TESTING

How to Test Java Microservices with Pact

Microservice applications present special testing challenges.

by Alex Soto Bueno, Andy Gumbrecht and Jason Porter

April 2, 2020

[This article is based on the "Contract Tests" chapter of the book *Testing Java Microservices* by Alex Soto Bueno, Andy Gumbrecht, and Jason Porter (Manning, 2018). —*Ed.*]

The microservices architecture involves a lot of intercommunication between microservices. Those interactions effectively form a contract between the services: This contract consists of expectations of input and output data as well as preconditions and postconditions.

A contract is formed for each service that consumes data from another service that provides (or produces) data based on the first service's requirements. If the service that produces data can change over time, it's important that the contracts with each service that consumes data from it continue to meet expectations. Contract tests provide a mechanism to explicitly verify that a component meets a contract.

Let's look at the tools you can use to write contract tests and, in particular, Pact, a family of test frameworks that provide support for consumer-driven contract testing. It has official implementations for Ruby, JVM languages, .NET, JavaScript, Go, Python, Objective-C, PHP, and Swift.

In our opinion, the Pact framework is the most widely adopted and mature project on the contract-testing scene. One of its main advantages is its support for almost all major languages used today for writing microservices. In addition, the same concepts can be reused regardless of the programming language, from front end to back end. For these reasons, we strongly believe that Pact is the most generic solution for writing consumer-driven contracts. It adapts well to microservices architectures developed in Java.

The Pact Framework

The Pact framework lets you write contracts on the consumer side by providing a mock HTTP server and a fluent API to define the HTTP requests made from a consumer to a service provider and the HTTP responses expected in return. These HTTP requests and responses are used in the mock HTTP server to mock the service provider. The interactions are then used to generate the contract between a service consumer and a service provider.

Pact also provides the logic for validating the contract against the provider side. All interactions that occur on the consumer are played back in the "real" service provider to ensure that the provider produces the response the consumer expects for given requests. If the provider returns something unexpected, Pact marks the interaction as a failure, and the contract test fails.

Any contract test is composed of two parts: one for the consumer and another for the provider. In addition, a contract file is sent from a consumer to a provider. Let's look at the lifecycle of a contract test using Pact.

Step one. Consumer expectations are set up on a mock HTTP server using a fluent API. Consumer communication takes place with the mock HTTP server handling HTTP requests and responses but never interacting with the provider. This way, the consumer doesn't need to know how to deploy a provider (because it might not be trivial to do so and will probably result in writing end-to-end tests instead of contract tests). The consumer verifies that its client/gateway code can communicate against the mock HTTP server with defined interactions.

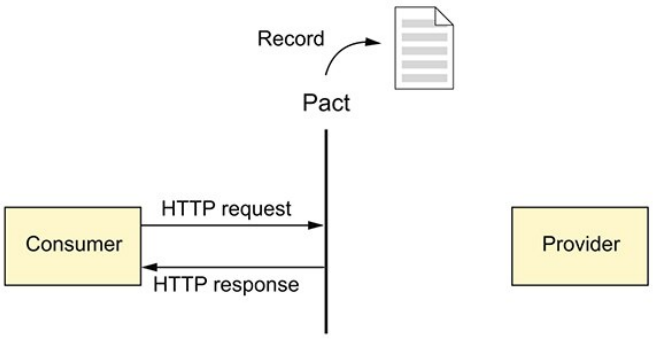
When consumer tests are run, all interactions are written into a pact contract file, which defines the contract that the consumer and provider must follow.

Step two. The pact contract file is sent to the provider project to be replayed against the provider service. The contract is played back against the real provider, and real responses from the provider are checked against the expected responses defined in the contract.

If the consumer is able to produce a pact contract file and the provider meets all the expectations, the contract has been verified by both parties, and they will be able to communicate.

These steps are described in **Figure 1**.

Step 1: Define consumer expectations.



Step 2: Verify expectations on the provider.

