Testing Microservices: Contracts, Simulation and Observability

Module Seven: Consumer-Driven Contract Testing

Purpose of this Lab

- Understanding the concept Consumer-Driven Contract (CDC) testing
- Understand the basics of the Pact verification framework

Questions

Feel free to ask questions and ask for help at any point during the workshop. Also, please do collaborate with others.

Pre-requisites

Make sure you have cloned the api-simulation-training Git repository, and that you are in the correct directory and branch! You should also open in your IDE of choice the “consumer” and “flight” Java projects from the root of the api-simulation-training directory

```
$ cd api-simulation-training
$ git checkout contract-testing
$ ls
1-introduction          3-matching              5-stateful
README.md               consumer                flights-service
2-api-simulation-basics 4-dynamic-responses     6-fault-injection
bonus-module            discussions
```

This section of the workshop assumes that you are part of a team that has built a service that consumes the existing “flights” service that you have been working on so far. The consuming

application is written in Java, and the functionality and code will be demonstrated as part of the workshop introduction.

Tip: Explore the FlightConsumer service

Adding a New Recommendation Endpoint
Imagine that you wanted to add support for a new “recommendation” endpoint that was part of the flights service portfolio. This endpoint simply returns a recommendation for your next flight. You sit down with with the flights service team and design the (very simple) endpoint and payload:

```
$ curl localhost:8081/api/v1/recommendations
{
  “location” : “New York”
}
```

You can create a test for this locally in your consuming service by using Hoverfly -- either by creating your own simulation manually, or by using the DSL. However, this will only help you testing locally. What about integration? This is where consumer-based contract testing helps

Adding Pact to Your Demo Consumer Application

In the consumer project’s build.gradle build script, add the Pact consumer dependency “testCompile('au.com.dius:pact-jvm-consumer-junit_2.12:3.5.13')” The full gradle.build file should look as follows:

```
buildscript {
  ext {
     springBootVersion = '1.5.10.RELEASE'
  }
  repositories {
     mavenCentral()
  }
  dependencies {

classpath("org.springframework.boot:spring-boot-gradle-plugin:${springBootVersion}")
  }
}

apply plugin: 'java'
apply plugin: 'eclipse'
apply plugin: 'org.springframework.boot'

group = 'com.example'
```

```
version = '0.0.1-SNAPSHOT'
sourceCompatibility = 1.8

repositories {
  mavenCentral()
}


dependencies {
  compile('org.springframework.boot:spring-boot-starter-web')
  testCompile('org.springframework.boot:spring-boot-starter-test')
  testCompile('io.specto:hoverfly-java:0.10.0')
  testCompile('au.com.dius:pact-jvm-consumer-junit_2.12:3.5.13')
}
```

Creating Your First Pact

Within Java there are three main components to creating a consumer Pact: the JUnit Rule; the Pact creation method; and the Pact Verification method. We will attempt to focus on core CDC concepts, but unfortunately we do need to look at least some Java and JUnit (test harness) syntax.

When coding in Java your tests are placed under the “test” folder using a similar directory/package structure as the main application. Let’s create a RecommendationConsumerPactTest.java class file under the directory test/java/com/example/demo/service/

You will need to use a JUnit Rule to configure Pact:

```
public PactProviderRuleMk2 mockProvider = new
PactProviderRuleMk2("recommendation_provider", "localhost", 8081, this);
```

This rule specifies the provider name, and configures the local run mock server. You can now write your Pact test based on the simple Recommendation Service specification above:

```
@Pact(provider = "recommendation_provider", consumer = "recommendation_consumer")
public RequestResponsePact createPact(PactDslWithProvider builder) {
   Map<String, String> headers = ImmutableMap.of("Content-Type",
MediaType.APPLICATION_JSON_VALUE);

   return builder
           .given("test state")
           .uponReceiving("ExampleJavaConsumerPactRuleTest test interaction")
           .path("/api/v1/recommendations")
           .method("GET")
           .willRespondWith()
           .status(200)
```

```
            .headers(headers)
            .body("{\"location\":\"London\"}")
            .toPact();
}
```

Now that you have the foundations of the Pact complete, you can write your supporting classes You can also attempt to TDD the application by creating a failing test first.

