Contract Tests with Spring Cloud Contract

Communication Stability

- can be done via shared model (problems: lack of autonomy, read all data - even not required)
- weakly-typed serialization
- read only what is needed, own model, autonomy
- problem with communication stability (the model is different)

Contract Tests

- how to verify communication stability in terms of content
- possible solutions:
  - e2e tests (problems: slow, late feedback, lot of resources, verifies not only communication stability)
  - mocks (problems: not always reflect the real instance of microservice)
- contract tests to the rescue
- verification of communication stability should not depend on the business logic

Contract Tests

- tests are fast, it is not needed to start other services with their whole setup (database, queues, message brokers etc.)
- consumer tests are based on stubs which are provided by the producer
- producer which provides those stubs must also fulfill requirements regarding the structure of the response
- stubs are automatically updated with the newest changes from the producer so consumer can detect problems with communication very fast

Contract

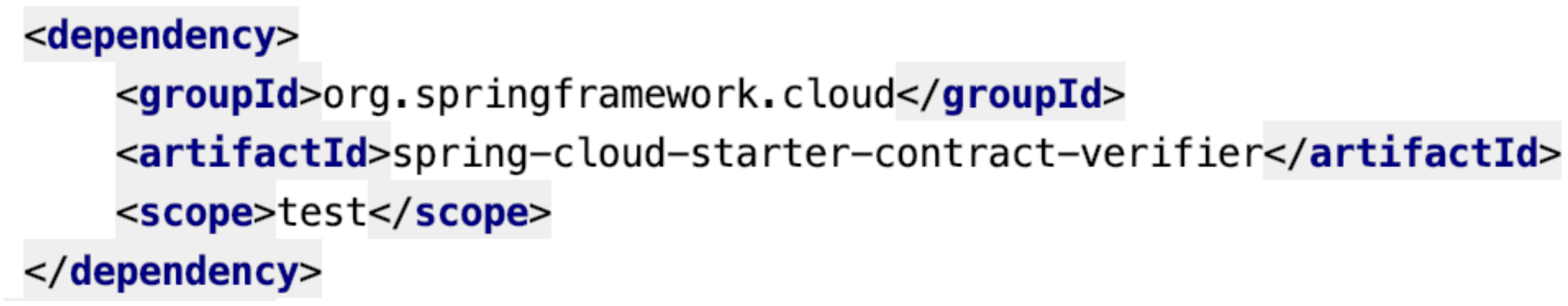
- is an agreement between the producer (API provider) and the consumer
- it is an in-between element
- it verifies both sides (producer and consumer)
- it defines the structure of the communication message (HTTP or queue)
- producer knows what must be send and consumer knows how to read it
- if the producer matches the contract and consumer matches the contract then the producer is compatible with the consumer

Contract

```
Contract.make {
    request {
        method 'GET'
        url '/documents/123456789'
        headers {
            contentType('application/json')
        }
    }
    response {
        status OK()
        body([
               id  : 123456789,
               status: "VALID"
        ])
        headers {
            contentType('application/json')
        }
    }
}
```