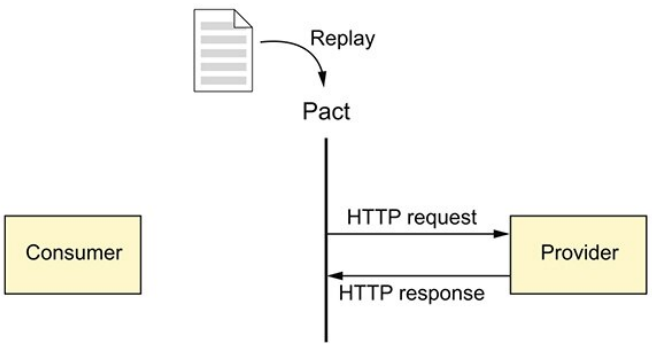


Figure 1. The two steps of the Pact contract-test lifecycle

To summarize, Pact offers the following features:

- It provides a mock HTTP server so you don't have to depend on the provider.
- It provides an HTTP client to automatically replay expectations.
- It uses states to communicate the expected state from the consumer side to the provider before replaying expectations. For example, an interaction might require that the provider database contain a user called Alexandra before replaying an expectation.
- Pact Broker is a repository for contracts, allowing you to share pacts between consumers and providers, versioning pact contract files so the provider can verify itself against a fixed version of a contract, and providing documentation for each pact as well as a visualization of the relationship between services.

Next, let's explore Pact JVM: the implementation of Pact for the Java Virtual Machine.

Pact in JVM Languages

Pact JVM is partially written in Scala, Groovy, and Java, but it can be used with any JVM language. It integrates perfectly with Java, Scala, Groovy, Grails (providing a Groovy DSL for defining contracts), and Clojure. In addition, it offers tight integration with test frameworks such as JUnit, Spock, ScalaTest, and Specs2, as well as build tools such as Maven, Gradle, Leiningen, and sbt. This article focuses on Java tools, but keep in mind that if you plan to use any other JVM language, you can still use Pact JVM for consumer-driven contract testing.

Let's see how to write consumer and provider tests using Pact JVM.

Consumer testing with Pact JVM. Pact JVM provides a mock HTTP server and a Java DSL for writing the expectations of the mock HTTP server. These expectations are materialized into pact contract files when the consumer test passes.

Pact JVM integrates with JUnit, providing a DSL and base classes for use with JUnit to build consumer tests. The first thing you do to write a consumer test using JUnit is register the `PactProviderRule` JUnit rule. This rule does the following:

- Starts and stops the mock HTTP server
- Configures the mock HTTP server with defined expectations
- Generates pact contract files from defined expectations if the test passes

Here's an example:

```
@Rule public PactProviderRule mockProvider =
new PactProviderRule("test_provider", "localhost", 
```

The first argument is the name of the provider that the current consumer contract is defining. This name is used to refer to the provider of a given contract. Next are two optional parameters: the host where the mock HTTP server is bound and the listening port. If values aren't specified, localhost and 8080 are used, respectively. Finally, the `this` instance is the test itself.

Next, you define the expectations by annotating a method with `au.com.dius.pact .consumer.Pact`. This method must receive a

class of type `PactDslWithProvider` and return a `PactFragment`. `PactDslWithProvider` is a Java class that's built around a DSL pattern to provide a description of the request that's expected to be received when a mock HTTP server is used.

As its name suggests, the `PactFragment` object is a fragment of a contract. It's used as the expectation in the mock HTTP server and also to generate a pact contract file that's used to verify the provider. The fragment may be the complete contract or only part of it. If more than one fragment is defined in the same test class, the pact contract file consists of the aggregation of all fragments.

The `@Pact` method must have this signature:

```
@Pact(provider="test_provider", consumer="test_consu
public PactFragment createFragment(PactDslWithProvi
//...
}
```

Notice that in the `@Pact` annotation, you set the name of the provider that should follow the contract and the name of the consumer that's defining the contract. This information is important to ensure the provider-side test is executed against all consumers for which the provider provides data.

The next snippet defines a request/response expectation. `PactDslWithProvider` has several options you can define:

```
return builder
   .uponReceiving("a request for something")
      .path("/hello")
      .method("POST")
      .body("{\"name\": \"Ada\"}")
   .willRespondWith()
      .status(200)
      .body("{\"hello\": \"Ada\"}")
   .uponReceiving("another request for something")
      .matchPath("/hello/[0-9]+")
      .method("POST")
      .body("{\"name\": \"Ada\"}")
   .willRespondWith()
      .status(200)
      .body("{\"hello\": \"Ada\"}")
   .toFragment();
```

This example above defines two expectations. The first request happens when the consumer sends a request using the POST method at `/hello`. The body of the message must contain exactly the JSON document `{"name": "Ada"}`. If this happens, the response is the JSON document `{"hello": "Ada"}`. The second request happens when the path starts with `/hello` followed by any valid number. The conditions are the same as for the first request.

Notice that you can define as many interactions as required. Each interaction starts with `uponReceiving` and is followed by `willRespondWith` to record the response.