You would want to write a test similar to this:

```
@Test
@PactVerification("recommendation_provider")
public void runTest() {
   Recommendation recommendation = recommendationConsumer.getRecommendation();

   assertThat(recommendation.getLocation(), is("London"));
}
```

The recommendation model object would look something like this simple POJO class

```
public class Recommendation {

   private String location;

   public Recommendation() {
   }

   public String getLocation() {
       return location;
   }
}
```

The recommendation consumer class would look something like this:

```
package com.example.consumer.service;

import com.example.demo.model.Recommendation;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import org.springframework.web.client.RestTemplate;

@Service
public class RecommendationConsumer {

   private RestTemplate restTemplate;

   @Autowired
   public RecommendationConsumer(RestTemplate restTemplate) {
       this.restTemplate = restTemplate;
```

```
    }

    public Recommendation getRecommendation() {
        Recommendation recommendation =

restTemplate.getForObject("http://localhost:8081/api/v1/recommendations",
Recommendation.class);
        return recommendation;
    }
}
```

We can now execute the test and observe the results:

```
11:40:25.877 [main] DEBUG org.apache.http.impl.execchain.MainClientExec -
Connection can be kept alive indefinitely
11:40:25.902 [main] DEBUG org.springframework.web.client.RestTemplate -
Created GET request for "http://localhost:8081/api/v1/recommendations"
11:40:25.939 [main] DEBUG org.springframework.web.client.RestTemplate -
Setting request Accept header to [application/xml, text/xml,
application/json, application/*+xml, application/*+json]
11:40:25.967 [Thread-2] DEBUG au.com.dius.pact.consumer.BaseMockServer -
Received request:       method: GET
      path: /api/v1/recommendations
      query: [:]
      headers: [Accept:application/xml, text/xml, application/json,
application/*+xml, application/*+json, Connection:keep-alive,
Host:localhost:8081, User-agent:Java/1.8.0_151]
      matchers: MatchingRules(rules={})
      generators: Generators(categories={})
      body: OptionalBody(state=EMPTY, value=)
11:40:26.096 [Thread-2] DEBUG au.com.dius.pact.model.RequestMatching$ -
comparing to expected request:
      method: GET
      path: /api/v1/recommendations
      query: [:]
      headers: [:]
      matchers: MatchingRules(rules={})
      generators: Generators(categories={})
      body: OptionalBody(state=MISSING, value=null)
11:40:26.366 [Thread-2] DEBUG au.com.dius.pact.model.Matching$ - Found a
matcher for text/plain ->
au.com.dius.pact.matchers.PlainTextBodyMatcher@c391115
11:40:26.381 [Thread-2] DEBUG au.com.dius.pact.model.RequestMatching$ -
Request mismatch: List()
```

```
11:40:26.455 [Thread-2] DEBUG au.com.dius.pact.consumer.BaseMockServer -
Generating response:    status: 200
      headers: [Content-Type:application/json]
      matchers: MatchingRules(rules={})
      generators: Generators(categories={})
      body: OptionalBody(state=PRESENT, value={"location":"London"})
11:40:26.460 [main] DEBUG org.springframework.web.client.RestTemplate - GET
request for "http://localhost:8081/api/v1/recommendations" resulted in 200
(OK)
11:40:26.461 [main] DEBUG org.springframework.web.client.RestTemplate -
Reading [class com.example.demo.model.Recommendation] as "application/json"
using
[org.springframework.http.converter.json.MappingJackson2HttpMessageConverte
r@39ad977d]
11:40:26.596 [main] DEBUG au.com.dius.pact.consumer.BaseMockServer -
Writing pact recommendation_consumer -> recommendation_provider to file
target/pacts/recommendation_consumer-recommendation_provider.json
```

So far this is nothing new -- you know that you can produce this kind of behaviour with Hoverfly. However, if you conduct a full build and verify of the application (using “gradle check” in this example), you will notice that a “pacts” directory is created under the target directory. There should be a file named “recommendation_consumer-recommendation_provider.json” -- this is the Pact consumer contract file generated as part of the test run.


```
{
   "provider": {
       "name": "recommendation_provider"
   },
   "consumer": {
       "name": "recommendation_consumer"
   },
   "interactions": [
       {
           "description": "A request for recommendation",
           "request": {
               "method": "GET",
               "path": "/api/v1/recommendations"
           },
           "response": {
               "status": 200,
               "headers": {
                   "Content-Type": "application/json"
               },
               "body": {
                   "location": "London"
```

```
                }
            },
            "providerStates": [
                {
                    "name": "no particular state"
                }
            ]
        }
    ],
    "metadata": {
        "pact-specification": {
            "version": "3.0.0"
        },
        "pact-jvm": {
            "version": "3.5.13"
        }
    }
}
```

You can use this for verification on the provider.