Producer Setup

- allows to write contracts
- used for test generation



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

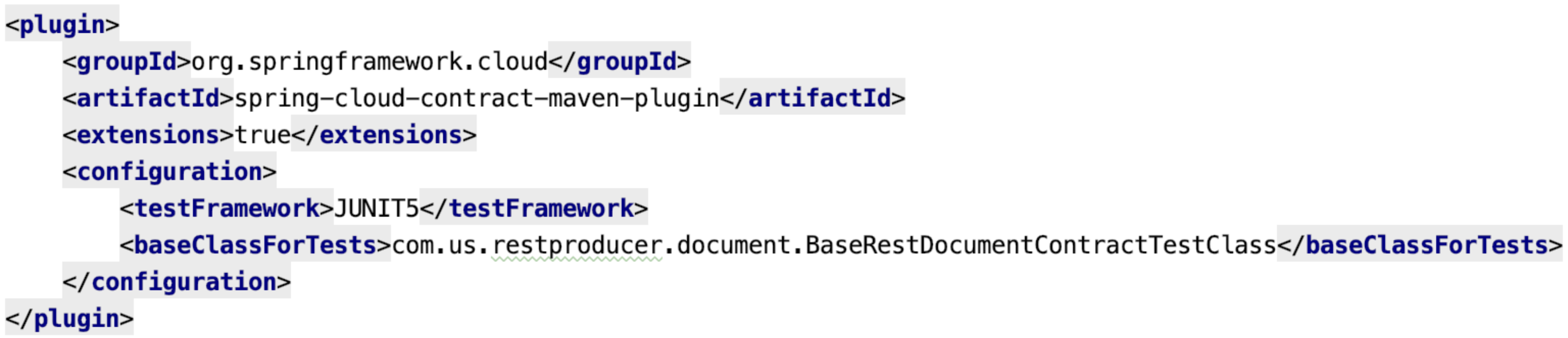
Producer Setup

- generates tests based on contract
- generates stubs based on contract

```xml
<plugin>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-contract-maven-plugin</artifactId>
    <extensions>true</extensions>
</plugin>
```

Producer Setup

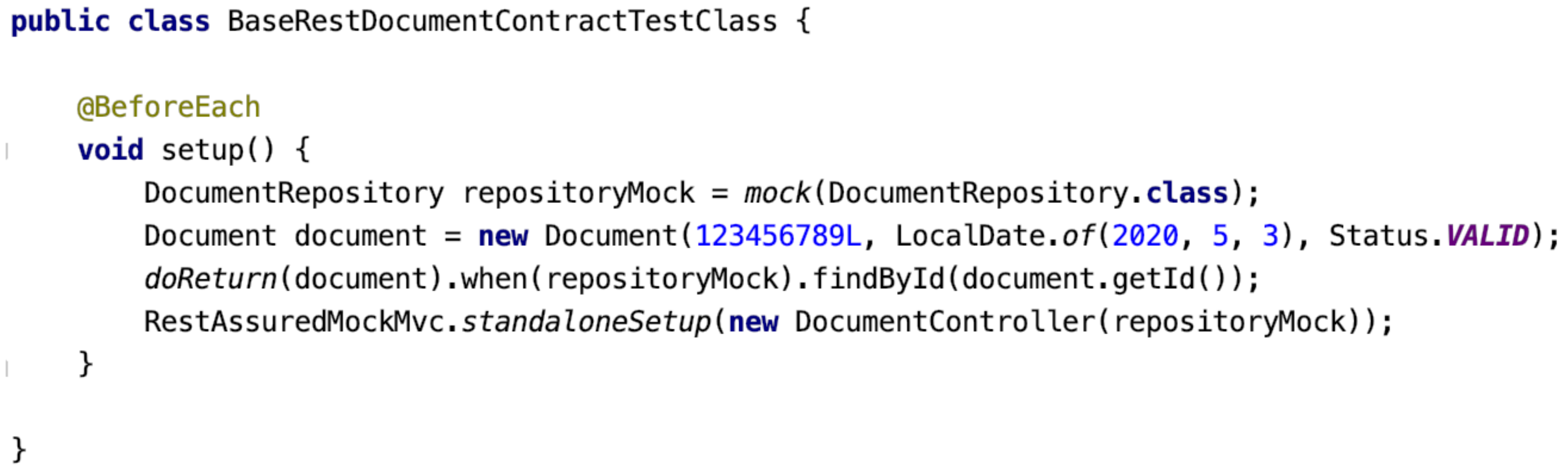
- test framework (JUnit, JUnit5, Spock)
- test mode (MockMvc - default for HTTP, WebTestClient for WebFlux)
- base class used by all contract tests (more than one class can be created)



```
<plugin>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-contract-maven-plugin</artifactId>
    <extensions>true</extensions>
    <configuration>
        <testFramework>JUNIT5</testFramework>
        <baseClassForTests>com.us.restproducer.document.BaseRestDocumentContractTestClass</baseClassForTests>
    </configuration>
</plugin>
```

