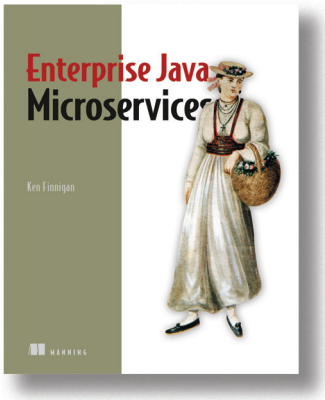
Author Picks

Exploring Testing Java Microservices

Chapters Selected by Alex Soto Bueno and Jason Porter

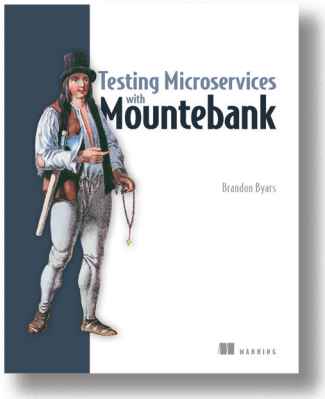
manning



Enterprise Java Microservices

by Ken Finnigan

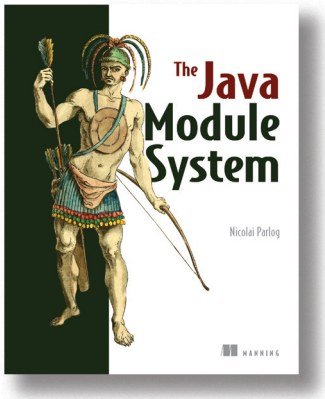
ISBN 9781617294242
272 pages
$39.99
September 2018



Testing Microservices with Mountebank

by Brandon Byars

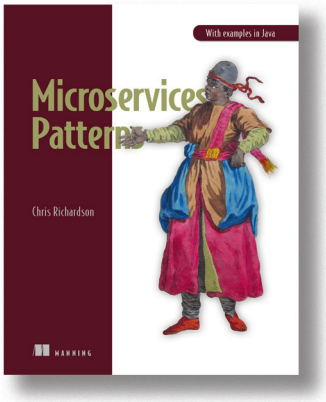
ISBN 9781617294778
240 pages
$39.99
December 2018



The Java Module System

by Nicolai Parlog

ISBN 9781617294280
400 pages
$35.99
May 2019



Microservices Patterns

by Chris Richardson

ISBN 9781617294549
520 pages
$39.99
October 2018

Exploring Testing Java Microservices

Chapters Selected by Alex Soto Bueno and Jason Porter

Manning Author Picks

For online information and ordering of these and other Manning books, please visit www.manning.com. The publisher offers discounts on these books when ordered in quantity.

For more information, please contact

Special Sales Department
Manning Publications Co.
20 Baldwin Road
PO Box 761
Shelter Island, NY 11964
Email: Candace Gillhoolley, cagi@manning.com

Manning Publications Co.
20 Baldwin Road Technical
PO Box 761
Shelter Island, NY 11964

Cover designer: Leslie Haimes

ISBN: 9781617297182
Printed in the United States of America
1 2 3 4 5 6 7 8 9 10 - EBM - 24 23 22 21 20 19

contents

introduction

DevOps has grown in popularity over the last few years, particularly within software companies that want to reduce their lead time to be measured in days and weeks as opposed to months and years—without compromising quality.

If you want your software to do the right things and do them well, you need to test it implacably. However, the testing phase becomes a bottleneck in the software development process, slowing down product release. This is even more apparent in a microservices architecture, since you're dealing with a distributed system.

As developers, we have a responsibility to deliver high-quality software. As architectures evolve, we need new tools to make this work smooth and easy to do. In this minibook, you'll learn some techniques and tools that will help you write automatic tests for your application based on microservices.

Chapter 6 from *Testing Java Microservices*
by Alex Soto Bueno, Andy Gumbrecht
and Jason Porter

Contract tests

This chapter covers

- Understanding and writing contract tests
- Using consumer-driven contracts
- Working with Pact JVM
- Integrating with Arquillian

So far in this book, you've learned about unit, component, and integration tests. One thing they have in common is that they don't test the entire application, but rather isolated parts of it. With unit tests, the unit under test consists of only one or a few classes; with integration tests, you test whether boundaries can connect to a real service. This is the first chapter in which you'll write tests to understand the application as a whole. In this chapter, you'll learn why it's important to use contract tests to verify the entire system, and how to write them.

6.1 Understanding contracts

The microservices architecture involves a lot of intercommunication between microservices. In this book's Gamer example, you saw interactions between the aggregator service, the video service, the comments service, and so on. These interactions effectively form a contract between the services: this contract consists of expectations of input and output data as well as pre- and post-conditions.

A contract is formed for each service that *consumes* data from another service, which 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.

6.1.1 Contracts and monolithic applications

In a monolithic application, services are developed in the same project, side by side. What makes them look different is that each service is developed in a separate module or subproject, running under the same runtime.

In this kind of application, you don't need to worry about breaking the contract (or compatibility) between services, because there's an invisible verifier called a *compiler*. If one service changes its contract, the compiler will reject that build due to a compilation error. Let's look at an example. Here's serviceA.jar:

```
public class ServiceA {
  void createMessage(String parameterA, String parameterB) {}
}
```

And this is serviceB.jar:

```
public class ServiceB {
  private ServiceA serviceA;

  public void callMessage() {
    String parameterA;
    String parameterB;
    // some logic for creating message
    serviceA.createMessage(parameterA, parameterB);
  }

}
```

The two services, service A and service B, are developed in two different JAR files. Service A calls service B by calling its method `createMessage`, which requires that you pass it two `String` parameters. And precisely this method is the contract between both services.

But what if service A changes its contract to something like the following?

```
public class ServiceA {
  void createMessage(String parameterA, Integer parameterB) {}
}
```

The method signature has been changed to receive one `String` and one `Integer`. This breaks compatibility with service B (one consumer of the service). This isn't an issue in a monolithic application, because you'll get a compilation error informing you that method `createMessage` requires `(String, Integer)` but `(String, String)` was found. Thus it's quick and easy to detect when a contract is broken.

From the point of view of testing, modules can be set in the test logic by instantiating them using the `new` keyword or, with container support like context dependency injection (CDI) or Spring inversion of control (IoC), by using Arquillian or the Spring Test Framework. But in with a microservices architecture, things become more complex and harder to detect. If a contract between two services is broken, it may not be detected for quite some time.

6.1.2 Contracts and microservice applications

Each microservice has its own lifecycle, is deployed in its own runtime, and lives remotely from other microservices. In this scenario, any change to the contract of one service can't be caught by the compiler. Figure 6.1 illustrates how each service runs in a different runtime.

Breaking the compatibility between services could happen and would be hard to detect. It's easier to break compatibility because you don't have direct feedback that something has been broken. It's harder to detect that you've broken compatibility because, depending on the kind of (or lack of) tests you run, you may find the problem in (pre)production environments.

Usually, each service is developed by a different team, making compatibility issues more complicated to detect if there isn't good communication between teams. In our

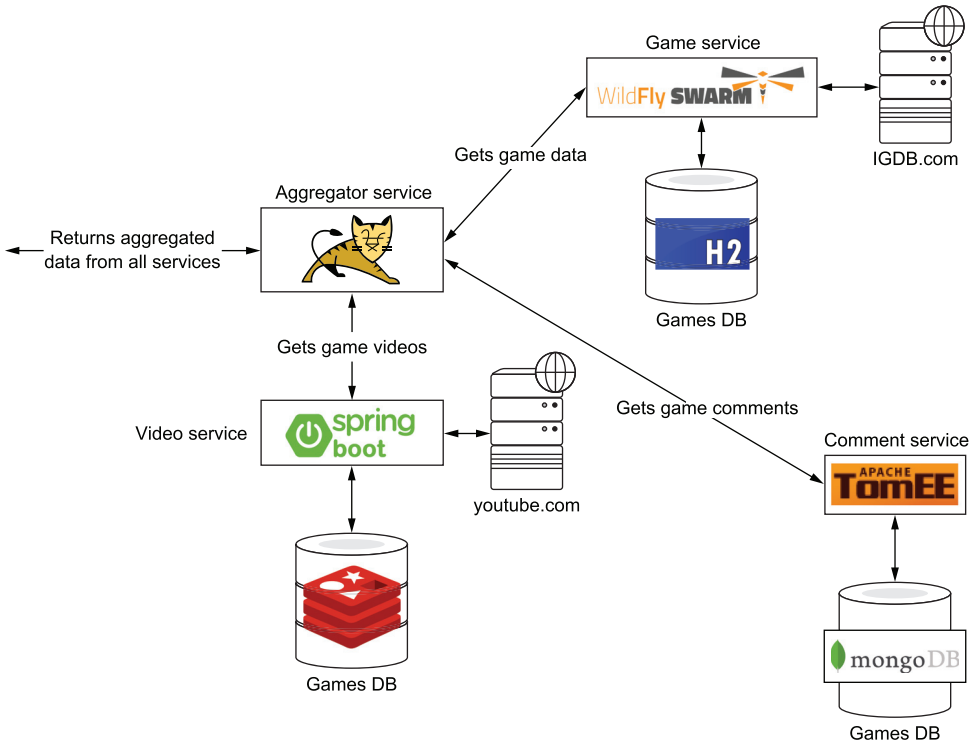


Figure 6.1 Big-picture overview of the example application

experience, the most common problems come from a change in the *provider* side so that the *consumer* can't interact with the provider.

The most common issues are these:

- A service renames its endpoint URL.
- A service adds a new mandatory parameter.
- A service changes/removes an existing parameter.
- A service changes the validation expression of input parameters.
- A service changes its response type or status code.

Consider an example where you have two services, producer and consumer A. The producer service exposes a blog-post resource in JSON format, which is consumed by consumer A.

A possible representation of this document might be thus:

```
{
  "id" : 1,
  "body" : "Hello World",
  "created" : "05/12/2012",
  "author" : "Ada"
}
```

The message contains four attributes: `id`, `body`, `created`, and `author`. Consumer A interacts only with the `body` and `author` fields, ignoring the others. This is summarized in figure 6.2.

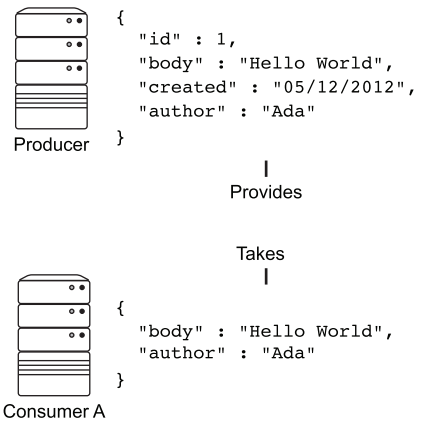


Figure 6.2 Data exchange between producer and consumer A

After some time, a new consumer consumes the producer resource API. This new consumer, consumer B, requires both the `author` and `body` fields, as well a new field (`author id`) that's the identifier for the blog post's author.

At this point, the maintainers of the producer service can take two different approaches:

- Add a new field at the root level.
- Create a composite object with the `author` field.

The first approach adds a new field to the document called `authorId` at same level as `author`. A representation of this document might be as follows:

```
{
  "id" : 1,
  "body" : "Hello World",
  "created" : "05/12/2012",
  "author" : "Ada",
  "authorId" : "123"
}
```

With this change, consumer B's requirements are met. If consumer A follows *Postel's Law*, it will be able to continue consuming messages from the producer service.

The robustness principle

The *Robustness Principle*, also known as *Postel's Law*, comes from Jon Postel. He wrote an early specification of the TCP protocol and asserted the following:

> *Be conservative in what you do, be liberal in what you accept from others.*
>
> —Jon Postel

This was an awful principle when applied to HTML, because it created the ridiculous browser battles that have now been largely resolved by a much stricter HTML5 specification. But for payload parsing, this principle still holds true. In other words, adapted to our case, producers and consumers should ignore any payload fields that aren't important to them.

Figure 6.3 shows that both consumers can still consume messages from the provider. But suppose the maintainers decide on the second approach and create a composite object from the `authorInfo` field:

```
{
  "id" : 1,
  "body" : "Hello World",
  "created" : "05/12/2012",
  "author" : {
      "name" : "Ada",
      "id" : "123"
  }
}
```

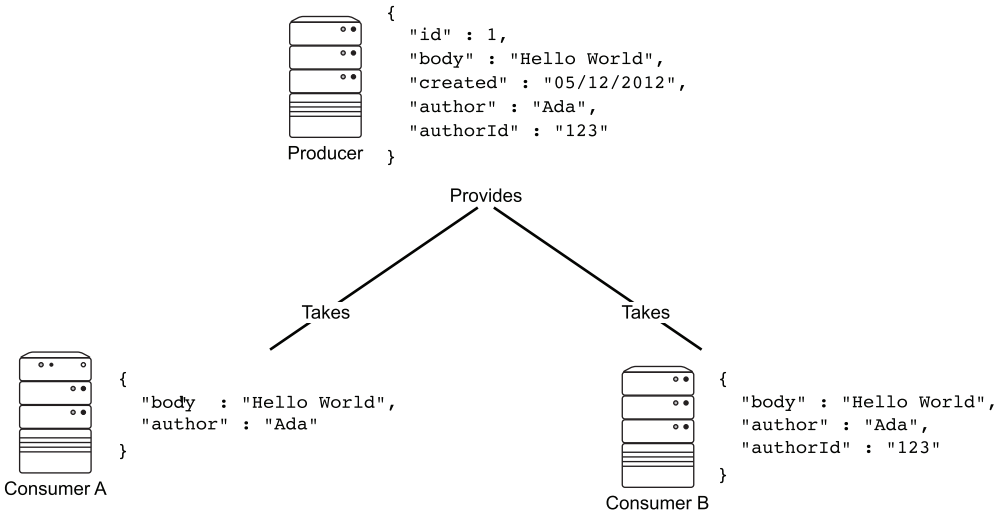


Figure 6.3 Data exchange between producer and consumers A and B

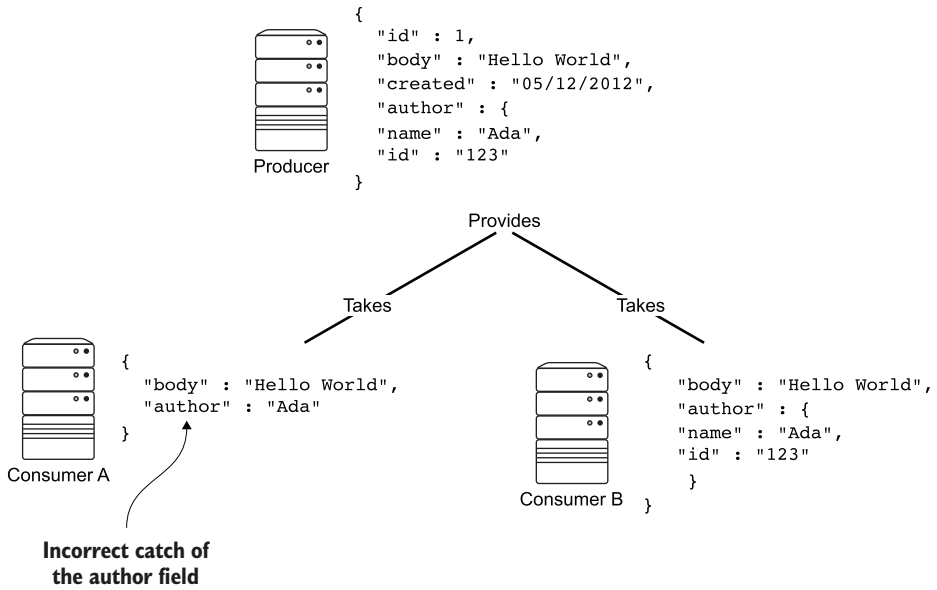


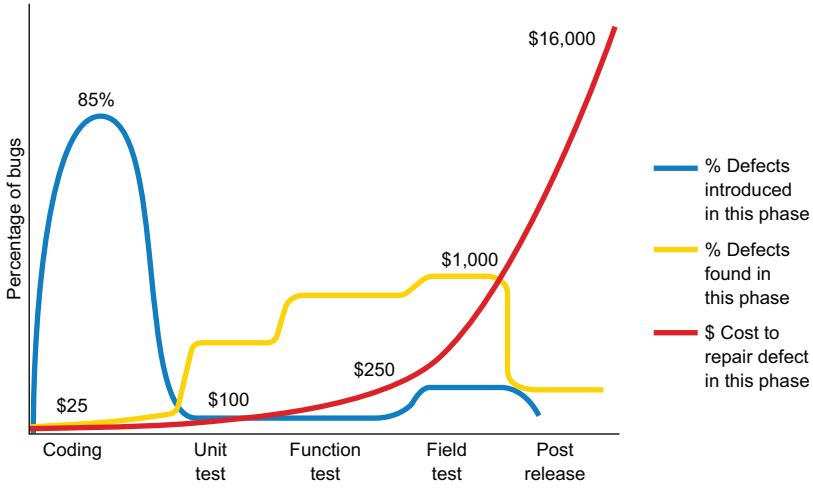
Figure 6.4 **Updated data-exchange scheme**