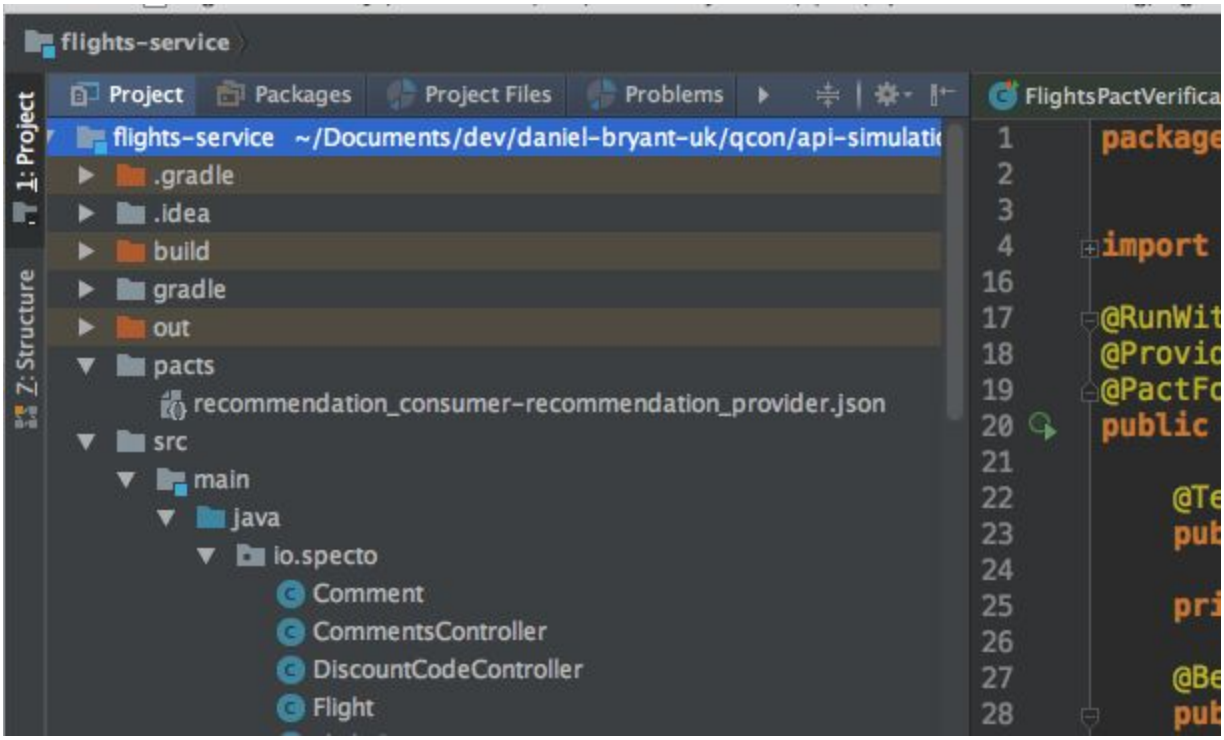
Adding Pact to the Provider

You can now open the flights-service provider in your IDE and navigate to the build.gradle build script. Add the Pact Provider dependency “testCompile('au.com.dius:pact-jvm-provider-junit_2.12:3.5.13')”

```
buildscript {
  ext {
     springBootVersion = '1.5.3.RELEASE'
  }
  repositories {
     mavenCentral()
  }
  dependencies {

classpath("org.springframework.boot:spring-boot-gradle-plugin:${springBootVersion}")
  }
}

apply plugin: 'java'
apply plugin: 'eclipse'
apply plugin: 'org.springframework.boot'

version = '0.0.1-SNAPSHOT'
sourceCompatibility = 1.8

repositories {
```

```
  mavenCentral()
}

dependencies {
  compile('org.springframework.boot:spring-boot-starter')
  compile('org.springframework.boot:spring-boot-starter-web')
  compile('com.google.guava:guava:21.0')
  compile 'com.fasterxml.jackson.datatype:jackson-datatype-jsr310:2.8.6'

  testCompile('org.springframework.boot:spring-boot-starter-test')
  testCompile('au.com.dius:pact-jvm-provider-junit_2.12:3.5.13')
}
```

