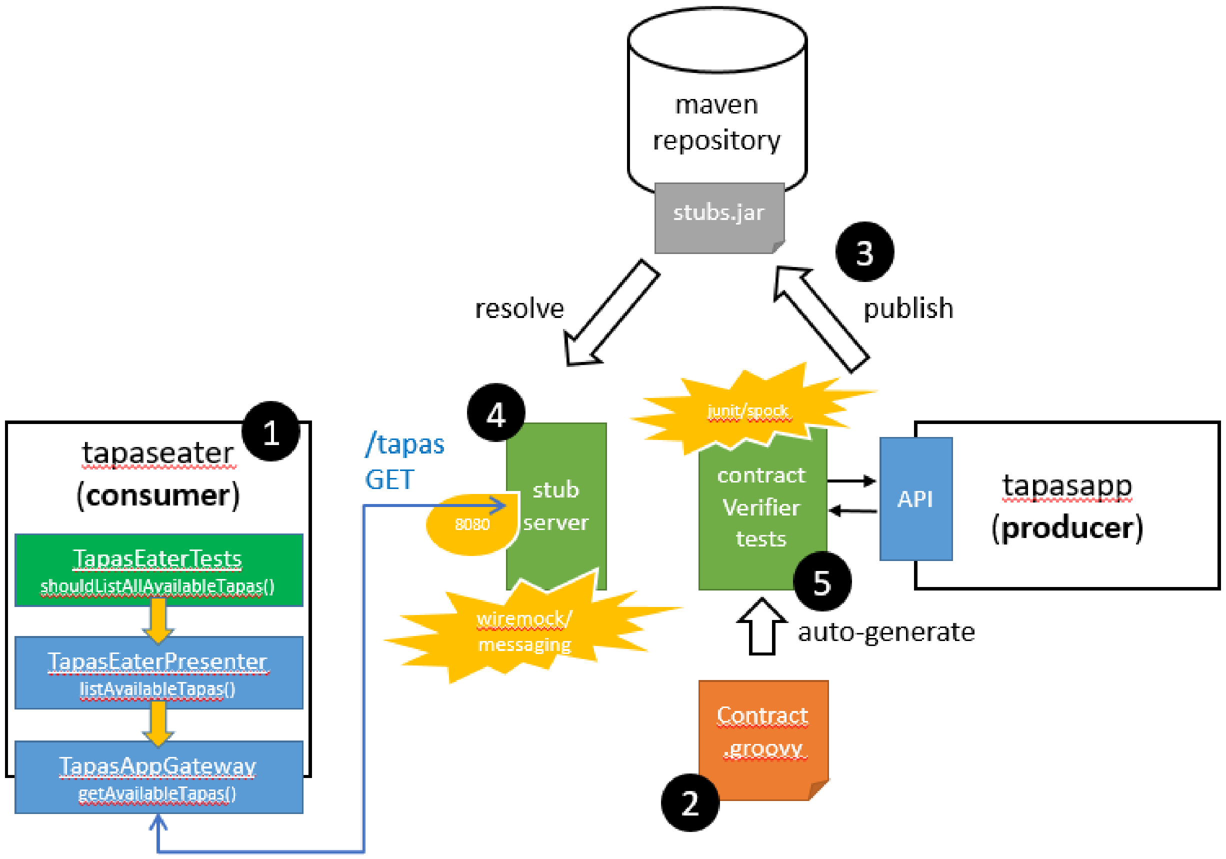
Spring-cloud-contract-exercise

We will be testing new functionality for retrieving a list of all available tapas calling a REST endpoint /tapas ; GET on the tapasapp service (producer).



CONSUMER

Step1: write consumer test

- We use a TDD approach on the consumer side (tapaseater service), writing the functional test for the new feature (retrieving the list of all available tapas) before implementing it.

Step2: write gateway implementation

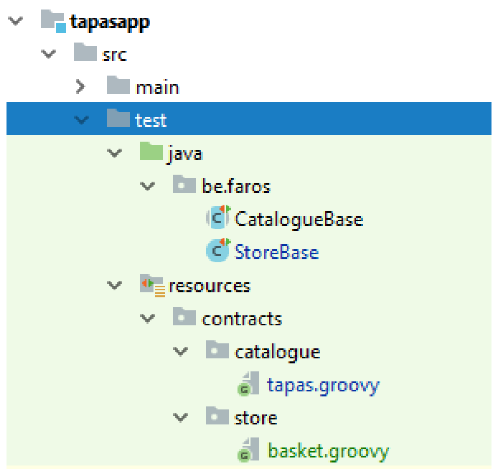
- At this moment we would write the gateway implementation (already provided for you) by sending a REST (GET) request to /tapas endpoint using Spring's RestTemplate.
- Run the previous test: it obviously fails because we have no endpoint on the producer side yet.

```
org.springframework.web.client.ResourceAccessException: I/O error on GET
request for "http://localhost:8080/tapas ": Connection refused.
```

Step3.a: create a new contract

The producer (tapasapp service) owns the contract(s), so physically, all the contract are in the producer's repository. But it's the consumer who writes the contract based on his needs. These contracts defines how the consumer expects the producer to behave.