With this change, consumer B's requirements are met, but the change breaks compatibility with consumer A. In figure 6.4, you can see that although consumer B can process messages from the provider, consumer A can't, because it expects an `author` field of type `string`.

This change would be caught at compilation time if you were using the monolithic approach, but in this case, you don't know about it immediately. From the producer's point of view, even if all of its tests pass, you still don't know that the contract has been broken.

This problem will occur with consumer services, when the new producer service is deployed into a full environment with all services running and operating normally. At this point, consumers will begin operating incorrectly, because the contract has been broken. A new patch should be developed for all consumer services that have adapted the producer API and are now failing.

The later you catch a bug, the harder it is to fix it—and, depending on the phase in application deployment, the urgency can be severe. Suppose you found this bug in a production environment. At this phase, you'd need to roll back the new producer service to the old one, as well as all consumers that have been updated, to get the environment up and running again. Then you'd spend a substantial amount of time determining why the deployment failed and fixing it. The graph in figure 6.5 shows the cost of a bug found during different phases of a project.



Source: Applied Software Measurement *by Capers Jones (McGraw-Hill, 1996)*

Figure 6.5 Costs of fixing bugs in specific development phases

Using a microservices architecture implies changing the way services are tested to detect such issues before a new producer service is deployed. Ideally, bugs should be detected in your CI/delivery run during the testing stage.

Deprecation method

You can also mix the two approaches to solving this problem, by deprecating the `author` field instead of removing it. The new document looks like this:

```
{
  "id" : 1,
  "body" : "Hello World",
  "created" : "05/12/2012",
  "author" : "Ada",
  "authorInfo" : {
      "name" : "Ada",
      "id" : "123"
  }
}
```

A new field called `authorInfo` has been created, and `author` is still valid but deprecated. This approach doesn't replace any tests, because you'll still have the same problem whenever you decide to remove the deprecated fields. But at least there's a transition, and there may be time for consumer maintainers to be notified about the change and adapt to it.

6.1.3 Verifying with integration tests

In chapter 5, you saw that you use integration tests to test whether it's possible for one system to communicate with another. Expressed in contract terms, you're testing that the *boundary* or *gateway* class of a given consumer can communicate correctly with a provider to get or post some data.

You may think that integration tests cover the use case where a contract is broken. But this approach has some issues that make running such tests for services difficult.

First, the consumer must know how to boot up the provider. Second, the consumer may depend on several providers. Each provider may have different requirements, such as a database or other services. So, starting a provider can imply starting several services and, without noticing it, converting integration tests into end-to-end tests.

The third and most important issue is that you need to create a direct relationship between the producer and all of its consumers. When any change is made to the producer, integration tests from all consumers related to this producer must be run to ensure that they can still communicate with the provider. This arrangement is difficult to maintain, because for each new consumer, you need to notify the producer team and provide a new set of running tests.

Although integration tests might be a solution for verifying that consumers of one producer can connect, such tests may not always be the best approach to follow.

6.1.4 What are contract tests?

As mentioned, a contract is a list of agreements between a service that acts as a client (or consumer) and another service that acts as a provider. The existence of a contract defining interactions between each consumer and its provider fixes all the problems described in section 6.1.3.

In figure 6.6, a contract between consumers and a provider is defined—let's say, by having a file describe it—and thus both provider and consumers have an agreement to follow. Now the relationship between consumers and the provider is indirect, because from the producer's point of view, you only need to verify that it meets the agreement described in the contract. The provider doesn't need to run the consumer's integration tests; it only needs to test that the consumer is able to consume requests and produce responses following the contract.

In this case, each provider has a/many contract(s) between it and all the consumers that are provisioning data. For every change made to a provider, all contracts are verified to detect any break, without having to run integration tests.

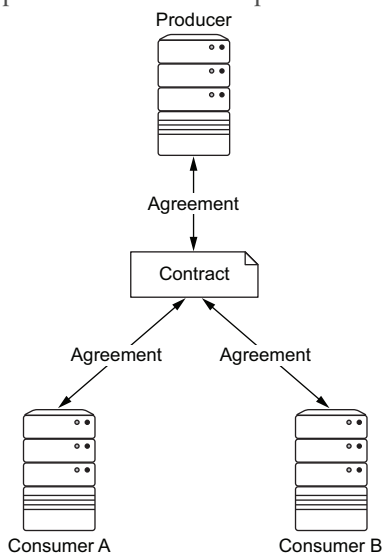


Figure 6.6 Provider and consumer interactions

The contract is also validated on the consumer side, verifying that its client classes (*gateways*) follow the contract. Notice that, again, you don't need to know how a producer is booted up or start any external dependency that may depend on it, because verifying the contract doesn't imply starting the producer; you're only verifying that the consumer also meets the contract.

Tests that verify contracts are known as *contract tests*. The next big question is, who's responsible for creating and maintaining contract files? We'll address this next.

6.1.5 Who owns the contracts?

As you just learned, the best way to validate that a consumer and a provider can communicate correctly and continuously is to define a contract between them. But we haven't addressed who has ownership of this contract: the consumer team or the provider team.

What are provider contracts?

If the ownership of the contract lies with the team that's developing the provider, this implies that they not only know the business behavior of their own service (the provider) but also the requirements of all consumers their service supports. This kind of contract is called a *provider contract* because the contract belongs to the provider, and consumers are merely viewers of it (see figure 6.7). One example of where such a contract might be beneficial is an internal security authentication/authorization service, where consuming services must conform to the provider contract.

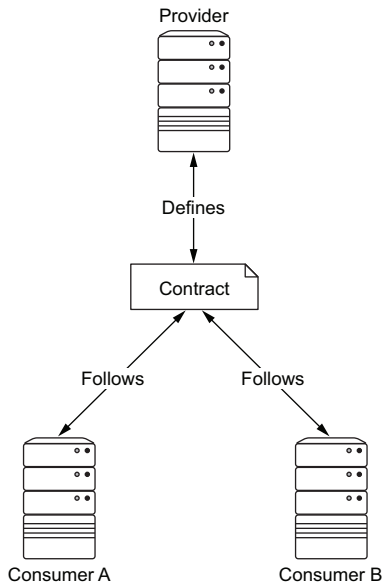


Figure 6.7 Provider contracts

Provider contracts define what the provider will offer to consumers. Each consumer must adapt to what the provider offers. Naturally, this implies that the consumer is coupled to the provider. If the contract developed by the provider team no longer satisfies the requirements of a consumer, the consumer team must start a conversation with the maintainers of the provider service to address this deficiency.

What are consumer contracts?

On the other hand, to fix the problem of one-size-fits-all contracts without forcing the provider team to define a complete contract, you can change the ownership of the contract by making the developers of the consumer service define what they need and send that contract to the provider team to implement. Such a contract is called a *consumer contract* because it belongs to the consumer.

Consumer contracts define the consumer's needs from the consumer's point of view. Hence this contract applies only to that individual consumer and its particular

use case. Consumer contracts can be used to complete an existing provider contract (if there is one), or they can help develop a new one. Figure 6.8 shows that there's one consumer contract for each provider-consumer relationship instead of a single contract for all consumers.

As an example, consumer contracts might be beneficial for an internal checkout service in an organization, where the pace of the service's evolution will be controlled by the provider, but the data pulled from this service is used in different contexts. These contexts may evolve individually, but there's an internal locus of control and evolution.

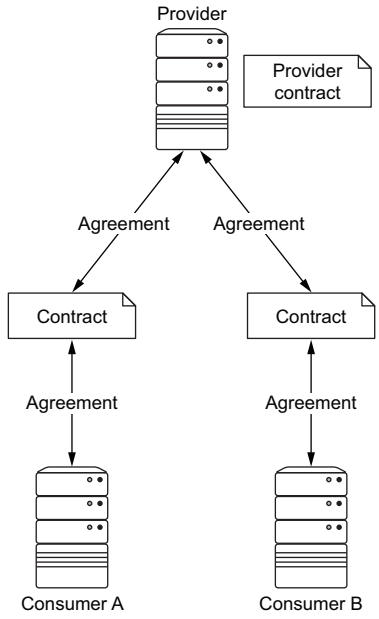


Figure 6.8 **Consumer contracts**

WHAT ARE CONSUMER-DRIVEN CONTRACTS?

A *consumer-driven contract* represents an aggregation of all the contracts a provider has with all of its consumers (see figure 6.9). Obviously, a provider can evolve or extend consumers' contracts by creating a provider contract, as long as the provider's obligations to all consumers are satisfied.

Consumer-driven contracts establish that a service provider is developed from its consumers' perspective. Each consumer communicates to its provider the specific

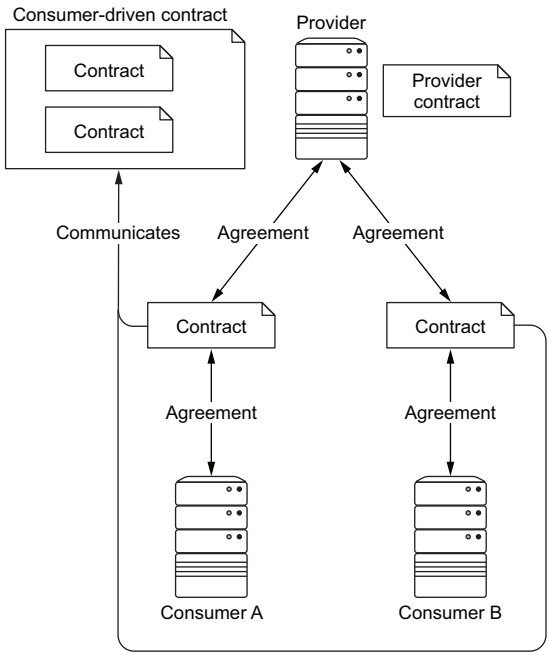


Figure 6.9 **Consumer-driven contracts**

requirements for meeting that consumer's use cases. These requirements create an obligation on the provider's side to meet all the expectations of the consumers.

Ideally, contract tests are developed, bundled by the consumer team, and sent to the producer team so they can develop the producing service. By assigning ownership of consumer contracts to consumer teams, you ensure that a provider's consumer-driven contract is what consumers need, rather than a provider's interpretation of consumer expectations. In addition, when the maintainers of the producing service change the provider-service code base, they know the impact of their changes on consumers, ensuring that there's no communication failure at runtime when the new version of the providing service is deployed.

There are also some obligations on the consumer side. Providers are obliged to follow a consumer-driven contract, and consumers must ensure that they follow their part of the contract—nothing more, nothing less. Consumers should consume only what they need from the provider side. This way, consumers protect themselves against evolution of the provider contract resulting from the provider adding to it.

One example of where consumer-driven contracts may be beneficial is an *external* (open to an outside organization) or *internal* user service that's used across the organization. Data is pulled from this service for multiple other contexts. These contexts all evolve individually, and there's an external locus of control/evolution.

So far in this chapter, we've discussed the problem of having services running in different runtimes, why integration tests aren't enough, and why consumer-driven contracts help you fix communication problems that may appear when updating the producer service. Let's see how to practice these principles, as well as which tools can help you use the consumer-driven-contracts pattern for the microservices architecture.

6.2 Tools

We've explained why it's important to write contract tests in the microservices architecture to avoid surprises when a service is deployed to production. Next, let's look at the tools you can use to write contract tests. These are the three most popular tools:

- *Spring Cloud Contract*—A test framework developed under the Spring ecosystem, in Groovy. Although it integrates well with Spring products, it can be used by any application developed with a JVM language.
- *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 languages.
- *Pacto*—A test framework for developing consumer-driven contract tests and/or document-driven contracts. It's written in Ruby, although it can be used with several languages such as Python and Java by using the Pacto server.

In our opinion, Pact (https://docs.pact.io) 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; the same concepts

can be reused independently of the programming language, from frontend to backend. 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 next section explores in depth how Pact works and how to write contract tests with Pact.

6.2.1 Pact

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 consumer to 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 service consumer and 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 consumer to provider. Let's look at the lifecycle of a contract test using Pact:

1. 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/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.
2. 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, then the contract has been verified by both parties, and they will be able to communicate.

These steps are summarized in figure 6.10.

