 7.2 Using @EnableStubRunnerServer

NOTE Stub Runner Boot

- Create a new Module or new project `stub-runner-server` which contains a Spring Boot starter application (you can use IntelliJ or start.spring.io to do so).
- The only required dependency is the following (it is important that the scope is `compile` (which is the default)

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

- Now annotate the class annotated with `@SpringBootApplication` with `@EnableStubRunnerServer`
- When starting the application we need to provide some settings so it knows where to find the stubs to provide
  - `--stubrunner.stubsMode="LOCAL"`
  - `--stubrunner.ids="de.fabiankrueger.scc:cashier:1.0-SNAPSHOT:stubs:9876"`
  - You can provide these as system properties, application.properties, in your IDE or as startup parameters when calling the jar.
- Start the application (there's a script in `scripts/stub-runner-server.sh`:

```
java -jar <app.jar> \
--stubrunner.stubsMode="LOCAL" \
--stubrunner.ids="de.fabiankrueger.scc:cashier:1.0-SNAPSHOT:stubs:8083"`
```

- call the provided stub:
- You can use `scripts/order-coffee.http` for this or use the integration test(s)

More features

- loading resquest/response from file
- Specifying the HTTP Request
- Specifying the HTTP Response
- Stateful contracts

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)

Ideas

- Cashier listens for `OrderPreparedEvent` and updates the state of the order
- Cashier provides a new REST endpoint to get and query orders
- The contracts for the new endpoint should be provided to a non-java client and thus should be kept in a dedicated Git repository.
- Connect stub runner with real message broker
- NodeJS as consumer
- NodeJS as producer