Now copy the recommendation_consumer-recommendation_provider.json Pact file from the consumer target directory to a directory name “pacts” under the flight-service project directory. The resulting directory structure can be seen below:



You can now add a verification test to the flights-service. Navigate to “test/java/io/specto/” and add a “RecommendationPactVerificationTests” class file.

You will need to add several Java/JUnit-specific commands, but the principles are very similar regardless of what language you are using.

Add the class annotations specified below to inform JUnit to:

- Use the PactRunner class to run the Pact verifications

- Verify the “recommendation_provider”
- Use the Pact contracts found in the “pacts” folder

```
@RunWith(PactRunner.class)
@Provider("flights_provider")
@PactFolder("pacts")
```

You can then configure how the service being verified (in this case the flights-service) is being executed, and also initialise the service using an appropriate method (here, SpringBoot), and configure any state required (in this case nothing).

The entire class without import statements looks like this:

```
@RunWith(PactRunner.class)
@Provider("recommendation_provider")
@PactFolder("pacts")
public class RecommendationPactVerificationTests {

   @TestTarget
   public final Target target = new HttpTarget("http", "localhost", 8081, "/");

   private static ConfigurableWebApplicationContext application;

   @BeforeClass
   public static void start() {
       application = (ConfigurableWebApplicationContext)
SpringApplication.run(FlightsApplication.class);
   }

   @State("test state")
   public void toGetState() {
   }
}
```

If you now execute this verification test, you will see an expected error:

```
Verifying a pact between recommendation_consumer and
recommendation_provider
  Given no particular state
  A request for recommendation
2018-03-04 12:09:21.776  INFO 80200 --- [nio-8081-exec-1]
o.a.c.c.C.[Tomcat].[localhost].[/]       : Initializing Spring
FrameworkServlet 'dispatcherServlet'
```

```
2018-03-04 12:09:21.776  INFO 80200 --- [nio-8081-exec-1]
o.s.web.servlet.DispatcherServlet        : FrameworkServlet
'dispatcherServlet': initialization started
2018-03-04 12:09:21.799  INFO 80200 --- [nio-8081-exec-1]
o.s.web.servlet.DispatcherServlet        : FrameworkServlet
'dispatcherServlet': initialization completed in 22 ms
    returns a response which
      has status code 200 (FAILED)
      includes headers
        "Content-Type" with value "application/json" (OK)
      has a matching body (FAILED)

Failures:

0) A request for recommendation returns a response which has status code
200
      assert expectedStatus == actualStatus
             |              |  |
             200            |  404
                            false

1) A request for recommendation returns a response which has a matching
body
      $ -> Expected location='London' but was missing

        Diff:

        {
        -    "location": "London"
        +    "path": "/api/v1/recommendations",
        +    "timestamp": "2018-03-04T12:09:21.843+0000",
        +    "error": "Not Found",
        +    "status": 404,
        +    "message": "No message available"
        }
```

You can see that although the header matched (with ‘Content-Type’:’application/json’), there were two failures:

1. The status codes did not match
2. The body (payload) did not match

You can now fix these errors by implementing a Recommendations controller

```
@RestController()
@RequestMapping("/api/v1/recommendations")
public class RecommendationsController {

   @RequestMapping(method = GET)
   public Recommendation getRecommendation() {
       Recommendation recommendation = new Recommendation("New York");
       return recommendation;
   }
}
```

(and associated Recommendation model object)

```
public class Recommendation {

   private String location;

   public Recommendation() {
   }

   public Recommendation(String location) {
       this.location = location;
   }

   public String getLocation() {
       return location;
   }
}
```