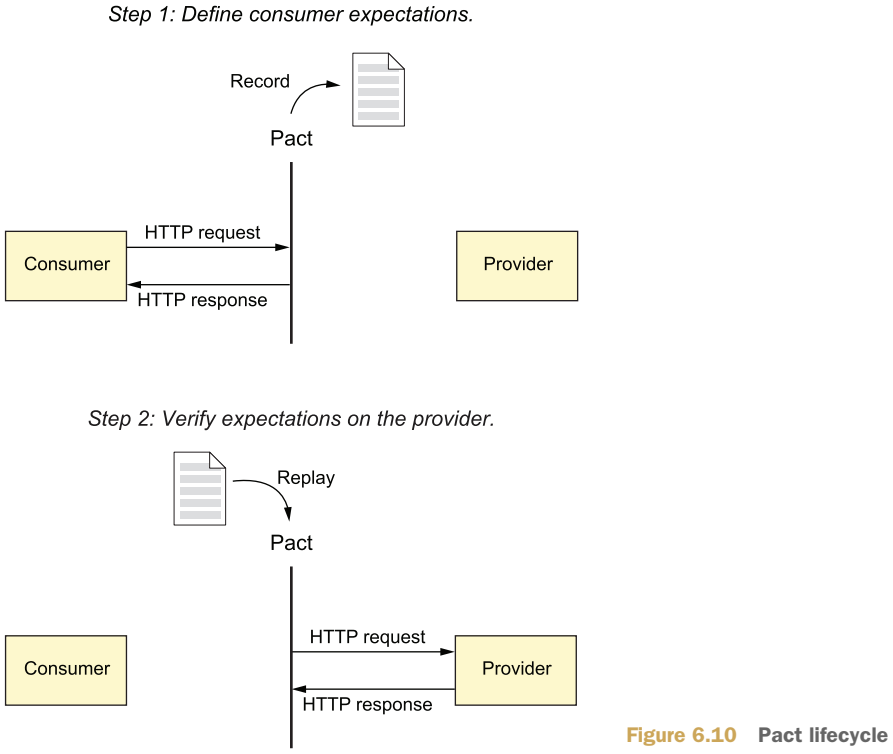


Figure 6.10 Pact lifecycle

In summary, Pact offers the following features:

- A mock HTTP server so you don't have to depend on the provider.
- An HTTP client to automatically replay expectations.
- 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 contains 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, we'll explore Pact JVM: the implementation of Pact for the Java virtual machine.

6.2.2 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 like JUnit, Spock, ScalaTest, and Specs2, as well as build tools such as

Maven, Gradle, Leiningen, and sbt. This book 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 domain-specific language (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", 8080,
                                                 this);
```

The first argument is the provider name 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.

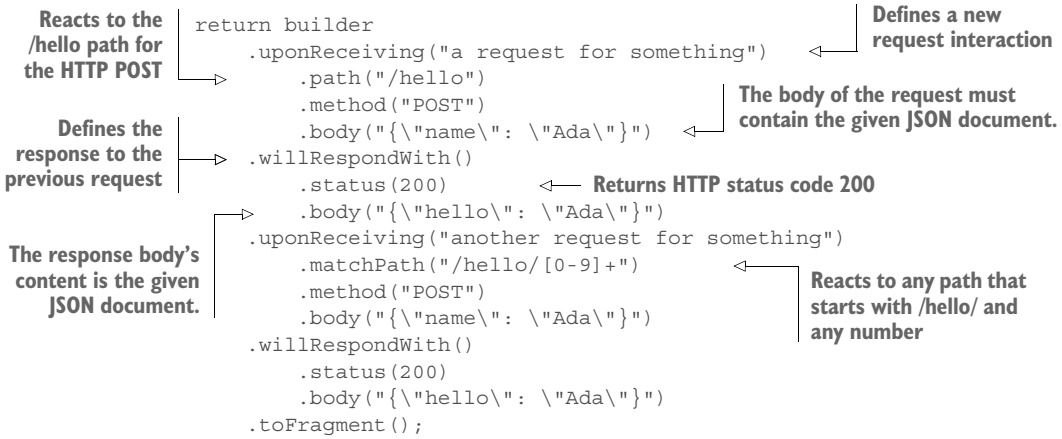
Next, you define the expectations by annotating a method with `au.com.dius.pact.consumer.Pact`. This method must receive a class of type `PactDslWithProvider` and return a `PactFragment`. `PactDslWithProvider` is a Java class that's built around a DSL pattern to provide a description of the request that's 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_consumer")
public PactFragment createFragment(PactDslWithProvider builder) {
  //...
}
```

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 for provider test execution, to be sure 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 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, then 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`, followed by `willRespondWith` to record the response.

> **TIP** 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 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`, then 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);
```

Defines a field named name, of type string, where value isn't important

Defines a field named happy, of type boolean, where value isn't important

Defines a field named age, of type number, with the specific value 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();
```

Defines that the list must contain at least one element

Specifies that each document in the list must contain an ID, a name, and a barcode

Here, 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.

(continued)

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 this example, 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 dataset 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*—Validates contracts in JUnit tests
- *Gradle, Lein, Maven, sbt*—Plugins for verifying contracts against a running provider
- *ScalaTest*—Extension to validate contracts against a running provider
- *Specs2*—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 examine 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.

Listing 6.1 Adding the Maven plugin

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

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

Listing 6.2 Configuring the Maven plugin

```
<plugin>
  <groupId>au.com.dius</groupId>
  <artifactId>pact-jvm-provider-maven_2.11</artifactId>
  <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</pactFileDirectory>
      </serviceProvider>
    </serviceProviders>
  </configuration>
</plugin>
```

Provider (or providers) to verify

Name of the provider (must be unique) and the location where it's deployed

Directory where all Pact contracts are stored

To verify contacts, 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, then 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.

Listing 6.3 Adding the Gradle plugin

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

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

Listing 6.4 Configuring the Maven plugin

```
pact {

  serviceProviders {

    provider1 {
      protocol = 'http'
      host = 'localhost'
      port = 8080
      path = '/'

      hasPactsWith('manyConsumers') {
        pactFileLocation = file('path/to/pacts')
      }
    }
  }
}
```

Provider (or providers) to verify

Name of the provider (must be unique) and the location where it's deployed

Directory where all Pact contracts are stored

To verify contacts, 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, then 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 AMQP messages.

> **Note**
> *Advanced Message Queuing Protocol* (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.

Listing 6.5 Using the JUnit runner

```
@RunWith(PactRunner.class)
@Provider("provider1")
@PactFolder("pacts")
public class ContractTest {

  @TestTarget
  public final Target target = new HttpTarget("localhost", 8332);
}
```

Sets the provider's name

Registers PactRunner

Sets where contract files are stored. In this case, the location resolves to src/test/resources(pacts).

Sets the target to use for tests

Configures the provider location

Notice that there's no test method annotated with `@Test`. This isn't required, because rather than 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`—Retrieves contracts from a project folder or resource folder; for example, `@PactFolder("subfolder/in/resource/directory")`.
- `PactUrl`—Retrieves contracts from URLs; for example, `@PactUrl(urls = {"http://myserver/contract1.json"})`.
- `PactBroker`—Retrieves contracts from Pact Broker; 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 that the provider can send a response that matches what the consumer expects. A typical scenario is setting up a datasource 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 6.11 summarizes the interaction between consumer, states, and provider.

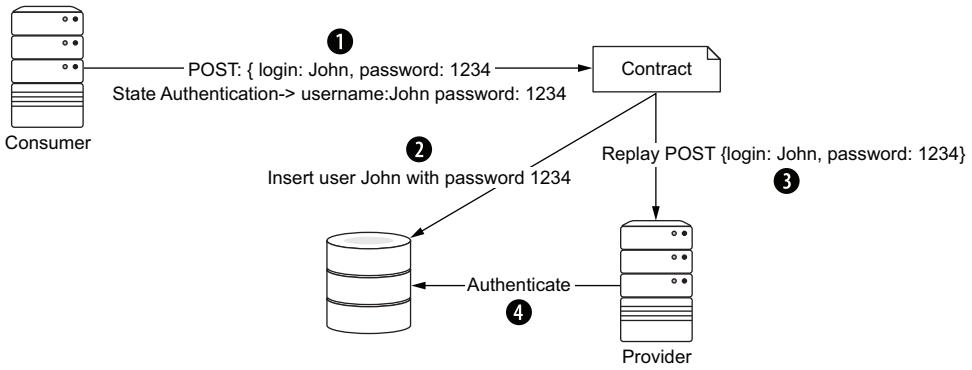


Figure 6.11 Interactions between consumer and provider

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(PactDslWithProvider builder) {
  Map<String, Object> parameters = new HashMap<>();
  parameters.put("login", "John");
  parameters.put("password", "1234")
  builder
    .given("State Authentication", parameters)
    .uponReceiving("")
    ....
```

Defines the data required in the state

Registers the state in the contract with a name and parameters

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> params) {
  //Insert data
}
```

Sets the name of the state to react to

The method receives as a Map the parameters defined by the consumer.

Notice that with states, you can share information between 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 integration 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.

6.2.3 Integrating Pact JVM into the Arquillian ecosystem with Algeron

Arquillian Algeron is an Arquillian extension that integrates Arquillian with contract testing. It provides common ground for integrating Arquillian with contract-testing frameworks.

Arquillian Algeron Pact is the integration of the Arquillian philosophy and extensions into the consumer-driven contracts approach using the Pact JVM core. By using Arquillian Algeron Pact, you get the best of both worlds: you can use the Pact-JVM approach to validate consumers and providers, and you can use Arquillian to run tests in an environment similar to production. Let's see how Arquillian Algeron Pact fits into the consumer and provider sides.

To implement a consumer gateway using a JAX-RS client, the code only uses the API interfaces (the implementation is usually provided by the application server). To run your tests, you'll need to define an implementation of JAX-RS; Apache CXF (http://cxf.apache.org) is a good choice. You can provide an implementation in your build tool, or you can write an Arquillian test. In Arquillian, the test and the business code are deployed and run on the application server you'll use in production with the same JAX-RS implementation and version.

On the provider side, you need to replay the contract against a running provider. You can rely on the build script to package and deploy the provider application, or you can use Arquillian to package and deploy the application in the test, avoiding dependence on a build tool.

> **TIP** When you're validating the provider side, you don't need to run providers with real databases or real external services; you can use in-memory databases or stubs. For this reason, using an Arquillian microdeployment can help you create deployment files that contain configuration files and classes that point to in-memory databases or stubbing implementations.

Arquillian Algeron offers other features in addition to integration between Pact and Arquillian:

- *Publishers*—On the consumer side, you can configure it to publish contracts into a specified repository if they're successfully generated. Currently supported publishers are *folder*, *URL*, *Git server*, and *Pact Broker*, but you can create your own as well.
- *Retrievers*—As with JUnit, you can configure contract loaders. In addition to those already supported in Pact JUnit (*folder*, *URL*, and *Pact Broker*), Arquillian Algeron supports *Git server* and *Maven artifact*.
- *Configuration*—Publishers and retrievers can be configured in an arquillian.xml file so everything is configured in a central place and not in every test class. Note that an annotation-based approach is also supported in retrievers.
- *TestNG*—Because Arquillian is test-framework-agnostic, you can use Pact and TestNG.

Later, we'll go deep into *publishers* and *retrievers* and how to use them.

TIP It isn't mandatory to use Arquillian Algeron Pact on both the consumer and provider sides. You can use Pact JVM or any Pact implementation in any language on either the consumer or provider side, and use Arquillian Algeron Pact on the other side.

As with Pact JVM, Arquillian Algeron Pact is divided into consumer and provider parts.

WARNING Because Arquillian Algeron uses Pact Core and not Pact JUnit, annotations are specific to Arquillian Algeron and aren't the same as in Pact JUnit.

Writing the consumer side with Arquillian Algeron Pact

Following is an example of the consumer part of using Arquillian Algeron Pact, from ClientGatewayTest.java.

Listing 6.6 Arquillian Algeron Pact, consumer side

Arquillian runner

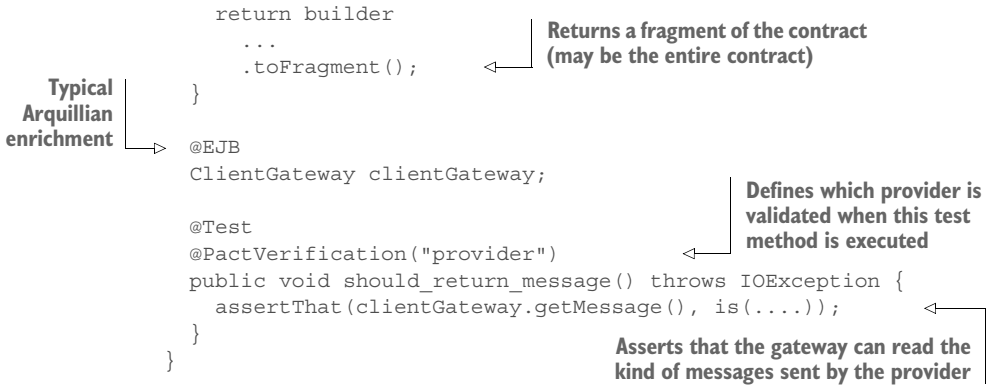
To define the contract, you annotate a method or class with org.arquillian.algeron.pact.consumer.spi.Pact and set the provider and consumer name.

Defines what you want to deploy to the defined container

```
@RunWith(Arquillian.class)
@Pact(provider="provider", consumer="consumer")
public class ClientGatewayTest {

  @Deployment
  public static JavaArchive createDeployment() {
    return ShrinkWrap.create(JavaArchive.class)
      .addClasses(ClientGateway.class);
  }

  public PactFragment createFragment(PactDslWithProvider builder) {
```



Here, you're writing a consumer contract test that defines a contract between `consumer` and `provider`. The client gateway is implemented using a JAX-RS client; it's an EJB, so it knows to run that gateway under the same runtime it will find on production. For this reason, it's a good approach to use Arquillian Algeron instead of Pact JVM alone.

The `@Pact` annotation defines the interaction between consumer and provider. The annotation can be used at the class level, which means all contracts defined in this test are for the same provider; or, it can be used at the method level, which lets you specify that a concrete `PactFragment` defines only the interaction for the consumer-provider tuple defined in the annotation. Thus you could define contracts for different providers in the same consumer class. When the annotation is defined at the method level, it takes precedence over one defined at the class level.

TIP Defining contracts for several providers in the same test isn't something we recommend, because doing so breaks the pattern that one class should test one thing.

Finally, for each test case, you need to specify which provider is validated when that test is run. You do so by using `@PactVerification` and setting the provider name. Notice that setting the provider name isn't mandatory if you're using the `@Pact` annotation at the class level, because the provider name is resolved from there.

The steps executed when you use an Arquillian Algeron Pact consumer test are similar to a standard Arquillian test:

1. The chosen application server is started.
2. The (micro)deployment application file is deployed.
3. The Pact Stub HTTP server is started.
4. All interactions (`PactFragments`) are recorded.
5. Tests are executed.
6. For successful tests, contract files are generated.
7. The application is undeployed.
8. The application server is stopped.

If you're implementing more than one method that returns a `PactFragment` instance, you need to use the `fragment` attribute at `@PactVerification(.. fragment="createFragment")` to specify which fragment method is under test for that `@Test` method.

> **TIP** You can use Arquillian standalone with Arquillian Algeron if you want to skip the deployment step. This is useful if you're using a client gateway that doesn't depend on any features of your runtime.

Writing the provider side with Arquillian Algeron Pact

Now, let's see how to write the provider part using Arquillian Algeron Pact. Because Arquillian Algeron uses Pact JVM, the approach is exactly the same: it replays all requests defined in the contract against the real provider and validates that the response is the expected one.

