**1. Introduction**

[**Spring Cloud Contract**](http://cloud.spring.io/spring-cloud-contract/)**is a project that, simply put, helps us write**[**Consumer-Driven Contracts (CDC)**](https://martinfowler.com/articles/consumerDrivenContracts.html)**.**

This ensures the contract between a *Producer* and a *Consumer*, in a distributed system – for both HTTP-based and message-based interactions.

In this quick article, we’ll explore writing producer and consumer side test cases for Spring Cloud Contract through an HTTP interaction.

**2. Producer – Server Side**

We’re going to write a producer side CDC, in the form of an *EvenOddController*– which just tells whether the *number* parameter is even or odd:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | @RestController  public class EvenOddController {        @GetMapping("/validate/prime-number")      public String isNumberPrime(@RequestParam("number") Integer number) {          return Integer.parseInt(number) % 2 == 0 ? "Even" : "Odd";      }  } |

**2.1. Maven Dependencies**

For our producer side, we’ll need the [*spring-cloud-starter-contract-verifier*](https://search.maven.org/classic/#search%7Cgav%7C1%7Cg%3A%22org.springframework.cloud%22%20AND%20a%3A%22spring-cloud-starter-contract-verifier%22) dependency:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | <dependency>      <groupId>org.springframework.cloud</groupId>      <artifactId>spring-cloud-starter-contract-verifier</artifactId>      <version>Edgware.SR1</version>      <scope>test</scope>  </dependency> |

And we’ll need to configure [*spring-cloud-contract-maven-plugin*](https://search.maven.org/classic/#search%7Cgav%7C1%7Cg%3A%22org.springframework.cloud%22%20AND%20a%3A%22spring-cloud-contract-maven-plugin%22) with the name of our base test class, which we’ll describe in the next section:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | <plugin>      <groupId>org.springframework.cloud</groupId>      <artifactId>spring-cloud-contract-maven-plugin</artifactId>      <version>1.2.2.RELEASE</version>      <extensions>true</extensions>      <configuration>          <baseClassForTests>              com.baeldung.spring.cloud.springcloudcontractproducer.BaseTestClass          </baseClassForTests>      </configuration>  </plugin> |

**2.2. Producer Side Setup**

We need to add a base class in the test package that loads our Spring context:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | @RunWith(SpringRunner.class)  @SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.MOCK)  @DirtiesContext  @AutoConfigureMessageVerifier  public class BaseTestClass {        @Autowired      private EvenOddController evenOddController;        @Before      public void setup() {          StandaloneMockMvcBuilder standaloneMockMvcBuilder            = MockMvcBuilders.standaloneSetup(evenOddController);          RestAssuredMockMvc.standaloneSetup(standaloneMockMvcBuilder);      }  } |

**In the */src/test/resources/contracts/*package, we’ll add the test stubs**, such as this one in the file *shouldReturnEvenWhenRequestParamIsEven.groovy*:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | import org.springframework.cloud.contract.spec.Contract  Contract.make {      description "should return even when number input is even"      request{          method GET()          url("/validate/prime-number") {              queryParameters {                  parameter("number", "2")              }          }      }      response {          body("Even")          status 200      }  } |

When we run the build, **the plugin automatically generates a test class named *ContractVerifierTest* that extends our *BaseTestClass*** and puts it in */target/generated-test-sources/contracts/*.

The names of the test methods are derived from the prefix “*validate\_”* concatenated with the names of our Groovy test stubs. For the above Groovy file, the generated method name will be *“validate\_shouldReturnEvenWhenRequestParamIsEven”*.

Let’s have a look at this auto-generated test class:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | public class ContractVerifierTest extends BaseTestClass {    @Test  public void validate\_shouldReturnEvenWhenRequestParamIsEven() throws Exception {      // given:      MockMvcRequestSpecification request = given();        // when:      ResponseOptions response = given().spec(request)        .queryParam("number","2")        .get("/validate/prime-number");        // then:      assertThat(response.statusCode()).isEqualTo(200);        // and:      String responseBody = response.getBody().asString();      assertThat(responseBody).isEqualTo("Even");  } |

**The build will also add the stub jar in our local Maven repository so that it can be used by our consumer.**

Stubs will be present in the output folder under *stubs/mapping/*.