- Switch to the producer side (tapasapp service) for writing the contract for this new feature
  - In this simplified exercise both services (consumer & producer) are in the same GIT repo but in a real environment this would mean checking out the producer's GIT repository.
  - We already created a file 'tapas.groovy' for you. In this file we'll define the contract using the Spring Cloud Contract Groovy DSL.
    - The file should be located in the src/test/resources/contracts/ folder for the spring-cloud-contract-plugin to find it (default behavior, look at tapasapp>pom.xml - 'contractsDirectory' if you need to change this)



Step3.b: write the contract

Spring Cloud Contract supports out of the box 2 types of DSL:

- Groovy DSL (used in this exercise)
- YAML

The contract consists of 2 elements:

- **Request**
  - Defined by a GET request for the endpoint /tapas
  - To the URL: /tapas

```
method GET()
url "/tapas"
```

- **Response**
  - The response answers with a status code of 200

```
status 200
```

  - Headers
    - Content-Type for both consumer & producer = 'application/json'

```
headers {
       contentType(applicationJson())
}
```

  - Body
    - Return 2 tapas elements defined by an id, name & price

```
body (
    [
        [
                id: 0,
                name: "All i oli",
                price: 1.5
        ],
        [
                id: 1,
                name: "Banderillas",
                price: 3
        ]
     ]
)
```

Step4: configure the Spring Cloud Contract Verifier Maven plugin

- The Spring Cloud Contract Verifier plugin uses the contract to:
  - Generate **tests** for testing the producers API
  - Produces and installs the **stub**(s) used by the consumer

- At this time we do not (yet) want to generate (producer) tests. As the consumer, we are mainly concerned about getting the generated stubs. Skip the test generation and execution but have the plugin generate the stubs for us:
  - tapasapp>mvn clean install -DskipTests
  - In the logs, you see something like this:
    - [INFO] Installing /some/path/tapasapp/target/tapasapp-0.0.1-SNAPSHOT-stubs.jar to /path/to/your/.m2/repository/be/contracts/be/faros/experimental/tapasapp/0.0.1-SNAPSHOT/tapasapp-0.0.1-SNAPSHOT-stubs.jar
  - Check your (maven) target folder and find the generated stub: tapasapp-0.0.1-SNAPSHOT-stubs.jar

Step5: Spring Cloud Contract Stub Runner

- Now back on the consumer service (tapaseater service) we add the stub runner dependency for automatic stub downloading from your maven repository (see next step).

Step6: Annotate your test class with @AutoConfigureStubRunner

- Put the @AutoConfigureStubRunner annotation on your test class and configure it:
  - ids = {"be.contracts.be.faros.experimental:tapasapp:+:stubs:8080"}
    - This references the generated stub in our local maven repo telling the stub runner to download the latest version (+) and expose the stub on port 8080
  - stubsMode = StubRunnerProperties.StubsMode.LOCAL
    - Pick stubs from a local storage (e.g. .m2)
    - Other options
      - CLASSPATH (default value) - pick stubs from the classpath
      - REMOTE - pick stubs from a remote location

Step7: Run the integration tests

- Now everything is in place to run our test on consumer side against the auto generated stubs (based on the contract).
- Run the test. In the logs you should see:
  INFO o.s.c.contract.stubrunner.StubServer: Started stub server for project [be.contracts.be.faros.experimental:tapasapp:0.0.1-SNAPSHOT:stubs] on port 8080
- The test should pass, Yippee!

PRODUCER

Now back to the producer (tapasapp service) where we want to make sure that our stub actually resembles the concrete API implementation. Spring Cloud Contract will generate tests on server-side based on the same contract that we can run making sure our implementation is compatible with the contract.

Step8: create base class

Before we can generate the tests we need to add a base test class. This base class is extended by all the auto-generated tests, and should contain all the setup necessary to run them (for example RestAssuredMockMvc controller setup or messaging test setup).

- 2 ways
  - Single Base Class for All Tests
  - Different Base Classes for Contracts (used in this exercise)

- We already created a CatalogueBase class to get you started
  - It uses SpringBoot testing support for creating a spring application context `@SpringBootTest(classes=TapasApp.class)`
  - Using `@MockBean` we mock the CatalogueSearching
- Use Mockito to return the (provided) list of tapas when calling catalogueSearching.getAllTapas()

Step9: run generated tests

- Run the (generated) API test on producer side (tapasapp service)
  - tapasapp>mvn test
- You should see the generated test(s) being run and pass
  - [INFO] Running be.faros.testing.tapasapp.StoreTest
  - [INFO] Running be.faros.testing.tapasapp.CatalogueTest
  - [INFO] Tests run: 2, Failures: 0, Errors: 0, Skipped: 0