If you now run the Pact verification again, you will see success:

```
Verifying a pact between recommendation_consumer and
recommendation_provider
  Given no particular state
  A request for recommendation
2018-03-04 13:54:50.634  INFO 84106 --- [nio-8081-exec-1]
o.a.c.c.C.[Tomcat].[localhost].[/]       : Initializing Spring
FrameworkServlet 'dispatcherServlet'
2018-03-04 13:54:50.635  INFO 84106 --- [nio-8081-exec-1]
o.s.web.servlet.DispatcherServlet        : FrameworkServlet
'dispatcherServlet': initialization started
2018-03-04 13:54:50.659  INFO 84106 --- [nio-8081-exec-1]
o.s.web.servlet.DispatcherServlet        : FrameworkServlet
```

```
'dispatcherServlet': initialization completed in 24 ms
    returns a response which
      has status code 200 (OK)
      includes headers
        "Content-Type" with value "application/json" (OK)
      has a matching body (OK)
2018-03-04 13:54:51.460  WARN 84106 --- [           main]
a.c.d.p.p.junit.InteractionRunner        : Set the provider version using
the pact.provider.version property
2018-03-04 13:54:51.462  INFO 84106 --- [       Thread-3]
ationConfigEmbeddedWebApplicationContext : Closing
org.springframework.boot.context.embedded.AnnotationConfigEmbeddedWebApplic
ationContext@5910de75: startup date [Sun Mar 04 13:54:47 GMT 2018]; root of
context hierarchy
2018-03-04 13:54:51.463  INFO 84106 --- [       Thread-3]
o.s.j.e.a.AnnotationMBeanExporter        : Unregistering JMX-exposed beans
on shutdown

Process finished with exit code 0
```

Making the Pact More Realistic

Simply hard-coding the value of “London” into our RecommendationController is not a viable long term solution (unless the business team really wants to sell only flights to London!). You can make the real solution slightly more realistic by randomly returning a location name based on a predefined list:

```
@RestController()
@RequestMapping("/api/v1/recommendations")
public class RecommendationsController {

   @RequestMapping(method = GET)
   public Recommendation getRecommendation() {
       String[] locationSelection = {"London", "New York", "Beijing", "Tokyo"};
       String location = locationSelection[new
Random().nextInt(locationSelection.length)];
       return  new Recommendation(location);
   }
}
```

Now if you run the RecommendationPactVerificationTests again it will intermittently fail:

```
Verifying a pact between recommendation_consumer and
recommendation_provider
  Given no particular state
  A request for recommendation
2018-03-04 14:00:21.584  INFO 84344 --- [nio-8081-exec-1]
o.a.c.c.C.[Tomcat].[localhost].[/]       : Initializing Spring
FrameworkServlet 'dispatcherServlet'
2018-03-04 14:00:21.585  INFO 84344 --- [nio-8081-exec-1]
o.s.web.servlet.DispatcherServlet        : FrameworkServlet
'dispatcherServlet': initialization started
2018-03-04 14:00:21.606  INFO 84344 --- [nio-8081-exec-1]
o.s.web.servlet.DispatcherServlet        : FrameworkServlet
'dispatcherServlet': initialization completed in 21 ms
    returns a response which
      has status code 200 (OK)
      includes headers
        "Content-Type" with value "application/json" (OK)
      has a matching body (FAILED)

Failures:

0) A request for recommendation returns a response which has a matching
body
      $.location -> Expected 'London' but received 'New York'
```

Let's now go back to our Consumer service and the RecommendationConsumerPactTests and update the Pact:

```
@Pact(provider = "recommendation_provider", consumer = "recommendation_consumer")
public RequestResponsePact createPact(PactDslWithProvider builder) {
   Map<String, String> headers = ImmutableMap.of("Content-Type",
MediaType.APPLICATION_JSON_VALUE);

   return builder
           .given("no particular state")
           .uponReceiving("A request for recommendation")
           .path("/api/v1/recommendations")
           .method("GET")
           .willRespondWith()
```

```
            .status(200)
            .headers(headers)
            .body(new PactDslJsonBody().stringType("location"))
            .toPact();
}
```

Notice the ".body(new PactDslJsonBody().stringType("location"))" -- this is how we specify looser matching. You can run a "gradle clean" to remove the old Pact contracts, and a "gradle check" to regenerate them. However, you'll notice that our current test fails:

```
Pact Test function failed with an exception:
Expected: is "London"
     but: was "B2lCiOYZMcrIb0JwSK8p"
```