By the way, to keep your tests as readable and simple as possible, and to stay focused on the "one method, one task" approach, we recommend using several fragments for all interactions, instead of defining one big `@Pact` method that returns everything.

One important aspect of the previous definitions is that the body content is required to be the same as that specified in the contract. For example, a request for something has a strong requirement that the response be provided only if the JSON document is `{"name": "Ada"}`. If the name is anything other than Ada, the response isn't generated. The same is true for the returned body. Because the JSON document is static, the response is always the same.

This can be a restriction in cases where you can't set a static value, especially when it comes to running contracts against the provider. For this reason, the builder's body method can accept a `PactDslJsonBody` that can be used to construct a JSON body dynamically.

The PactDslJsonBody class. The `PactDslJsonBody` builder class implements a DSL pattern that you can use to construct a JSON body dynamically as well as define regular expressions for fields and type matchers. Let's look at some examples.

The following snippet generates a simple JSON document without an array:

```
DslPart body = new PactDslJsonBody()
   .stringType("name")
   .booleanType("happy")
   .id()
   .ipAddress("localAddress")
   .numberValue("age", 100);
```

Using the xType form, you can also set an optional value parameter that's used to generate example values when returning a mock response. If no example is provided, a random one is generated.

The previous `PactDslJsonBody` definition will match any body like this:

```
{
   "name" : "QWERTY",
   "happy": false,
   "id" : 1234,
   "localAddress" : "127.0.0.1",
   "age": 100,
}
```

Notice that any document containing all the required fields of the required type and having an age field with the value 100 is valid.

`PactDslJsonBody` also offers methods for defining array matchers. For example, you can validate that a list has a minimum or maximum size or that each item in the list matches a given example:

```
DslPart body = new PactDslJsonBody()
   .minArrayLike("products", 1)
      .id()
      .stringType("name")
      .stringMatcher("barcode", "a\\d+", "a1234")
      .closeObject()
   .closeArray();
```

In the example above, the products array can't be empty, and every product should have an identifier and a name of type `string` as well as a barcode that matches the form "a" plus a list of numbers.

If the size of the elements isn't important, you can do this:

```
PactDslJsonArray.arrayEachLike()

   .date("expireDate", "mm/dd/yyyy", date)
   .stringType("name")
   .decimalType("amount", 100.0)
   .closeObject()
```

In the example above, each array must contain three fields: `expireDate`, `name`, and `amount`. Moreover, in the mocked response, each element will contain a date variable value in the `expireDate` field, a random string in the `name` field, and the value 100.0 in `amount`.

As you can see, using `DslPart` to generate the body lets you define field types instead of concrete, specific field/value pairs. This makes your contract more resilient during contract validation on the provider side. Suppose you set `.body("{'name': 'Ada'}")` in the provider-validation phase: You expect the provider to produce the same JSON document with the same values. This may be correct in most cases; but if the test data set changes and, instead of returning `.body("{'name': 'Ada'}")`, it returns `.body("{'name': 'Alexandra'}")`, the test will fail—although from the point of view of the contract, both responses are valid.

Now that you've seen how to write consumer-driven contracts with Pact on the consumer side, let's look at how to write the provider part of the test.

Provider testing with Pact JVM. After executing the consumer part of the test and generating and publishing the pact contract file, you need to play back the contract against a real provider. This part of the test is executed on the provider side, and Pact offers several tools to do so:

- JUnit: A tool that validates contracts in JUnit tests
- Gradle, Leiningen, Maven, and sbt: Plugins for verifying contracts against a running provider
- ScalaTest: An extension to validate contracts against a running provider
- Specs2: An extension to validate contracts against a running provider

In general, all of these integrations offer two ways to retrieve published contracts: by using Pact Broker and by specifying a concrete location (a file or a URL). The way to configure a retrieval method depends on how you choose to replay contracts. For example, JUnit uses an annotations approach, whereas in Maven, a plugin's configuration section is used for this purpose.

Let's see how you can implement provider validation using Maven, Gradle, and JUnit.