Listing 6.7 Arquillian Algeron Pact, provider side

```
@RunWith(Arquillian.class)
@Provider("provider")
@ContractsFolder("pacts")
public class MyServiceProviderTest {

  @Deployment(testable = false)
  public static WebArchive createDeployment() {
    return ShrinkWrap.create(WebArchive.class).addClass(MyService.class);
  }

  @ArquillianResource
  URL webapp;

  @ArquillianResource
  Target target;

  @Test
  public void should_provide_valid_answers() {
    target.testInteraction(webapp);
  }

}
```

Arquillian runner

Sets the name of the provider used in the test

Configures where to get the pact contract files

The test must be run as a client; sets testable to false.

URL where the application is deployed

A Target is a class that makes all the requests to the provider. Arquillian Algeron Pact uses an HTTP client target by default.

Executes the interaction against the deployed application, and validates the response

This test validates that the provider meets the expectations defined by the consumer. Because you need to deploy a real provider, this is a good approach, because you can use Arquillian features to package, deploy, and start the application.

The first thing you do in a provider test is specify which provider you're validating. You do so by using `org.arquillian.algeron.pact.provider.spi.Provider` and setting the name of the provider given in the consumer test.

Arquillian Algeron supports two ways to retrieve contracts: by using annotations or by configuring the retriever in arquillian.xml. The latter option will be covered in the

section "Registering publishers and retrievers," later in this chapter. In this test, contracts are retrieved from `pact` using the `org.arquillian.algeron.provider.core.retriever.ContractsFolder` annotation, but other annotations are also supported:

- `org.arquillian.algeron.provider.core.retriever.ContractsUrl`—Retrieves contracts from a URL
- `org.arquillian.algeron.pact.provider.loader.git.ContractsGit`—Retrieves contracts from a Git repository
- `org.arquillian.algeron.pact.provider.loader.maven.ContractsMavenDependency`—Retrieves contracts from Maven artifacts
- `org.arquillian.algeron.pact.provider.core.loader.pactbroker.PactBroker`—Retrieves contracts from a Pact Broker server

It's important to make this test run as a client, which means the test isn't executed in the application server. You can use Arquillian as the deployer; but replaying the interactions must be done from outside the container, as any other consumer would do.

Finally, you need to enrich the test with the URL where the application is deployed and an instance of `org.arquillian.algeron.pact.provider.core.httptarget.Target`, which you use to replay all interactions by calling the `testInteraction` method.

The steps executed in an Arquillian Algeron Pact provider test are similar to a standard Arquillian test:

1. The chosen application server is started.
2. The (micro)deployment application file is deployed.
3. Contracts are retrieved from the given location.
4. For each contract, Arquillian Algeron extracts each request/response pair, sends a request to the provider, and validates the response against the contract response.
5. The application is undeployed.
6. The application server is stopped.

Next, let's examine how you can use a feature provided by Arquillian Algeron that lets you publish contracts automatically and retrieve them in the provider.

REGISTERING PUBLISHERS AND RETRIEVERS

As mentioned earlier, Arquillian Algeron offers the possibility of publishing contracts to a repository and retrieving them for validation. Publishers are configured in an Arquillian configuration file called arquillian.xml. Retrievers can be configured using annotations, as you saw in the previous section, or in arquillian.xml.

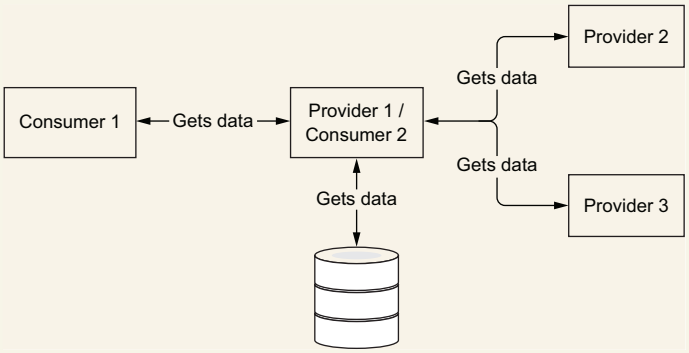
At the time of writing, Arquillian Algeron defines four publishers out of the box:

- *Folders*—Copies contracts to a given folder
- *Url*—Makes a POST request to a given URL, with the contract as the body
- *Git*—Pushes contracts into a Git repository
- *pact-broker*—A specific publisher to store the contract in Pact Broker

Microdeployments

A provider can also be a consumer of other services, or it may have dependencies on datasources. This is inconvenient, because to validate the contract against the provider, you may have to boot up other providers and datasources, which is difficult and sometimes leaves you with flaky tests.

In the following figure, you can see that provider 1 is also a consumer of two services and requires a database.



A provider that's also a consumer

The solution for avoiding dependency problems in tests depends on the kind of dependency. If you're dealing with another provider, you can use the service-virtualization approach using WireMock, as explained in chapter 4. If you're using a datasource, you can use your own stubs at database entry points with the required data, or use an in-memory database, as explained in chapter 5.

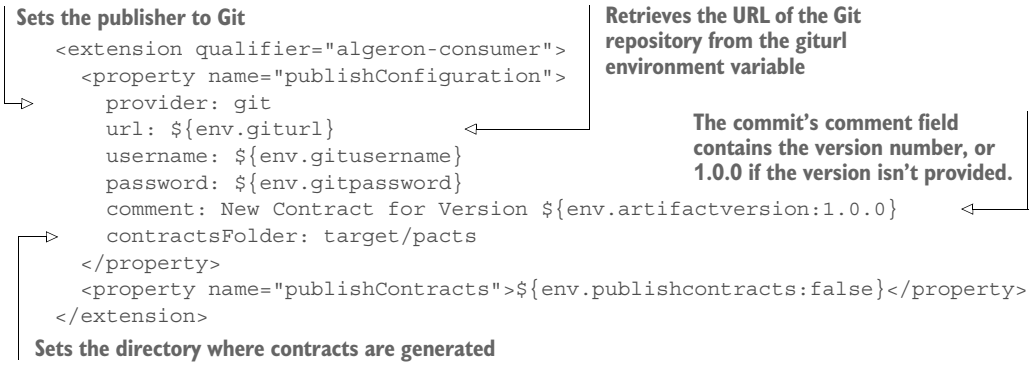
But in all cases, your deployment file will be different than the one you use in production. It contains test-configuration files that point to the service-virtualization instances and configure them to use an in-memory database or packaged alternative classes as stubs. In this context, as you learned in chapter 4, ShrinkWrap and microdeployment are helpful for generating deployment files dynamically in tests.

Arquillian Algeron also provides an SPI so you can implement your own publisher, but this topic is beyond the scope of this book. Refer to the Arquillian Algeron documentation for more information.

It's important to note that a Arquillian Algeron consumer won't publish contracts by default. This is a safety precaution to avoid publishing contracts every time consumer tests are run locally. To modify this behavior, you need to set the `publishContracts` configuration attribute to `true`. You should only do that if you're publishing a new version of a consumer, and this action should be performed by your continuous (CI/CD) environment.

TIP You can configure arquillian.xml attributes with system properties or environment variables by using a `${system_property}` placeholder or a `${env.environment_variable}` placeholder. You can add a default value by following the variable name with a colon (`:`) and the value.

Here's an example of how to configure a Git publisher in arquillian.xml:




```
<extension qualifier="algeron-consumer">
  <property name="publishConfiguration">
    provider: git
    url: ${env.giturl}
    username: ${env.gitusername}
    password: ${env.gitpassword}
    comment: New Contract for Version ${env.artifactversion:1.0.0}
    contractsFolder: target/pacts
  </property>
  <property name="publishContracts">${env.publishcontracts:false}</property>
</extension>
```

This snippet configures the publisher to push generated contracts to a Git repository. The publishing process is executed only when the environment variable `publishcontracts` is set to `true`; otherwise, contracts are generated in a local directory but not published.

Next, we'll look at how to configure retrievers in arquillian.xml. Registering a retriever in arquillian.xml uses the same approach as registering a publisher. The same retrievers mentioned in section 6.2.3, which can be used as annotations, are supported here.

Here's how to configure a retriever to get contracts from a Git repository:

```
<extension qualifier="algeron-provider">
  <property name="retrieverConfiguration">
    provider: git
    url: ${env.giturl}
    username: ${env.gitusername}
    password: ${env.gitpassword}
  </property>
</extension>
```

The format is similar to that for providers. Obviously, the property name and extension name are different, because retrievers are an important part of the provider side.

After this thorough introduction to contract testing, let's explore how to apply them to the book's example.

6.3 Build-script modifications

As you now know, contract tests are divided between the consumer side and the provider side. Each has its own dependencies. Let's look at each of these dependency cases, when using either Pact JVM or Arquillian Algeron.

6.3.1 Using Pact JVM for contract testing

If you're using Pact JVM for consumer-driven contracts, you need to add dependencies. For the consumer part, add the following dependencies:

```
dependencies {
  testCompile group: 'au.com.dius',
              name: 'pact-jvm-consumer-junit_2.11',
              version: '3.5.0'
}
```

And for the provider part, add these dependencies:

```
dependencies {
  testCompile group: 'au.com.dius',
              name: 'pact-jvm-provider-junit_2.11',
              version: '3.5.0'
}
```

6.3.2 Using Arquillian Algeron for contract testing

To use Arquillian Algeron with Pact JVM, you need to use at least two dependencies: Arquillian Algeron Pact, and Pact itself.

For the consumer part, add these dependencies:

```
dependencies {
  testCompile group: 'org.arquillian.algeron',
              name: 'arquillian-algeron-pact-consumer-core',
              version: '1.0.1'
  testCompile group: 'au.com.dius',
              name: 'pact-jvm-consumer_2.11',
              version: '3.5.0'
}
```

If you're using the Git publisher, you also need to add this dependency:

```
dependencies {
  testCompile group: 'org.arquillian.algeron',
              name: 'arquillian-algeron-consumer-git-publisher',
              version: '1.0.1'
}
```

For the provider part, add the following dependencies:

```
dependencies {
  testCompile group: 'org.arquillian.algeron',
              name: 'arquillian-algeron-pact-provider-core',
              version: '1.0.1'
  testCompile group: 'au.com.dius',
              name: 'pact-jvm-provider_2.11',
              version: '3.5.0'
}
```

Also add this, if you want to integrate with AssertJ:

```
dependencies {
  testCompile group: 'org.arquillian.algeron',
              name: 'arquillian-algeron-pact-provider-assertj',
              version: '1.0.1'
  testCompile group: 'org.assertj',
              name: 'assertj-core',
              version: '3.8.0'
}
```

If you're using the Git retriever, add this dependency:

```
dependencies {
  testCompile group: 'org.arquillian.algeron',
              name: 'arquillian-algeron-provider-git-retriever',
              version: '1.0.1'
}
```

If you're using the Maven retriever, add this dependency:

```
dependencies {
  testCompile group: 'org.arquillian.algeron',
              name: 'arquillian-algeron-provider-maven-retriever',
              version: '1.0.1'
}
```

And if you're using the Pact Broker retriever, you also need to add this dependency:

```
dependencies {
  testCompile group: 'org.arquillian.algeron',
              name: 'arquillian-algeron-pact-provider-pact-broker-loader',
              version: '1.0.1'
}
```

After you've registered the dependencies in the build script, you can start writing contract tests.

6.4 Writing consumer-driven contracts for the Gamer application

Let's write a contract test for the unique consumer that's provided in the current application: the aggregator service. We'll also show you the provider side, which validates the given contract. In this case, tests are created in a new module/subproject in the main project called, for example, c-tests.

6.4.1 Consumer side of the comments service

The aggregator service communicates with services such as game and comments, so it's effectively a consumer of all of them. Let's examine how to write the contract between the aggregator service and the comments service. The class responsible for communicating with the comments service is `book.aggr.CommentsGateway`. This is a

simple `boundary` class that acts as a gateway between the aggregator service and the comments service. You'll use Arquillian Algeron, to take advantage of its publishing capabilities.

First, here's the contract for storing comments (code/aggregator/c-tests/src/test/java/book/aggr/CommentsContractTest.java).

Listing 6.8 Storing comments

```
@RunWith(Arquillian.class)
@Pact(provider = "comments_service", consumer =
        "games_aggregator_service")
public class CommentsContractTest {


    private static final String commentObject = "{" + "  'comment' " +
            ": 'This Game is Awesome'," + "  'rate' : 5," + "  " +
            "'gameId': 1234" + "}";

    private static final String commentResult = "{" + "   'rate': " +
            "5.0," + "   'total': 1," + "   'comments': ['This Game" +
            " is Awesome']" + "}";

    public PactFragment putCommentFragment(PactDslWithProvider
                                                    builder) {
        final Map<String, String> headers = new HashMap<>();
        headers.put("Content-Type", "application/json");

        return builder.uponReceiving("User creates a new comment")
                .path("/comments").method("POST").headers(headers)
                .body(toJson(commentObject))
                .willRespondWith().status(201).matchHeader
                        ("Location", ".*/[0-9a-f]+",
                                "/comments/1234").toFragment();
    }

    @Test
    @PactVerification(fragment = "putCommentFragment") //
    public void shouldInsertCommentsInCommentsService() throws
            ExecutionException, InterruptedException {

        final CommentsGateway commentsGateway = new CommentsGateway();
        commentsGateway.initRestClient(url.toString()); //
        //

        JsonReader jsonReader = Json.createReader(new StringReader
                (toJson(commentObject)));
        JsonObject commentObject = jsonReader.readObject();
        jsonReader.close();

        final Future comment = commentsGateway
                .createComment(commentObject);
```

Arquillian test runner

Sets the Pact annotation with the consumer and provider names

Creates the PactFragment that defines the contract for posting comments

Sets the body message

Configures the method as a contact validator of the fragment defined in putCommentFragment

Connects to the HTTP stub server

```
        final Response response = comment.get();
        final URI location = response.getLocation();

        assertThat(location).isNotNull();
        final String id = extractId(location);

        assertThat(id).matches("[0-9a-f]+");
        assertThat(response.getStatus()).isEqualTo(201);

    }
```

This test uses Arquillian standalone, because there's no `@Deployment` method. At this point, you don't need to deploy anything to the container. The contract for sending a comment to the comments service is defined in `putCommentFragment`, which defines the contract with the expected body and the canned response. Finally, there are the assertions for validating that the `CommentsGateway` class works as expected.

Now let's write the contract for getting comments for a given `gameId` (code/aggregator/c-tests/src/test/java/book/aggr/CommentsContractTest.java). In this case, you need to set a state for telling the provider which data you expect to be returned when the contract is validated against it.

Listing 6.9 Getting comments for a game

```
@StubServer
URL url;

public PactFragment getCommentsFragment(PactDslWithProvider
                                                builder) {

    final Map<String, String> headers = new HashMap<>();
    headers.put("Content-Type", "application/json");

    return builder.given("A game with id 12 with rate 5 and " +
            "message This Game is Awesome")
            .uponReceiving("User gets comments for given Game")
            .matchPath("/comments/12").method("GET")
            .willRespondWith().status(200).headers(headers)
            .body(toJson(commentResult)).toFragment();

}

@Test
@PactVerification(fragment = "getCommentsFragment")
public void shouldGetComentsFromCommentsService() throws
        ExecutionException, InterruptedException {
    final CommentsGateway commentsGateway = new CommentsGateway();
    commentsGateway.initRestClient(url.toString());

    final Future<JsonObject> comments = commentsGateway
            .getCommentsFromCommentsService(12);
    final JsonObject commentsResponse = comments.get();
```

Sets state information

```
    assertThat(commentsResponse.getJsonNumber("rate")
            .doubleValue()).isEqualTo(5);
    assertThat(commentsResponse.getInt("total")).isEqualTo(1);
    assertThat(commentsResponse.getJsonArray("comments"))
            .hasSize(1);

}
```

Asserts responses

Notice that the definition of the contract is similar to the previous one. The biggest difference is the use of the `given` method to set a state. In this case, you're setting the data that will be required on the provider side.