Configure Base Class (Spring Web)

- for HTTP (Spring Web Mvc) base class should setup RestAssuredMockMvc
- MockMvc in contract test will use this setup to handle request defined in the contract



```java
public class BaseRestDocumentContractTestClass {

    @BeforeEach
    void setup() {
        DocumentRepository repositoryMock = mock(DocumentRepository.class);
        Document document = new Document(123456789L, LocalDate.of(2020, 5, 3), Status.VALID);
        doReturn(document).when(repositoryMock).findById(document.getId());
        RestAssuredMockMvc.standaloneSetup(new DocumentController(repositoryMock));
    }

}
```

Contract Products

- generated test which checks if a producer is compatible with the contract
- stub which consumer uses to verify the communication with a producer

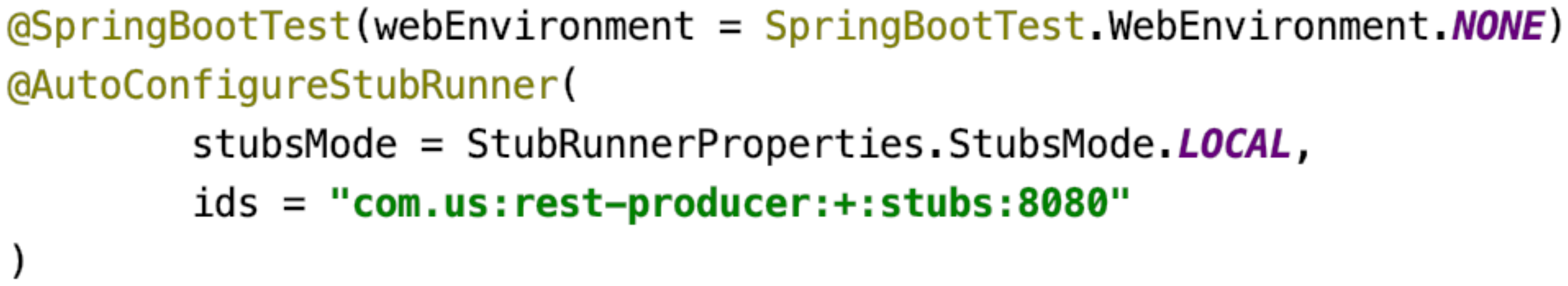
Consumer Setup

- finds generated stubs (on the classpath, from remote location, in local .m2)