Using Maven for verifying contracts. Pact offers a Maven plugin for verifying contracts against providers. To use it, add the following to the plugins section of `pom.xml`:

```
<plugin>
   <groupId>au.com.dius</groupId>
   <artifactId>pact-jvm-provider-maven_2.11</artifa
   <version>3.5.0</version>
</plugin>
```

Then you need to configure the Maven plugin, defining all the providers you want to validate and the location of the consumer contract that you want to use to check them:

```
<plugin>
   <groupId>au.com.dius</groupId>
   <artifactId>pact-jvm-provider-maven_2.11</artifa
   <version>3.2.10</version>
   <configuration>
      <serviceProviders>
         <serviceProvider>
            <name>provider1</name>
            <protocol>http</protocol>
            <host>localhost</host>
            <port>8080</port>
            <path>/</path>
            <pactFileDirectory>path/to/pacts</pactF
         </serviceProvider>
      </serviceProviders>
   </configuration>
</plugin>
```

To verify contracts, execute `mvn pact:verify`. The Maven plugin will load all pact contracts defined in the given directory and replay those that match the given provider name. If all the contracts validate against the provider, the build will finish successfully; if not, the build will fail.

Using Gradle for verifying contracts. The Gradle plugin uses an approach similar to Maven's to verify contracts against providers. To use it, add the following to the plugins section of `.build.gradle`:

```
plugins {
   id "au.com.dius.pact" version "3.5.0"
}
```

Then configure the Gradle plugin, defining the providers you want to validate and the location of the consumer contract you want to use to check them:

```
pact {
   serviceProviders {
      provider1 {
         protocol = 'http'
         host = 'localhost'
         port = 8080 path = '/'
         hasPactsWith('manyConsumers') {
            pactFileLocation = file('path/to/pacts'
         }
      }
   }
}
```

To verify contracts, execute `gradlew pactVerify`. The Gradle plugin will load all Pact contracts defined in the given directory and replay those that match the given provider name. If all the contracts validate against the provider, the build will finish successfully; if not, the build will fail.

Finally, let's see how to validate providers by using JUnit instead of relying on a build tool.

Using JUnit for verifying contracts. Pact offers a JUnit runner for verifying contracts against providers. This runner provides an HTTP client that automatically replays all the contracts against the configured

provider. It also offers convenient out-of-the-box ways to load pacts using annotations.

Using the JUnit approach, you need to register `PactRunner`, set the provider's name with the `@Provider` annotation, and set the contract's location. Then, you create a field of type `au.com.dius.pact.provider.junit.target.Target` that's annotated with `@TestTarget` and instantiates either `au.com.dius.pact.provider.junit .target.HttpTarget` to play pact contract files as HTTP requests and assert the response or `au.com.dius.pact.provider.junit.target.AmqpTarget` to play pact contract files as Advanced Message Queuing Protocol (AMQP) messages. (AMQP is an application layer protocol for message-oriented middleware. The features it defines are message orientation, queuing, routing, reliability, and security.)

Let's look at an example using `HttpTarget`, from `PactTest.java`:

```
@RunWith(PactRunner.class)
@Provider("provider1")
@PactFolder("pacts")
public class ContractTest {
   @TestTarget
   public final Target target = new HttpTarget("loca
}
```

Notice that there's no test method annotated with `@Test`. That isn't required, because rather than having a single test, there are many tests: one for each interaction between a consumer and the provider.

When this test is executed, the JUnit runner gets all the contract files from the pacts directory and replays all the interactions defined in them against the provider location specified in the `HttpTarget` instance.

`PactRunner` automatically loads contracts based on annotations on the test class. Pact provides three annotations for this purpose:

- `PactFolder`: This retrieves contracts from a project folder or resource folder, for example, `@PactFolder("subfolder/in/resource/directory")`
- `PactUrl`: This retrieves contracts from URLs, for example, `@PactUrl(urls = {"http://myserver/contract1.json"})`
- `PactBroker`: This retrieves contracts from PactBroker, for example, `@PactBroker (host="pactbroker", port = "80", tags = {"latest", "dev"})`
- `Custom`: To implement a custom retriever, create a class that implements the `PactLoader` interface and has one default empty constructor or a constructor with one argument of type `Class` (which represents the test class). Annotate the test like this: `@PactSource(CustomPactLoader.class)`

You can also easily implement your own method.

Pact states. When you're testing, each interaction should be verified in isolation, with no context from previous interactions. But with consumer-driven contracts, sometimes the consumer wants to set up something on the provider side before the interaction is run, so the provider can send a response that matches what the consumer expects. A typical scenario is setting up a data source with expected data. For example, when testing the contract for an authentication operation, the consumer may require the provider to insert into a database a concrete login and password beforehand, so that when the interaction occurs, the provider logic can

react appropriately to the data. **Figure 2** summarizes the interaction between consumers, states, and provider.