Finally, you need to configure Arquillian Algeron to publish contracts in a shared place, so the provider can retrieve and validate them (code/aggregator/c-tests/src/test/resources/arquillian.xml). For the sake of simplicity, the folder approach is used here, but in the real world you'd probably use a Git repository.

Listing 6.10 Publishing contracts in a shared location

```
<?xml version="1.0"?>
<arquillian
xsi:schemaLocation="http://jboss.org/schema/arquillian
http://jboss.org/schema/arquillian/arquillian_1_0.xsd"
xmlns="http://jboss.org/schema/arquillian"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    <extension qualifier="algeron-consumer">
        <property name="publishConfiguration"> provider: folder
          outputFolder: /tmp/mypacts
          contractsFolder: target/pacts
        </property>
        <property name="publishContracts">
          ${env.publishcontracts:true}
        </property>
    </extension>
</arquillian>
```

Now that you've written the contract for the consumer side, let's see what you need to do to validate it against the provider.

6.4.2 Provider side of the comments service

To validate contracts generated on the consumer side, you need to create a test on the provider project that downloads contracts and replays them against a running instance of the provider (code/comments/c-tests/src/test/java/book/comments/boundary/CommentsProviderTest.java). You need to deploy the real comments service, so using Arquillian Algeron is a good choice: it takes care of creating the deployment file and deploying it to the application server. Contracts are stored in the same folder defined in the `publishConfiguration` property, as discussed in the previous section.

Listing 6.11 Testing the comments service on the provider side

```
@RunWith(Arquillian.class)
@ContractsFolder(value = "/tmp/mypacts")
@Provider("comments_service")
public class CommentsProviderTest {

    static {
        System.setProperty("MONGO_HOME",
                "/mongodb-osx-x86_64-3.2.7");
    }

    @ClassRule
    public static ManagedMongoDb managedMongoDb =
            newManagedMongoDbRule().build();

    @Rule
    public MongoDbRule remoteMongoDbRule = new MongoDbRule(mongoDb
            ().databaseName("test").host("localhost").build());

    @Deployment(testable = false)
    public static WebArchive createDeployment() {
        final WebArchive webArchive = ShrinkWrap.create(WebArchive
                .class).addPackage(CommentsResource.class
                .getPackage()).addClass(MongoClientProvider.class)
                .addAsWebInfResource("test-resources.xml",
                        "resources.xml").addAsWebInfResource
                        (EmptyAsset.INSTANCE, "beans.xml")
                .addAsLibraries(Maven.resolver().resolve("org" +
                        ".mongodb:mongodb-driver:3.2.2")
                        .withTransitivity().as(JavaArchive.class));

        return webArchive;
    }

    private static final String commentObject = "{" + "  'comment' " +
            ": '%s'," + "  'rate' : %d," + "  'gameId': %d" + "}";

    @State("A game with id (\\d+) with rate (\\d+) and message (.+)")
    public void insertGame(int gameId, int rate, String message)
            throws MalformedURLException {

        RestAssured.given().body(toJson(String.format
                (commentObject, message, rate, gameId)))
                .contentType(ContentType.JSON).post(new URL
                (commentsService, "comments")).then().statusCode(201);

    }

    @ArquillianResource
    URL commentsService;

    @ArquillianResource
    Target target;
```

Sets the contract location

Configures the provider

Sets the MongoDB home directory

Uses NoSQLUnit managed MongoDB

Configures a MongoDB remote connection

Ensures that the test runs in client mode

Populates the required data to the provider from the contract definition

Injects the Target class for replaying the contract

```
@Test
@UsingDataSet(loadStrategy = LoadStrategyEnum.DELETE_ALL)
public void should_provide_valid_answers() {
    PactProviderAssertions.assertThat(target).withUrl
            (commentsService).satisfiesContract();
}
```

Cleans the database after each run

WARNING Be sure to adapt the contract location and MongoDB home to your environment before running the test.

This test prepares the environment by starting the MongoDB database and Apache TomEE. Then it deploys the application and replays the contract against the configured environment. Note the following three important things:

- You use NoSQLUnit to prepare the MongoDB environment. NoSQLUnit can be used in integration tests, as you saw in chapter 5, and also in any other kind of test.
- The state method `insertGame` is only used by the part of the contract that defined the state on the consumer side. This is the part of the contract that validates receiving comments from the service.
- In the state method, the test uses the POST method to populate the data, so you're effectively using the comments service endpoint to insert data into the database. You use the `RestAssured` test framework for this purpose.

Figure 6.12 summarizes the lifecycle when you run this test. First, the Arquillian test uses NoSQLUnit to start an instance of MongoDB installed on the system property/environment variable `MONGO_HOME`. Then, it starts an instance of Apache TomEE and deploys the comments service inside it. If a contract defines a state following the form `A game with id (\d+) with rate (\d+) and message (.+)`, some data is populated in MongoDB using the comments service. Finally, the test replays each of the contracts, cleaning the database before each execution.

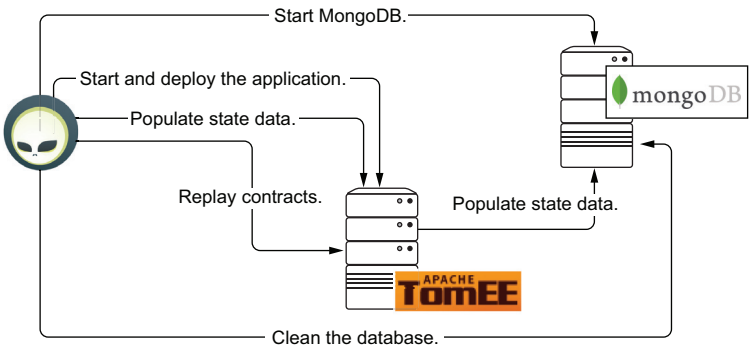


Figure 6.12 The test lifecycle

6.5 Contract type summary

The following table summarizes the types of contracts we've discussed.

Table 6.1 Consumer, provider, and consumer-driven contracts

Contract	Complete	Number	Bounded
Provider	Yes	Single	Space/Time
Consumer	No	Multiple	Space/Time
Consumer-driven	Yes	Single	Consumers

The provider- and consumer-driven approaches are *complete*: they provide a complete set of functionalities. With the consumer approach, contracts are incomplete from the point of view of the functionalities available to the system. In addition, the provider- and consumer-driven approaches are *singular* in their expression of business functionality; but with the consumer approach, each consumer has its own contract.

Exercise

You should now be able to write any pair of consumer/provider tests. We recommend that you try to define the consumer side of the game service, and then the provider side.

> **TIP** Take a look at chapter 4, which introduced writing tests for WildFly Swarm using Arquillian.

Summary

- 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 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 only needs to know 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 (like those described in section 6.1.5), you may want to use provider-driven contracts or consumer contracts.

Chapter 8 from *Testing Java Microservices*
by Alex Soto Bueno, Andy Gumbrecht
and Jason Porter

Docker and testing

This chapter covers

- Handling difficulties with high-level tests
- Understanding how Docker can help you with testing
- Creating reproducible testing environments
- Working with Arquillian Cube

A recurring message delivered by this book, and a message most developers would agree with, is that a high-level test implies a slower test—and that a huge amount of effort is usually required to prepare the test environment in which to run it. You encountered this in chapter 5, where you learned that to write an integration test against a database, you need an environment with the database you're going to use in production. The same thing happened in chapter 7, where you saw that you might need one or more microservices deployed along with their databases. And in the case of a web frontend, you might also need specific browsers installed.

When you want to test the application's big picture, you need more pieces in your environment to make it run. Here are a few:

- Databases
- Other microservices
- Multiple browsers
- Distributed caches

The problem is that first, you need to define your test environment; and second, you must provide a runtime under which to run this configuration. This probably won't be your development machine, a disadvantage for a developer because you'll lose the possibility of reproducing or debugging any given problem locally. Remote debugging, although not impossible, is much more complex.

Using Docker can mitigate some of these issues. Docker lets developers run test environments on their local machine.

8.1 Tools in the Docker ecosystem

The Docker ecosystem includes several tools such as Docker Compose, Docker Machine, and libraries that integrate with the ecosystem, such as Arquillian Cube. This section presents an overview of some of them.

8.1.1 Docker

Docker is an open source platform that simplifies the creation, deployment, and running of applications in *containers*. Imagine a container as being like a box where you deploy an application along with all the dependencies it needs, neatly packaged. In packaging the application this way, you can ensure that the application will run in a well-defined environment (one provided by the container), regardless of the underlying operating system. This enables you to move containers from one machine to another—for example, from a developer machine to a production machine—without worrying about the external configuration.

You can think of Docker as a virtual machine, but without the need to install and set up the entire OS yourself. Docker reuses the same Linux kernel regardless of where the container is actually running. This approach gives your startup time a significant performance boost and reduces the size of the application.

Figure 8.1 shows a comparison between a VM and a Docker container. The primary difference is that instead of having a guest OS for each application, a container runs atop a machine OS.

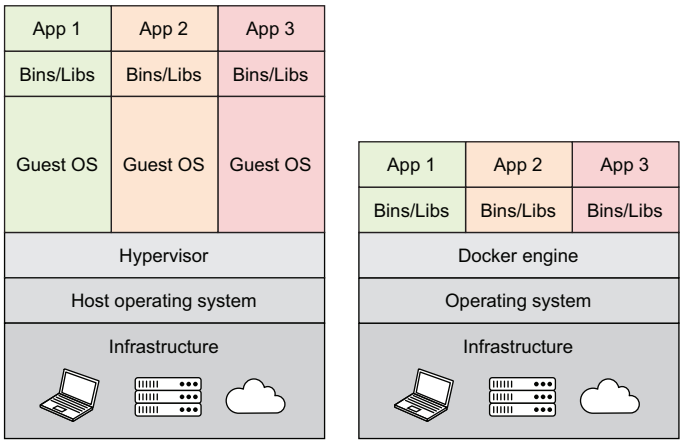


Figure 8.1 Virtual machine vs. container

To run Docker containers, you need at least three components:

- *Docker client*—A command-line interface (CLI) program that sends commands from the user to a host where the Docker daemon is installed/running.
- *Docker daemon*—A program that runs on the host OS and performs all major operations such as building, shipping, and running Docker containers. During the development and testing phases, the Docker daemon and the Docker client are probably running in the same machine.
- *Docker registry*—An artifact repository for sharing Docker images. There's a public Docker registry at http://hub.docker.com.

NOTE The difference between an image and a container is that an *image* is all the bits and bobs, such as the application and configuration parameters. It doesn't have state and never changes. On the other hand, a *container* is a running instance of an image on the Docker daemon.

Figure 8.2 shows the schema of the Docker architecture. The Docker client communicates with the Docker daemon/host to execute commands. If a required Docker image isn't present on the host, it's downloaded from the Docker registry. Finally, the Docker host instantiates a new container from a given image.

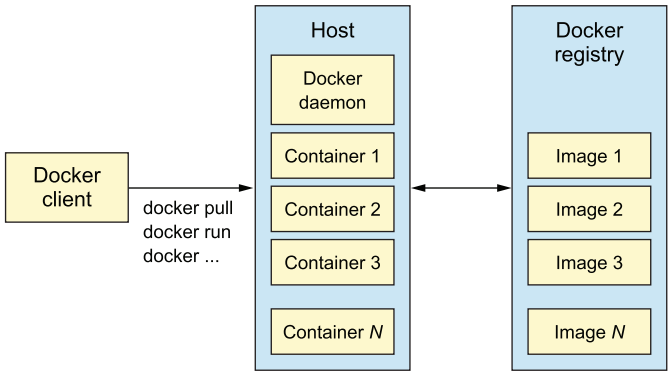


Figure 8.2 Docker architectural schema

Using the Docker client to retrieve an image from the Docker registry and running it on the Docker host might look like this:

```
docker pull jboss/wildfly
docker run -ti -p 8080:8080 jboss/wildfly
```

Pulls (downloads) the image from the Docker registry to the Docker host

Starts the JBoss/WildFly container on the Docker host

After executing the preceding commands, you can navigate in your browser to http://<dockerHostIp>;:8080, where you should see the WildFly server Welcome page shown in figure 8.3.

Figure 8.3 Welcome to WildFly

Now that you understand the Docker essentials, let's look at a Docker tool called *Docker Machine*.

8.1.2 Docker Machine

Docker Machine helps you to create Docker hosts on virtualization platforms like VirtualBox and VMware. It also supports most popular infrastructure as a service (IaaS) platforms, such as Amazon Web Services (AWS), Azure, DigitalOcean, OpenStack, and Google Compute Engine. In addition to installing Docker on given hosts, it also configures a Docker client to communicate with them.

Usually, the virtualization approach with VirtualBox is the best way to go on development machines. Note that in this case you need to have VirtualBox installed, because Docker Machine doesn't perform this installation step.

To create a VirtualBox image with Docker installed, run the following command:

```
docker-machine create --driver virtualbox dev
```

After some time, everything will be installed and ready to be used on your local machine. The final steps are to begin creating the VirtualBox image and configure the client to point to the instance:

```
docker-machine start dev
eval $(docker-machine env dev)
```

Starts the host named dev

Configures the Docker client environment variables

Docker on a local machine

If you want to use Docker Machine on a local machine, the best way to get started is to use Docker Toolbox. It's available for both Windows and macOS.

Docker Machine has an installer that installs the following tools:

- Docker client
- Docker Machine
- Docker Compose
- Oracle VirtualBox
- Kitematic (Docker GUI) and a preconfigured shell

8.1.3 Docker Compose

Docker Compose is a multi-container-management tool composed of two elements:

- A format in YAML describing one or more containers that work together to form the application. It also specifies how containers interact with each other, as well as some other information such as networks, volumes, and ports.
- A CLI tool to read a Docker Compose file and create the application defined by the file.

By default, any Docker Compose file is appropriately named docker-compose.yml. A simple docker-compose.yml file might look like the following.

Listing 8.1 docker-compose.yml

When you run `docker-compose up` from the terminal on the file in listing 8.1, two containers are started: *tomcat* and *pingpong*. They're connected with a `link` named *pingpong*. And port 8081 for the Tomcat service is exposed to clients of the Docker host, forwarding all traffic to the internal container port 8080. The containers are reachable at a hostname identical to the alias or service name, if no alias is specified.

One of the best testing-related aspects of Docker Compose is the extension feature. It enables you to share common configuration snippets between different files.