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

Consumer Setup

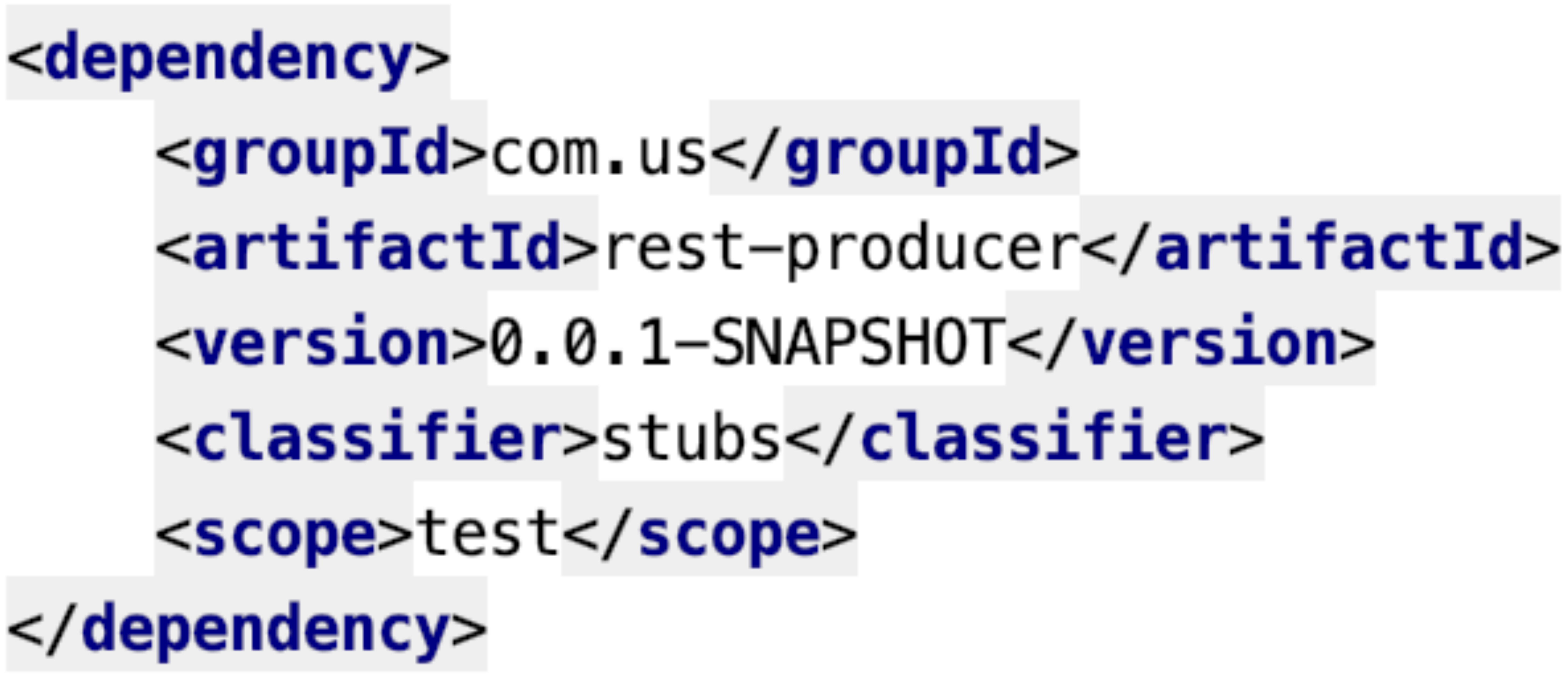
- LOCAL stubs mode search for stubs in local .m2 repository
- id contains: groupId, artifactId, version, classifier, port on which stub should be run
- with this setup the stub provided by rest-producer is running on localhost port 8080
- the consumer verifies the ability to read producer response (check if consumer matches the contract)



```
@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.NONE)
@AutoConfigureStubRunner(
        stubsMode = StubRunnerProperties.StubsMode.LOCAL,
        ids = "com.us:rest-producer:+:stubs:8080"
)
```

Consumer Setup

- the CLASSPATH stub mode searches for stub in the classpath
- stubs can be added as maven test dependency
- maven dependency with ranges
- remote stub mode does not require the maven dependency but some additional settings must be provided
- from version 3.0 the remote mode will be able to use maven repository definitions and user maven settings (credentials)



```xml
<dependency>
    <groupId>com.us</groupId>
    <artifactId>rest-producer</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <classifier>stubs</classifier>
    <scope>test</scope>
</dependency>
```