**3. Consumer – Client Side**

**The consumer side of our CDC will consume stubs generated by the producer side** through HTTP interaction to maintain the contract, so **any changes on the producer side would break the contract**.

We’ll add *BasicMathController,*which will make an HTTP request to get the response from the generated stubs:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | @RestController  public class BasicMathController {        @Autowired      private RestTemplate restTemplate;        @GetMapping("/calculate")      public String checkOddAndEven(@RequestParam("number") Integer number) {          HttpHeaders httpHeaders = new HttpHeaders();          httpHeaders.add("Content-Type", "application/json");            ResponseEntity<String> responseEntity = restTemplate.exchange(            "<http://localhost:8090/validate/prime-number?number=>" + number,            HttpMethod.GET,            new HttpEntity<>(httpHeaders),            String.class);            return responseEntity.getBody();      }  } |

**3.1. The Maven Dependencies**

For our consumer, we’ll need to add the [*spring-cloud-contract-wiremock*](https://search.maven.org/classic/#search%7Cgav%7C1%7Cg%3A%22org.springframework.cloud%22%20AND%20a%3A%22spring-cloud-contract-wiremock%22) and [*spring-cloud-contract-stub-runner*](https://search.maven.org/classic/#search%7Cgav%7C1%7Cg%3A%22org.springframework.cloud%22%20AND%20a%3A%22spring-cloud-contract-stub-runner%22) dependencies:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | <dependency>      <groupId>org.springframework.cloud</groupId>      <artifactId>spring-cloud-contract-wiremock</artifactId>      <version>1.2.2.RELEASE</version>      <scope>test</scope>  </dependency>  <dependency>      <groupId>org.springframework.cloud</groupId>      <artifactId>spring-cloud-contract-stub-runner</artifactId>      <version>1.2.2.RELEASE</version>      <scope>test</scope>  </dependency> |

**3.2. Consumer Side Setup**

Now it’s time to configure our stub runner, which will inform our consumer of the available stubs in our local Maven repository:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22 | @RunWith(SpringRunner.class)  @SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.MOCK)  @AutoConfigureMockMvc  @AutoConfigureJsonTesters  @AutoConfigureStubRunner(    workOffline = true,    ids = "com.baeldung.spring.cloud:spring-cloud-contract-producer:+:stubs:8090")  public class BasicMathControllerIntegrationTest {        @Autowired      private MockMvc mockMvc;        @Test      public void given\_WhenPassEvenNumberInQueryParam\_ThenReturnEven()        throws Exception {            mockMvc.perform(MockMvcRequestBuilders.get("/calculate?number=2")            .contentType(MediaType.APPLICATION\_JSON))            .andExpect(status().isOk())            .andExpect(content().string("Even"));      }  } |

Note that the *ids* property of the *@AutoConfigureStubRunner* annotation specifies:

* *com.baeldung.spring.cloud* — the *groupId* of our artifact
* *spring-cloud-contract-producer* — the *artifactId* of the producer stub jar
* *8090* — the port on which the generated stubs will run

**4. When the Contract is Broken**

If we make any changes on the producer side that directly impact the contract without updating the consumer side, **this can result in contract failure.**

For example, suppose we’re to change the *EvenOddController* request URI to */validate/change/prime-number* on our producer side.

If we fail to inform our consumer of this change, the consumer will still send its request to the */validate/prime-number* URI, and the consumer side test cases will throw *org.springframework.web.client.HttpClientErrorException: 404 Not Found*.

**5. Summary**

We’ve seen how Spring Cloud Contract can help us maintain contracts between a service consumer and producer so that we can push out new code without any worry of breaking the contracts.

And, as always, the full implementation of this tutorial can be found [over on GitHub](https://github.com/eugenp/tutorials/tree/master/spring-cloud).