Let's look at an example of a Docker Compose file using the `extend` keyword. You should first define the common or abstract container definition that other containers

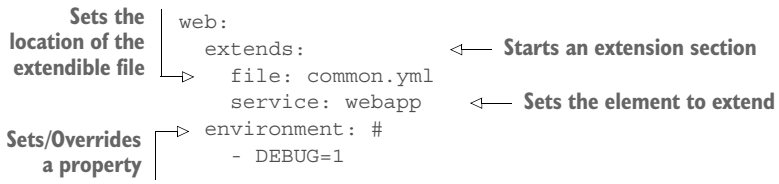
may extend in a Docker Compose file, which, by default, should *not* be named docker-compose.yml. You should strive to follow the naming convention that the file called docker-compose.yml is the *only* file that will be used by the running system, so to that end let's name this file common.yml.

Listing 8.2 common.yml

```
webapp:
  image: myorganization/myservice
  ports:
    - "8080:8080"
  volumes:
    - "/data"
```

There's nothing new here beyond the previous definition, other than defining a new service called `webapp`. Now, let's define the docker-compose.yml file to use common.yml, and set new parameters for the service.

Listing 8.3 docker-compose.yml



You can run `docker-compose` in a terminal:

```
docker-compose -f docker-compose.test.yml up
```

As you can see, using Docker and Docker Compose, you can readily define testing environments that can be used on any machine where tests need to run. The two main advantages of using these tools together are as follows:

- Every environment that runs tests contains exactly the same versions of the required libraries, dependencies, and/or servers. It doesn't matter which physical machines are run: it could be the development, testing, or preproduction machine. All of them contain the same running bits and bobs.
- You don't need to install anything on testing machines other than Docker. Everything else is resolved at runtime.

NOTE This chapter provides a very basic introduction to Docker Compose, and we encourage you to learn more at https://docs.docker.com/compose.

8.2 Arquillian Cube

So far, you've read that Docker and Docker Compose make a perfect pair for testing. They should help you define reliable and reproducible testing environments, so that each time you execute tests, you know the same environment is set up correctly.

The problem is that you need to manually run `docker-compose up` or add a step to the build tool in order to fire it up before executing the tests. You also need to deal with the Docker host IP address, which may be `localhost`—but not necessarily, because it may be a Docker machine or a remote Docker host.

Although starting Docker Compose manually could be a good approach, our opinion is that tests should be self-executing as much as possible and not require manual intervention or complicated runtimes. You've guessed it: there's a cool Arquillian extension that will help you achieve your goals. *Arquillian Cube* is an extension that can be used to manage Docker containers from within an Arquillian test. It uses an approach similar to what Arquillian Core does for application servers, but modified for Docker containers. Arquillian Cube can be used in the following scenarios:

- Preparing testing environments for high-level tests
- Testing Dockerfile compositions
- Validating Docker Compose compositions
- White box and black box testing

As you can see in figure 8.4, before executing tests, Arquillian Cube reads a Docker Compose file and starts all the containers in the correct order. Arquillian then waits until all the services are up and running so they're able to receive incoming connections. After that, tests are executed with the test environment running. Following execution, all running containers are stopped and removed from the Docker host.

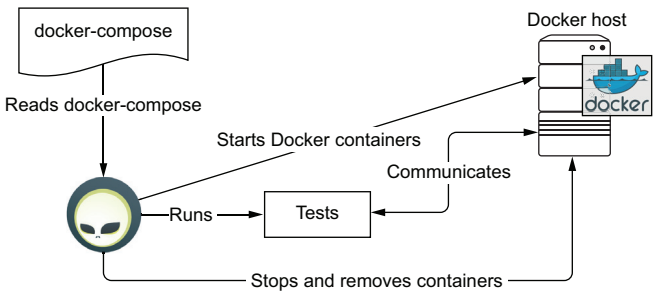


Figure 8.4 Arquillian Cube lifecycle

With Arquillian Cube, your Docker-based tests are fully automated. You can define your testing environment in the Docker Compose format, and the Arquillian runner takes care of everything for you. This again leaves you, the developer, free to write the actual tests.

NOTE It isn't necessary to use Docker for production. You can also just take advantage of Docker to prepare a valid testing environment.

When using Docker in production, you can use Arquillian Cube to write tests that validate that your container image is created correctly or, for example, that the container is bootable and is accessible from outside of the Docker host. Obviously, in production, you can use a Docker Compose file that's almost, if not the same as, the one used to create the testing environment, and validate that all containers defined can communicate between each other and that environment variables are correctly set.

The potential use is unbounded, as there are an endless number of ways to use Arquillian Cube and Docker together for testing. We'll cover the most common use cases in the following sections.

8.2.1 Setting up Arquillian Cube

Arquillian Cube requires that you set up several parameters. It uses some default parameters that *may* work in most situations and others that are intelligently deduced from the environment. Sometimes you may need to modify these automatically defined parameters. Table 8.1 describes the most important configuration attributes that can be set in an arquillian.xml file.

Table 8.1 Arquillian Cube parameters

Attribute	Description	Default behavior
`serverUri`	URI of the Docker host where containers will be instantiated.	Gets the value from the environment variable `DOCKER_HOST` if set; otherwise, for Linux it's set to `unix:///var/run/docker.sock`, and on Windows and macOS it's set to `https://<docker_host_ip>;:2376`. `docker_host_ip` is resolved automatically by Arquillian Cube by obtaining the boot2docker or docker machine IP.
`dockerRegistry`	Sets the location of the Docker registry from which to download images.	By default, this is the public Docker registry at https://registry.hub.docker.com.
`username`	Sets the username to connect to the Docker registry. (You'll need an account.)	
`password`	Sets the password to connect to the Docker registry. (You'll need an account.)	
`dockerContainers`	Embeds the Docker Compose content as an Arquillian property, instead of as a Docker Compose file.	
`dockerContainersFile`	Sets the location of the Docker Compose file. The location is relative to the root of the project; but it can also be a URI that's converted to a URL, so you can effectively have Docker Compose definitions on remote sites.	

Table 8.1 Arquillian Cube parameters *(continued)*

Attribute	Description	Default behavior
dockerContainersFiles	Sets a comma-separated list of Docker Compose file locations. Internally, all of these locations are appended into one.	
tlsVerify	Boolean to set if Arquillian Cube should connect to the Docker server with Transport Layer Security (TLS).	Gets the value from the TLS_VERIFY environment variable if set; otherwise, it's automatically sets to false if the serverUri scheme is http or true if it's https. You can force a value by setting this property.
certPath	Path where certificates are stored if you're using HTTPS.	Gets the value from the DOCKER_CERT_PATH environment variable if set; otherwise, the location is resolved from boot2docker or docker-machine. You can force a value by setting this property.
machineName	Sets the machine name if you're using Docker Machine to manage your Docker host.	Gets the value from the DOCKER_MACHINE_NAME environment variable if set; otherwise, the machine name is resolved automatically if in the current Docker machine instance only one machine is running. You can force a value by setting this property.

TIP Remember that the arquillian.xml configuration attributes can be configured using system properties or environment variables by using ${system_property} placeholders or ${env.environment_variable} placeholders.

Arquillian Cube connection modes

The test environment is started and stopped for each test suite. This means that, depending on the elements' boot-up time, testing time may be affected, especially with small test suites.

With Arquillian Cube, you have the option to bypass the creation/start of Docker containers that are already running on a Docker host with the same container name. This allows you to prestart the containers (for example, in the continuous integration [CI] build script, or before starting work) and connect to them to avoid the extra cost during test execution.

Here's an example of how to configure the connectionMode property:

```
<extension qualifier="cube">;
  <property name="connectionMode">;STARTORCONNECT</property>;
</extension>;
```

You can set the following modes for this property:

- STARTANDSTOP—The default, if not specified. Creates and stops all Docker containers.
- STARTORCONNECT—Bypasses the creation/start steps if a container with the same container name is already running and this named container isn't to be terminated after the tests complete. If the container configured for Cube is not already running, then Arquillian will start it *and stop it* at the end of the execution, behaving much like the STARTANDSTOP mode.
- STARTORCONNECTANDLEAVE—Exactly the same as STARTORCONNECT mode; but if a container is started by Arquillian Cube, then it *won't be stopped* at the end of the execution, so it can be reused in the next cycle.

Now that you're familiar with the common configuration parameters in Arquillian Cube, let's explore how to write tests using it.

8.2.2 Writing container tests

The first use case we'll cover for Arquillian Cube is validating that the Dockerfile defined in the service to containerize the application is correct. Although you can perform several checks, the most common ones are as follows:

- Docker is able to build the image without any errors.
- The service exposes the correct ports.
- The service is started correctly and can correctly serve incoming requests.

Let's start by configuring Docker in arquillian.xml, and then create a minimal script to build and run the image under test.

Listing 8.4 Configuring Docker

```
<?xml version="1.0"?>;
<arquillian xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns="http://jboss.org/schema/arquillian"
  xsi:schemaLocation="http://jboss.org/schema/arquillian
  http://jboss.org/schema/arquillian/arquillian_1_0.xsd">;

  <extension qualifier="docker">;
    <property name="machineName">;dev</property>;
    <property name="dockerContainers">;
      myservice:
        build: ./docker
        ports:
          - "8080:8080"
    </property>;
  </extension>;

</arquillian>;
```

This line is required only if you're using Docker Machine.

Defines in Docker Compose format how to build and run the image

When you're using Docker Machine with more than one machine running, the machine named `dev` is used for building and running the Docker container under test. Then, with the `dockerContainers` property, you embed a Docker Compose container definition to build an image from an expected Dockerfile located in the docker directory and exposing port 8080. As mentioned earlier, Dockerfile is the default filename for a Docker definition. The Dockerfile may look like the following:

```
FROM tomee:8-jdk-7.0.2-webprofile

ADD build/libs/myservice.war /usr/local/tomee/webapps/myservice.war
EXPOSE 8080
```

The image defined here is based on the Apache TomEE `tomee:8-jdk-7.0.1-webprofile` Docker image. The `ADD` command adds your project-deployment WAR file to the specified image path—the TomEE hot deployment path in this image is /usr/local/tomee/ webapps/, so adding a WAR file here will automatically deploy the application. Finally, the `EXPOSE` command exposes the TomEE HTTP port 8080 to the outside world.

Using this information, you can write a test to validate that the image is correctly built, exposes the correct port, and runs correctly.

Listing 8.5 Validating the image

```
@RunWith(Arquillian.class)                          Arquillian runner
public class DockerImageCreationTest {

  @ArquillianResource                               The test is enriched with a Docker
  private DockerClient docker;                      client to access the Docker host.

  @HostIp                                           The test is enriched with
  private String dockerHost;                        the Docker host IP.

  @HostPort(containerName = "myservice", value = 8080)   Gets the binding port
  private int myservicePort;                             for exposed port 8080
                                                         of the container
  @Test                                                  myservice
  public void should_expose_correct_port() throws Exception {
    assertThat(docker)
      .container("myservice")                       Asserts that the built image
      .hasExposedPorts("8080/tcp");                 is exposing port 8080
  }

  @Test
  public void should_be_able_to_connect_to_my_service() throws Exception {
    assertThat(docker)
      .container("myservice")                       Asserts that the
      .isRunning();                                 container is running

    final URL healthCheckURL = new URL("http", dockerHost,
➥ myservicePort, "health");
```

```
    String healthCheck = getHealthCheckResult(healthCheckURL);
    assertThat(healthCheck).isEqualTo("OK");
  }

}
```

Asserts that the healthCheck endpoint returns that the service is up and running

There are several things to note about this test. First, you apply the Arquillian runner, but without a `@Deployment` method. This is because these tests don't need to deploy anything in an application server: the container image receives the deployment file required to run the test, and the server is already running. You're effectively using all the elements provided by Arquillian but without deploying anything.

> **NOTE** Any test without a `@Deployment` annotated method must be used with either the `arquillian-junit-standalone` or `arquillian-testng-standalone` dependency, instead of the `container` dependency. All tests are run in the as-client mode, because they can't be deployed into an application server.

The second thing to note is that Arquillian Cube offers some enrichers for tests. In this test, the `DockerClient` object is injected. This object offers you some powerful operations to communicate with the Docker host and get information about running containers. Moreover, the test is enriched with the Docker host IP or hostname in the `dockerHost` variable. The binding port for the container's exposed port 8080 is also injected as the `myservicePort` variable. These variables provide information that allows the test to communicate with the TomEE server and the hosted application.

Last but not least are the test methods to verify that the Dockerfile is correctly configured, the build is correct, and the service it's exposing is correctly deployed. Arquillian Cube provides custom AssertJ assertions. So, for example, you can write assertions to assert that a specific Docker image is instantiated in the Docker host, or a port is exposed, or a specific process is running as expected.

If the construction defined by the Dockerfile fails, Arquillian Cube throws an exception, causing the test to fail. A health check of the endpoint for the deployed service is used to verify that the microservice deployed in the Docker container is operating correctly.

After test execution, Arquillian Cube removes the built image from the Docker host. This ensures that disk space doesn't increase every time you run a test, and also makes sure each test is run in isolation from the next.

Next, let's examine how you can use Arquillian Cube to test more-complex scenarios like integration tests.

8.2.3 Writing integration tests

You learned in chapter 5 that it's possible to validate the connection between two systems, such as the communication between a microservice and a database (such as SQL or NoSQL), or between two microservices. In such cases, it's normal to test your gateway code against any real system that you're going to use in production. This is a big difference when compared to *component tests*, where stubs or fakes are usually used.

The single biggest challenge for integration tests is how to consistently set up the environment to run these tests. For example, you'll probably need the same databases you're using in production on both the developer and CI machines. You may also need a way to deploy all dependent microservices for the actual microservice under test. In addition, ensuring that versions are maintained across all environments and machines isn't a trivial task. Prior to Docker, this kind of setup wasn't easy to realize without having everything in place on all machines.

You've seen that Docker and Docker Compose can help you prepare a consistent environment for testing, and how Arquillian Cube can help automate the process. In this section, we'll look at an example.

Arquillian deployment and Docker

As we said in chapter 4, Arquillian has three ways to manage an application server:

- *Embedded*—The application server shares the same JVM and classpath with the test runtime (IDE, build tool, and so on).
- *Managed*—The application server is booted up independently of the test runtime. It effectively creates a new JVM, independent of the actual test JVM.
- *Remote*—Arquillian doesn't manage the lifecycle of the application server. It expects to reuse an instance that's already up and running.

With this in mind, you can use Arquillian to deploy your (micro)deployment file in an application server that it's running in a Docker container. From an Arquillian point of view, this application server instance is a remote instance where the lifecycle is managed by another party (in this case, Docker).

Think back to the fact that the runtime adapter in the classpath is how Arquillian knows how to manage the application-server lifecycle. For example, in the case of Apache Tomcat, for remote mode you need to define the `org.jboss.arquillian.container:arquillian-tomcat-remote-7:1.0.0.CR7` dependency.

As you can see, it's possible to take advantage of (micro)deployment and use Docker to set up (part of) the environment.

Let's create an integration test using Arquillian Cube, to test the integration between a service and its database. You'll use a microdeployment approach to package the classes related to the persistence layer. In order to add to the previous Docker Compose file format, make sure to use Docker compose format version 2 rather than version 1.

The following listing shows what a test looks like.