Contract for Messaging

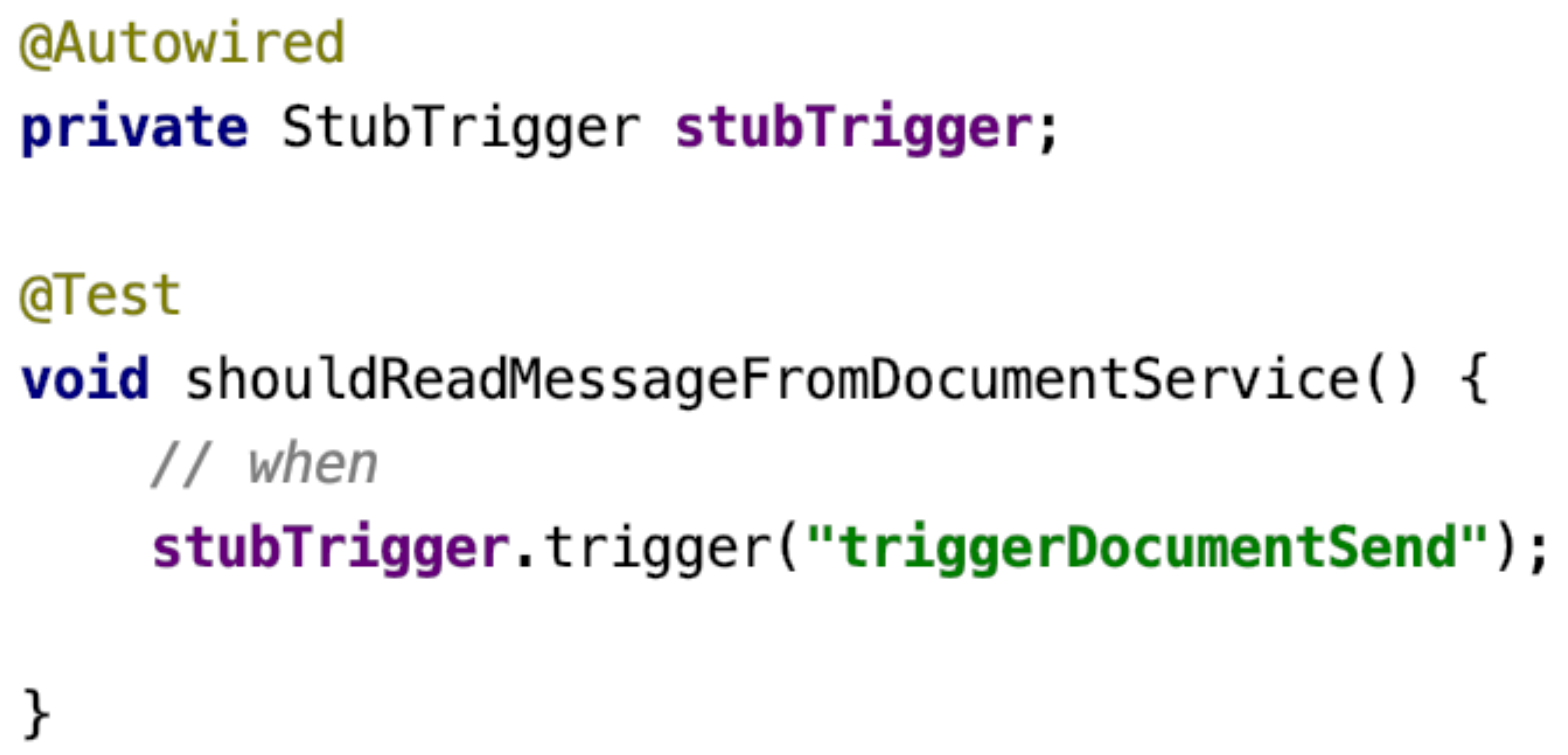
- for the HTTP (Spring Web MVC, Spring Webflux) the contract has request / response definition
- for messaging there is no request
- base class for the producer test must provide a method to generate a message

Contract for Messaging

```
Contract.make {
    description("should send message with validated document")
    label("triggerDocumentSend")
    input {
        triggeredBy("triggerDocumentValidatedMessage()")
    }
    outputMessage {
        sentTo("documentValidated")
        body([
                id  : 123456789,
                status: "VALID"
        ])
        headers {
            contentType(applicationJson())
        }
    }
}
```

Consumer Setup (Messaging)

- stubs which should be used for the test are defined in the same way as for HTTP contracts
- StubTrigger triggers the outputMessage (by label) which is defined in the contract
- it results in the message being received on a defined topic with content and headers match the contract



```
@Autowired
private StubTrigger stubTrigger;

@Test
void shouldReadMessageFromDocumentService() {
    // when
    stubTrigger.trigger("triggerDocumentSend");

}
```

Consumer Driven Contracts

- in TDD the test makes the object API better, simpler, easier to use because the test is a client
- in contract test the API of the service is created based on client expectation
- API design is focused on client ease of use
- API changes are cheaper, faster and easier when recognized or requested as fast as possible