This is because our looser matching no longer generates the String "London". There are many ways around this -- we could specify a default return value of "London" for testing purposes, or use a Pact "generator" to return the value "London". However, for the moment, let's simply relax our tests (when choosing to do this, you must ensure that your test still verifies the required syntax/semantics of the functionality):

```
@Test
@PactVerification("recommendation_provider")
public void runTest() {
   Recommendation recommendation = recommendationConsumer.getRecommendation();

   assertThat(recommendation.getLocation(), is(instanceOf(String.class)));
}
```

Now, if you re-run the test it should pass, and our new Pact contract is generated in "target/pacts" recommendation_consumer-recommendation_provider.json (you may need to "gradle clean" and "gradle check" again in order to regenerate the pact file). The updated Pact file should look like this:

```
{
   "provider": {
       "name": "recommendation_provider"
   },
   "consumer": {
       "name": "recommendation_consumer"
   },
   "interactions": [
       {
           "description": "A request for recommendation",
           "request": {
               "method": "GET",
               "path": "/api/v1/recommendations"
```

```
            },
            "response": {
                "status": 200,
                "headers": {
                    "Content-Type": "application/json"
                },
                "body": {
                    "location": "string"
                },
                "matchingRules": {
                    "body": {
                        "$.location": {
                            "matchers": [
                                {
                                    "match": "type"
                                }
                            ],
                            "combine": "AND"
                        }
                    }
                },
                "generators": {
                    "body": {
                        "$.location": {
                            "type": "RandomString",
                            "size": 20
                        }
                    }
                }
            },
            "providerStates": [
                {
                    "name": "no particular state"
                }
            ]
        }
    ],
    "metadata": {
        "pact-specification": {
            "version": "3.0.0"
        },
        "pact-jvm": {
            "version": "3.5.13"
        }
    }
}
```

Now, copy this pact to our flights-service, and re-run the tests there:

```
Verifying a pact between recommendation_consumer and
recommendation_provider
  Given no particular state
  A request for recommendation
2018-03-04 14:13:16.268  INFO 84918 --- [nio-8081-exec-1]
o.a.c.c.C.[Tomcat].[localhost].[/]       : Initializing Spring
FrameworkServlet 'dispatcherServlet'
2018-03-04 14:13:16.268  INFO 84918 --- [nio-8081-exec-1]
o.s.web.servlet.DispatcherServlet        : FrameworkServlet
'dispatcherServlet': initialization started
2018-03-04 14:13:16.288  INFO 84918 --- [nio-8081-exec-1]
o.s.web.servlet.DispatcherServlet        : FrameworkServlet
'dispatcherServlet': initialization completed in 20 ms
    returns a response which
      has status code 200 (OK)
      includes headers
        "Content-Type" with value "application/json" (OK)
      has a matching body (OK)
2018-03-04 14:13:17.140  WARN 84918 --- [           main]
a.c.d.p.p.junit.InteractionRunner        : Set the provider version using
the pact.provider.version property
2018-03-04 14:13:17.142  INFO 84918 --- [       Thread-3]
ationConfigEmbeddedWebApplicationContext : Closing
org.springframework.boot.context.embedded.AnnotationConfigEmbeddedWebApplic
ationContext@37cd92d6: startup date [Sun Mar 04 14:13:12 GMT 2018]; root of
context hierarchy
2018-03-04 14:13:17.143  INFO 84918 --- [       Thread-3]
o.s.j.e.a.AnnotationMBeanExporter        : Unregistering JMX-exposed beans
on shutdown
```

Success!

Hopefully this should have introduced the basics of a Pact workflow. If there is time, you can now experiment with the following exercises:

Exercises

1. In the consumer service, retroactively add a Pact to verify the existing FlightsConsumer service functionality
    a. Be aware that the Flights service returns a list of flights, and so you will have to use [PactDslJsonArray.arrayEachLike()] method in the pact body

2. Manually create a new Pact file for a new endpoint of your choosing on the flights-service (the best approach is to copy an existing Pact file, and modify this)
    a. Add this into the “pacts” directory of the flights-service and attempt to implement a matching service in Java