Listing 8.6 Integration test

```
@RunWith(Arquillian.class)
public class UserRepositoryTest {

  @Deployment
  public static WebArchive create() {
    return ShrinkWrap.create(WebArchive.class)
```

Creates a microdeployment using only the required persistence-layer classes and files

```
            .addClasses(User.class, UserRepository.class,
                                              UserRepositoryTest.class)
            .addAsWebInfResource(EmptyAsset.INSTANCE, "beans.xml")
            .addAsResource("test-persistence.xml", "META-
    INF/persistence.xml")
            .addAsManifestResource(new StringAsset(
                                            "Dependencies: com.h2database.h2\n"
    ),
                                    "MANIFEST.MF");
  }

  @Inject
  private UserRepository repository;

  @Test
  public void shouldStoreUser() throws IOException {
    repository.store(new User("test"));
    User user = repository.findUserByName("test");

    assertThat(user.getName()).isEqualTo("test");
  }
}
```

This is no different than running any other Arquillian test, so the test won't be aware of whether it's running against a local or a remote instance.

TIP You can also benefit from using the Arquillian Persistence Extension in these tests if you need to.

The next step is defining a docker-compose.yml file that starts the server and the database.

Listing 8.7 Starting the server and database

```
version: '2'
services:
  tomcat:
    env_file: envs
    build: src/test/resources/tomcat
    ports:
      - "8089:8089"
      - "8088:8088"
      - "8081:8080"
  db:
    image: zhilvis/h2-db
    ports:
      - "1521:1521"
      - "8181:81"
```

Sets environment variables from a file named envs

The Tomcat image is built from a Dockerfile.

Uses the H2 server Docker image

TIP Always pay close attention to the indentation in YAML files. It's vital!

In this file, a default network is created and shared between both containers. The container name is the hostname alias used by each container to look up other instances. For example, a `tomcat` container configuration for reaching `db` might be `jdbc:h2:tcp://db:1521/opt/h2-data/test`.

The Dockerfile should add a tomcat-users.xml file, ready with a user that has the roles to be able to deploy remotely. You need to define environment variables in order to configure Tomcat to accept deploying external applications on the fly, as well as set the password:

```
CATALINA_OPTS=-Djava.security.egd=file:/dev/urandom
JAVA_OPTS= -Djava.rmi.server.hostname=dockerServerIp \
           -Dcom.sun.management.jmxremote.rmi.port=8088 \
           -Dcom.sun.management.jmxremote.port=8089
           -Dcom.sun.management.jmxremote.ssl=false
           -Dcom.sun.management.jmxremote.authenticate=false
```

Trick to change how entropy is calculated so that Tomcat starts up quickly

The dockerServerIp parameter is replaced automatically at runtime by the Docker host IP.

The JMX console is configured to accept remote communication.

Entropy

The entropy trick in the Tomcat configuration snippet is used only on Linux platforms, but it can also improve startup on Windows machines. The default `SecureRandom` implementation is very slow, because it must wait for the OS to build up entropy—and this can take minutes. Specifying `urandom` is slightly less secure for extreme cryptography algorithms. On some systems, you may need to use the alternative syntax if you still notice significant startup times (note the extra slashes):

```
-Djava.security.egd=file:/dev/urandom
```

Other options are available, such as defining a physical file of random numbers. Search the internet for java.security.egd to learn more about this subject.

Last but not least, you need to configure the Arquillian Cube Extension to load the provided Docker Compose file. You also need to configure the remote adapter to set the user declared in the tomcat-users.xml file to connect to the Tomcat server and deploy the application.

Listing 8.8 Configuring the Arquillian Cube Extension and the remote adapter

```
<?xml version="1.0"?>;
<arquillian xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns="http://jboss.org/schema/arquillian"
  xsi:schemaLocation="http://jboss.org/schema/arquillian
  http://jboss.org/schema/arquillian/arquillian_1_0.xsd">;

  <extension qualifier="docker">;
    <property name="dockerContainersFile">;docker-compose.yml</property>;
  </extension>;

  <container qualifier="tomcat">;
    <configuration>;
        <property name="user">;admin</property>;
```

Sets the location of the Docker Compose file. In this case, it's located in the project's root directory.

Configures the Tomcat adapter with admin and mypass as authentication parameters to deploy

```
        <property name="pass">;mypass</property>;
    </configuration>;
  </container>;
</arquillian>;
```

This is all you need to configure. Arquillian Cube automatically takes care of where Tomcat is running and deploying the test application to the correct (remote) Docker host IP. It's important to note that most of these steps are specific to Tomcat, and that using another container might require different steps and touch different files.

> **WARNING** The `qualifier` value in arquillian.xml must be the same container name as that defined in the docker-compose.yml file. In the previous example, the container name is `tomcat` and the qualifier is also `tomcat`, which makes sense.

When you run this test, the following steps are executed:

1. Arquillian Cube reads the Docker Compose file and then builds and instantiates the specified images to the Docker host.
2. Arquillian Core deploys the microdeployment file that contains the persistence-layer classes into the Tomcat container instance running in Docker Host.
3. The test is executed when the entire test environment is set up and booted.
4. After all tests are executed, the microdeployment file is undeployed.
5. The Docker container instances are terminated and removed from the Docker host.

Note that this test now runs in its own provided testing environment that hosts the required database used in production. You don't need to install any software dependencies on your actual development environment or CI environment for each project. Docker and Arquillian Cube take care of providing the dependencies for the test automatically.

Now that you've seen how to write an integration test using Arquillian Cube, let's move on to how you can use it for end-to-end testing.

8.2.4 Writing end-to-end tests

Chapter 7 explained that you can validate your application from start to finish by, in theory, simulating a real-world user of your application, or at least performing the actions of a real user. In practice, these tests are usually the most difficult to write because they cover a lot of interactions—in most cases (but not always) interactions with the UI. They also require that you set up a full test environment with all possible elements the application might interact with, such as

- Server
- Databases
- Distributed caches
- Browsers

You now know that Docker, Docker Compose, and Arquillian Cube can help you prepare the environment for tests. Note that in end-to-end tests, you probably won't need to create a deployment file in your tests; you'll reuse an existing, versioned Docker image of the core of the application. For this reason, and as you saw in chapter 4, where no deployment method is provided, you'll need to use the `standalone` dependency of Arquillian Core.

Let's see what a docker-compose.yml file might look like for the same application you tested in section 8.2.3.

Listing 8.9 Docker Compose file for an end-to-end test

```
version: '2'
services:
  myservice:
    env_file: envs
    image: superbiz/myservice:${version:-latest}
    ports:
      - "8081:8080"
  db:
    image: zhilvis/h2-db
    ports:
      - "1521:1521"
      - "8181:81"
```

The image version is set using a system property or environment variable. If it isn't set, the default value "latest" is used, denoted by the :- symbols.

In this file, you aren't building a new Docker container, but rather are reusing the one built during the process of building the microservice. Each time you run end-to-end tests, the Docker image bundling the microservice may be a different version; for this reason, the final image name containing the microservice is generated dynamically at testing time by setting the version using a system property or environment variable named `version`.

The configuration file (arquillian.xml) doesn't change from the previous use case.

Listing 8.10 Configuration file for an end-to-end test

```
<?xml version="1.0"?>;
<arquillian xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns="http://jboss.org/schema/arquillian"
  xsi:schemaLocation="http://jboss.org/schema/arquillian
  http://jboss.org/schema/arquillian/arquillian_1_0.xsd">;

  <extension qualifier="docker">;
    <property name="dockerContainersFile">;docker-compose.yml</property>;
  </extension>;

</arquillian>;
```

Sets the docker-compose.yml file location

Using multiple Docker Compose file definitions

In simple cases where a microservice isn't a consumer of another microservice, using a single Docker Compose file may do the trick. But if a microservice under test is itself a consumer of one or more microservices, you may also need to start all of these services prior to testing. This is also valid when you want to write an end-to-end test, not for a given microservice and all of its dependencies (which can be other microservices), but for the entire system.

In such cases, you can still rely on creating a single Docker Compose file containing all microservices and dependencies required for testing. But this may not be a good idea in terms of readiness, maintainability, and reflecting changes to the microservices environment.

Our opinion is that each microservice should define its own Docker Compose file to set up the testing/production environment it needs in order to run. This makes end-to-end tests easy, because you can merge all the definitions with the useful Arquillian Cube `dockerContainersFiles` property.

In the following snippet, Arquillian Cube downloads all the remote Docker Compose files and merges them into a single composition. Arquillian Cube then starts all defined containers, after which the test is executed:

```
<?xml version="1.0"?>;
<arquillian xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns="http://jboss.org/schema/arquillian"
  xsi:schemaLocation="http://jboss.org/schema/arquillian
  http://jboss.org/schema/arquillian/arquillian_1_0.xsd">;

  <extension qualifier="docker">;
    <property name="dockerContainersFiles">;
      docker-compose.yml,
      http://myhub/provider1/test/docker-compose.yml,
      http://myhub/provider2/test/docker-compose.yml
    </property>;
  </extension>;

</arquillian>;
```

A list of locations where Docker Compose files are stored

As you can see, it isn't necessary to define the testing environment in a single location. Each microservice can define its own testing environment.

Finally, you can write the end-to-end tests as you learned in chapter 7, using any of the frameworks exposed there. You can enrich each test with different Docker/container environment values such as the Docker host IP, and resolve port binding values for given exposed ports:

```
@HostIp
String ip;

@HostPort(containerName = "tomcat", value = 8080)
```

Injects the Docker host IP

Resolves the binding port for exposed port 8080 of the tomcat container

```
int tomcatPort;

@CubeIp(containerName = "tomcat")
String ip;
```

Injects the IP of the tomcat container

After injecting the Docker host IP and container binding port, you can configure any test framework used for endpoint testing against the Docker container. For example, you could configure REST Assured (http://rest-assured.io) to test a microservice that's running in the Docker host by doing something like this:

```
RestAssured.when()
           .get("http://" + ip + ":" + tomcatPort + "/myresource")
           .then()
           ....
```

This is one way to configure any testing framework by constructing the URL required. But Arquillian Cube offers tight integration with REST Assured and Arquillian Drone/Graphene, so you don't need to deal with this in every test.

8.3 Rest API

Because Arquillian Cube provides integration with REST Assured, you don't need to repeat the same configuration code in all the tests where REST Assured is used with Docker. Having this integration means you can inject an instance of `io.restassured.builder.RequestSpecBuilder` that's preconfigured with the current Docker host IP and port. (The sidebar "About port resolution" explains how the port resolution works.) The following test uses REST Assured integration:

```
@RunWith(Arquillian.class)
public class MyServiceTest {

  @ArquillianResource
  RequestSpecBuilder requestSpecBuilder;

  @Test
  public void should_be_able_to_connect_to_my_service() {
    RestAssured
      .given()
      .spec(requestSpecBuilder.build())
      .when()
      .get()
      .then()
      .assertThat().body("status", equalTo("OK"));
  }
}
```

RequestSpecBuilder with Docker parameters predefined

REST Assured is configured with a request specification.

As you can see, this test is similar to any test using REST Assured. The only difference is that now you're setting the request-specification object configured with Docker values.

About port resolution

Arquillian Cube REST Assured integration tries to automatically resolve which port is the binding port of the public microservice. By default, Arquillian Cube scans all Docker containers defined in Docker Compose files, and if there's only one binding port, it's the one that's used.

If there are several binding ports, then the `port` configuration property must be defined for the exposed port that Arquillian Cube should use to communicate with the microservice. For example, if you're using the binding configuration 8080:80, where the exposed port is 80 and the binding port is 8080, then when you set the `port` property to `80`, the extension will resolve to 8080.

To set the `port` property, you need to add it to arquillian.xml:

```
<extension qualifier="restassured">;
  <property name="port">;80</property>;
</extension>;
```

Sets the REST Assured configuration section

Exposes port to resolve

If there's no exposed port with the given number, then the port specified in the configuration property is also used as the binding port.

8.4 Arquillian Drone and Graphene

When you're running end-to-end tests that involve a browser as a frontend UI, one of the problems you may encounter is setting up the testing environment. You'll need to install all requirements on every machine where tests are run, including the required browsers (with specific versions).

As you learned in chapter 7, the de facto tool for web-browser tests is Selenium WebDriver. The Arquillian ecosystem offers Arquillian Drone and Graphene as the integration extension that uses WebDriver.

The Selenium project offers Docker images for the Selenium standalone server with Chrome and/or Firefox preinstalled. So, you effectively don't need to install a browser in the testing environment, because the browser is treated like any other test dependency managed by Docker, such as databases, distributed caches, and other services.

8.4.1 Integrating Arquillian Cube and Arquillian Drone

Arquillian Cube integrates with Arquillian Drone by automatically executing several cumbersome tasks:

- Starting the Docker container with the correct `browser` property for the `webdriver` extension set to `Firefox`, if not already set. The Selenium version of the image is the same as that defined in the test classpath.
- Providing a `WebDriver` that can connect to the container.
- Creating a virtual network computing (VNC) Docker container that records all test executions that occur for each test in a browser container and stores them on the local machine in MP4 format.

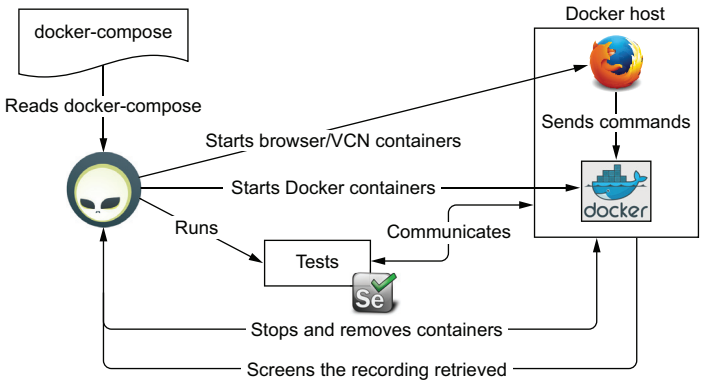


Figure 8.5 Arquillian integrations

These interactions are summarized in figure 8.5.

Table 8.2 describes the most important configuration attributes that you can define in arquillian.xml.

Table 8.2 Arquillian Cube Graphene configuration parameters

Attribute	Description	Default behavior
`recordingMode`	Recording mode to be used. The valid values are `ALL`, `ONLY_FAILING`, and `NONE`.	`ALL`
`videoOutput`	Directory where videos are stored.	Creates target/reports/videos or, if target doesn't exist, build/reports/videos.
`browserImage`	Docker image to be used as a custom browser image instead of the default image.	
`browserDockerfileLocation`	Dockerfile location to be used to build a custom Docker image instead of the default Dockerfile. This property has preference over `browserImage`.	

NOTE Custom images must expose port 4444 so the `WebDriver` instance can reach the browser. If VNC is used, port 5900 must also be exposed.