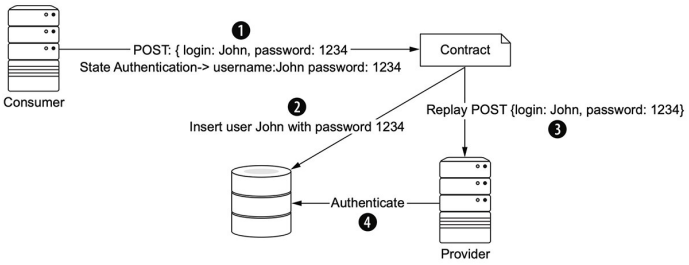


Figure 2. The interaction between the consumer, states, and provider

First, the consumer side defines that the authentication process should be done using a POST method containing a JSON body:

```
{ "login": "John", "password": "1234" }
```

Because this snippet will be used when replaying the contract against the provider, the consumer needs to warn the provider that it should prepare the database with the given information before executing this interaction. For this reason, the consumer creates a state called State Authentication with all the required data. The state information is stored in the contract.

When the contract is replayed against the provider, before the interaction occurs, the state data is injected into the test so the test can prepare the environment for contract validation. Finally, contract validation is executed with the database containing the expected user information.

To define the state from the consumer side, you need to use the special method given when defining the contract:

```
@Override
protected PactFragment createFragment(PactDslWithPr
   Map<String, Object> parameters = new HashMap<>()
   parameters.put("login", "John");
   parameters.put("password", "1234")
   builder
      .given("State Authentication", parameters)
      .uponReceiving("")
      …..
```

To react to a state on the provider side, you need to create a method annotated with `@State`:

```
@State("State Authentication")
public void testStateMethod(Map<String, Object> para
   //Insert data
}
```

Notice that with states, you can share information between the consumer and provider, so you can configure the state of the test before interaction. Pact states are the preferred way to prepare the state of the provider from the consumer side.

Maven and Gradle integrations also provide methods for setting states on the provider side. In these cases, for each provider, you specify a state-change URL to use for changing the state of the provider. This URL receives the `providerState` description from the pact contract file before each interaction, via a POST method.

Conclusion

There are many benefits of using Pact for consumer-driven contract testing:

- Using consumer-driven contracts provides faster execution of tests.
- You won't end up with flaky tests, because with HTTP stub servers, you always receive reliable responses.
- Tests are split between the consumer and provider, so it's easier to identify the cause of a failure.
- Incorporating consumer-driven contracts is a design process.
- "Consumer-driven contracts" doesn't mean "dictator-consumer-driven contracts." The contract is the starting point of a collaborative effort that begins on the consumer side, but both sides must work on it.
- With contract tests, you avoid having to know from the consumer side how to package and deploy the provider side. The consumer side needs to know only how to deploy its part. When you validate the contract on the provider, the provider knows how to deploy itself and how to mock/stub its own dependencies. This is a huge difference from end-to-end tests, where you must start a full environment to be able to run the tests.

A consumer-driven contract may not always be the best approach to follow. Normally it is, but in some situations, you may want to use provider-driven contracts or consumer contracts.

Alex Soto Bueno

Alex Soto Bueno is a Software Engineer at Red Hat in the Developers group. He is passionate about the Java world and software automation, and he believes in the open source software model. Alex is the creator of the NoSQLUnit project, a member of the JSR 374 (Java API for JSON processing) Expert Group, the co-author of *Testing Java Microservices* for Manning and the *Istio Refcard*, and a contributor to several open source projects. A Java champion since 2017 and international speaker, he has talked to audiences about new testing techniques for microservices, continuous delivery in the 21st century, and Java. Follow him at @alexsotob.

Andy Gumbrecht

Andy Gumbrecht is an Apache TomEE PMC member, developer and former evangelist at Tomitribe. Now working as a Software Architect at Phoenix-Contact in Germany, he is still an active contributor of Apache projects including OpenEJB/TomEE and The Arquillian testing framework. Andy has been using in production environments and contributing to Apache OpenEJB and TomEE since 2009. Andy has been fitting in tight code since getting a Sinclair ZX81 with a whopping 1K memory back in 1982. Find him on Twitter @AndyGeeDe.

Jason Porter

Jason Porter is a software engineer currently working on the Red Hat Developer Program Team, Arquillian, Quarkus, and other developer experience projects within Red Hat. His specialties include Wildfly, Quarkus, CDI, JSF, Java EE, solr, and Gradle. He has worked with PHP, Ruby, Groovy, SASS, the rest of the web language arena (HTML, CSS, JS). His current position as Senior Software Engineer at Red Hat has him work primarily on the developers.redhat.com website, He's very interested in the developer experience and helping to improve it at all aspects. Follow him at @lightguardjp.

Share this Page