Here's an example of a typical configuration:

```
<?xml version="1.0"?>;
<arquillian xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns="http://jboss.org/schema/arquillian"
  xsi:schemaLocation="http://jboss.org/schema/arquillian
  http://jboss.org/schema/arquillian/arquillian_1_0.xsd">;

  <extension qualifier="docker">;
```

Typical Arquillian Cube configuration

```
    <property name="dockerContainersFile">;docker-compose.yml</property>;
  </extension>;

  <extension qualifier="webdriver">;
    <property name="browser">;${browser:chrome}</property>;
  </extension>;

  <extension qualifier="cubedrone">;
    <property name="recordingMode">;NONE</property>;
  </extension>;

</arquillian>;
```

Configures the browser property from either a system property or an environment variable, using "chrome" as the default

Disables recording capabilities

Here, Arquillian Cube is configured to start all containers defined in the docker-compose.yml file. Notice that this file doesn't contain any information regarding browsers, because this is autoresolved by the Arquillian Cube Drone integration.

The browser is specified by setting the `browser` system property or environment variable to `firefox` or `chrome`. If it isn't defined, `chrome` is used as the default. Finally, the recording feature is disabled.

> **WARNING** At time of writing, the Selenium project offers images only for Firefox and Chrome. Creating Internet Explorer images is still a task left to the user.

The actual test shown in listing 8.11 (HelloWorldTest.java) looks similar to any Drone and Arquillian Cube test, with one slight difference. All browser commands (hence, `WebDriver`) are executed *inside* the Docker host, meaning you're governed by Docker host rules. Thus, in this test, instead of using `HostIp` to get the Docker host IP, you use `CubeIp`, which returns the *internal IP address* of the given container. This is required because the browser is running inside the Docker host, and to connect to another container in the same Docker host, you need either the host name or the internal IP address.

Listing 8.11 HelloWorldTest.java

```
@RunWith(Arquillian.class)
public class HelloWorldTest {

  @Drone
  WebDriver webDriver;

  @CubeIp(containerName = "helloworld")
  String ip;

  @Test
  public void shouldShowHelloWorld() throws Exception {
    URL url = new URL("http", ip, 80, "/");
    webDriver.get(url.toString());
```

Uses the Drone annotation to inject the WebDriver instance

Injects the internal IP of the helloworld container

URL to connect the browser to the microservice

```
    final String message = webDriver.findElement(By.tagName("h1")).getText();
    assertThat(message).isEqualTo("Hello world!");
  }

}
```

Next, let's look at how Arquillian Cube and Arquillian Graphene are integrated.

8.4.2 Integrating Arquillian Cube and Arquillian Graphene

Arquillian Graphene is a set of extensions for the WebDriver API, focused on rapid development and usability in a Java environment. It strives for reusable tests by simplifying the use of web page abstractions (page objects and page fragments).

Arquillian Graphene depends on Arquillian Drone to provide an instance of `WebDriver`, so everything that's valid in integration between Arquillian Cube Docker (such as recording capabilities) is also valid for Arquillian Cube Graphene.

One of the primary things that differentiates a test written in Arquillian Drone from a test written in Arquillian Graphene is that the latter test automatically resolves the host and the context of the application. In Arquillian Drone, you need to explicitly set them by calling the `webdriver.get(...)` method.

This autoresolution feature provided by Arquillian Graphene works only when you're running tests in container mode. (Arquillian manages the deployment file for classes that have a `@Deployment` method.) When you're using standalone mode (no `@Deployment` declaration), which may be the case in end-to-end tests, you need to configure Arquillian Graphene in arquillian.xml with the URL where the application is deployed:

```
<extension qualifier="graphene">;
  <property name="url">;http://localhost:8080/myapp</property>;
</extension>;
```

Sets the URL to be used by Graphene tests

The problem is that when you're using Arquillian Cube, you may not know the Docker host IP address at configuration time—only during the runtime phase. So you can't reliably set it yet!

Arquillian Cube integrates with Arquillian Graphene by providing a special keyword, `dockerHost`, that can be defined in the `url` property and is replaced at runtime by the current Docker host IP when the test environment fires up. Additionally, if the *host* part of `url` isn't `dockerHost` or a valid IP, then this host is considered to be the Docker container name and is replaced by its container internal IP.

Knowing this, the previous example can be rewritten so it's Arquillian Cube Graphene aware:

```
<extension qualifier="docker">;
  <property name="dockerContainersFile">;docker-compose.yml</property>;
</extension>;

<extension qualifier="graphene">;
  <property name="url">;http://helloworld:8080/myapp</property>;
</extension>;
```

Sets the URL to be used by Graphene

Based on this information, you now know the following:

- The `helloworld` part of the URL will be replaced by the container's internal IP.
- The port used should be the exposed port for the `helloworld` container.

You can now define a page object, as in any other Arquillian Graphene test:

```
@Location("/")
public class HomePage {
```

Path of the page represented by this page object

```
  @FindBy(tagName = "h1")
  private WebElement welcomeMessageElement;

  public void assertOnWelcomePage() {
    assertThat(this.welcomeMessageElement.getText().trim())
              .isEqualTo("Hello world!");
  }
}
```

Note that in this case, you aren't setting any information about the actual hostname, you're just setting the relative context location of this page.

Finally, the test has no changes versus a normal test. Everything is managed underneath by Arquillian Cube:

```
@RunWith(Arquillian.class)
public class HomePageTest {

  @Drone
  WebDriver webDriver;

  @Test
  public void shouldShowHelloWorld(@InitialPage HomePage homePage) {
    homePage.assertOnWelcomePage();
  }
}
```

As you can see, the host information isn't present in any test. It's resolved and provided to the environment via the arquillian.xml file. This makes the tests reusable in any environment, because you can change the base URL dynamically before executing the tests.

> **NOTE** Arquillian Cube Graphene autoresolution is required only if you're using Arquillian in standalone mode (using the standalone dependency). If you're using container mode, the URL is resolved by the deployment method, and you don't need to specify anything. In our opinion, an end-to-end test with Docker (or in general) should be written using Arquillian standalone mode; this simulates a real production environment much more closely, which is ultimately what you're trying to achieve.

8.5 Parallelizing tests

One of the problems you may encounter when running tests against a single Docker host is that each container running in the host must have a unique name. In general, this may not be an issue, but in some situations it can lead to a conflict:

- If you run tests defined in the same project in parallel, then, assuming they're reusing the same Docker Compose file, the same Docker host is used. This will cause a conflict because you're using the same container name for each test.
- Different projects are running tests in your CI environment and reusing the same Docker host. For example, two microservices have defined a Docker container named `db`, and they're building at the same time.

Workarounds are available to mitigate these problems:

- The first problem can be resolved by setting the `arq.extension.docker.serverUri` property in each parallel execution to use a different Docker host.
- The second problem can be resolved by using one agent/slave for each project, each of which has its own Docker host.

In conjunction with these workarounds, Arquillian Cube offers a helpful tool called the *star operator* (`*`).

The star operator lets you indicate to Arquillian Cube that you want to generate part of the Docker container name randomly. All generated information is automatically adapted to use the random element. The only thing you need to do is add an asterisk character (`*`) at the end of the container name in the Docker Compose file. Here's an example:

```
tomcat:
  image: tutum/tomcat:7.0
  ports:
    - "8081:8080"
  links:
    - pingpong*

pingpong*:
  image: jonmorehouse/ping-pong
  ports:
    - "8080:8080"
```

Sets a link to a partially random container name

Sets the container name as partially random

Given this Docker Compose file, Arquillian Cube will substitute the `*` character for a UUID generated for each execution at runtime. Binding ports are changed to a random port (in the range 49152–65535). And a new environment variable with a link to the random container setting at the new host location is provided to containers; the form of this environment variable is `<containerName>_HOSTNAME`.

The resulting docker-compose.yml file after Arquillian Cube has applied the changes might look like this:

```
tomcat:
  image: tutum/tomcat:7.0
  ports:
    - "8081:8080"
  links:
    - pingpong_123456
  environment:
    - ping_HOSTNAME=ping_123456

pingpong_123456:
  image: jonmorehouse/ping-pong
  ports:
    - "54678:8080"
```

Environment variable with the hostname of the new container

Link updated to a random container name

Container defined with a random name

Binding port updated to a random port

WARNING Using the star operator will make your Docker Compose file incompatible with the `docker-compose` CLI. Also note that the `hostname` entry in DNS defined by the `links` section is also randomly generated, because the container name has been changed.

The star operator isn't an all-or-nothing solution—you can use it together with other approaches. The ideal scenario has one Docker host for each parallel execution or slave/agent.

8.6 Arquillian Cube and Algeron

In chapter 6, you learned about consumer-driven contracts and how they're run using the Arquillian Algeron Extension. You execute them in two steps: the first step is on the consumer side, where you start a stub HTTP server and send requests to it; and the second step is to replay and verify all interactions that occurred against a real provider. These interactions are summarized in figure 8.6.

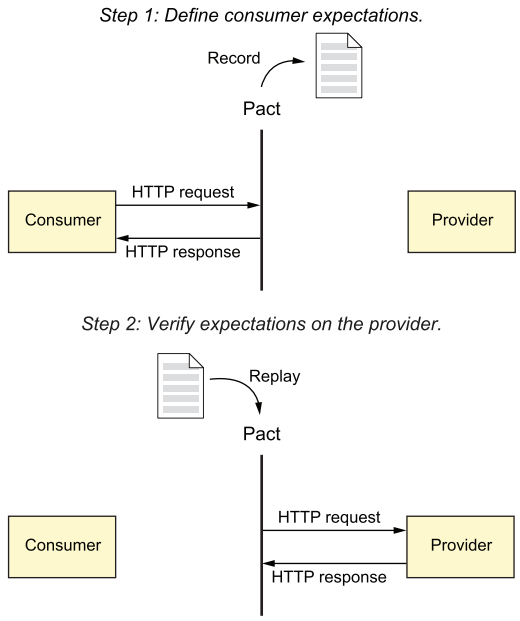


Figure 8.6 Pact lifecycle

To run provider contract tests, you need to deploy the provider service and then replay and verify all interactions. This is where Docker and Arquillian Cube can help you, by simplifying the deployment phase of the provider service.

So far, there isn't much difference between using Arquillian Algeron with Arquillian Cube. But let's look at a quick example where a new Arquillian Cube enrichment method is introduced:

```
@RunWith(Arquillian.class)
@Provider("provider")
@ContractsFolder("pacts")
public class MyServiceProviderTest {

  @ArquillianResource
  @DockerUrl(containerName = "helloworld", exposedPort = "8080",
➥ context = "/hello")
  URL webapp;

  @ArquillianResource
  Target target;

  @Test
  public void should_provide_valid_answers() {
    target.testInteraction(webapp);
  }

}
```

Arquillian Cube and Arquillian Algeron annotations

Enriches the URL with valid Docker values

Enriches the Arquillian Algeron Target

Replays verification against the Docker container

This test is basically the same as any other Arquillian Cube and Arquillian Algeron Pact provider, but in this case the test is enriched with a URL to access the provider. This URL is created by resolving `dockerHost` as the host part. The port part is appended by obtaining the exposed port set in the annotation for the specific container (`"helloworld"`, in this case). The annotation-defined context is then appended. For example, the resulting URL might have the value `http://192.168.99.100:8081/hello`. The rest of the test is pretty much the same as any other.

> **TIP** You can use the `@DockerUrl` annotation in any Arquillian Cube standalone test, not just when using Arquillian Algeron. But note that the enrichment test is in *standalone* mode (there's no `@Deployment` method). The `@DockerUrl` enrichment works only when you're running Arquillian in standalone mode.

Of course, you still need to define a Docker Compose file and configure Arquillian Cube. But for the sake of simplicity, and because you've seen that in previous sections, we skipped these steps.

8.7 Using the container-objects pattern

So far, you've seen how to "orchestrate" Docker containers using the Docker Compose file. Arquillian Cube also offers another way to define Docker containers using a Java object.

Using Java objects to define the container configuration enables you to add some dynamism to the Docker container definition, such as modifying a Dockerfile's content programmatically or modelling container attributes such as IP address, username, and password. Also, because you're creating Java objects, you can use any of the resources the language provides, such as extending definitions, injecting values from tests, or packaging values as delivery artifacts.

You can think of *container objects* as a way to model containers in a reusable, maintainable way. Because they're Java objects, nothing prevents you from reusing them across multiple projects. This reduces the amount of duplicated code, and any fixes only need to be applied in one place instead of in multiple projects.

Before we show you how to implement container objects, let's look at an example where they're useful. Suppose your microservice (or project) needs to send a file to an FTP server. You need to write an integration test validating that your business code can execute this operation correctly.

Your test must be able to execute the following operations:

- Find the hostname/IP and port where the FTP server is running.
- Define the username and password required to access the FTP server and store files.
- Assert the existence of a file on the FTP server to verify that the file is correctly sent.

One way to write this test would be to use the Docker Compose approach:

```
ftp:
  image: andrewvos/docker-proftpd
  ports:
    - "2121:21"
  environment:
    - USERNAME=alex
    - PASSWORD=aixa
```

This approach has several problems:

- You need to copy this docker-compose.yml file to all projects where you want to write an integration test using the FTP server.
- Tests need to know internal details of the Docker container, such as username and password.
- Tests contain logic specific to the Docker container, such as how to validate that a file has been copied.
- Any changes need to be propagated to all use cases.

You obviously have a good contender here for writing a container-object pattern that encapsulates all logic related to the FTP server. The following listing shows what this container object might look like (FtpContainer.java).

Listing 8.12 Container-object pattern

Defines the Cube name and binding ports

Sets the Docker image

Configures environment variables to be set in the container

Enables enrichments in container objects

Encapsulates operations related to the container

```
@Cube(value = "ftp",
    portBinding = FtpContainer.BIND_PORT + "->;21/tcp")
@Image("andrewvos/docker-proftpd")
@Environment(key = "USERNAME", value = FtpContainer.USERNAME)
@Environment(key = "PASSWORD", value = FtpContainer.PASSWORD)
public class FtpContainer {

    static final String USERNAME = "alex";
    static final String PASSWORD = "aixa";
    static final int BIND_PORT = 2121;

    @ArquillianResource
    DockerClient dockerClient;

    @HostIp
    String ip;

    public String getIp() {
        return ip;
    }

    public String getUsername() {
        return USERNAME;
    }

    public String getPassword() {
        return PASSWORD;
    }

    public int getBindPort() {
        return BIND_PORT;
    }

    public boolean isFilePresentInContainer(String filename) {

        InputStream file = null;

        try (
            file = dockerClient
              .copyArchiveFromContainerCmd("ftp", "/ftp/" + filename)
              .exec()){
            return file != null;
        } catch(Exception e){
            return false;
        } finally{
            if (null != file) {
                try {
                    file.close();
                } catch (IOException e) {
                    //no-op
                }
            }
        }
    }
}
```

As you can see in this class, a container object is a plain old Java object (POJO). This object is annotated with configuration parameters required for starting the container, such as `name` and `bind/expose` ports using the `@Cube` annotation, and which Docker image is mapped using the `@Image` annotation. Any Arquillian test enrichment that you've learned so far can be applied to a container object, such as `host_ip`, `host_port`, and `docker client`.

Now you've seen how to define a container object, here's how to use it in a test (FtpClientTest.java).

Listing 8.13 Using a container object in a test

```
@RunWith(Arquillian.class)
public class FtpClientTest {

  public static final String REMOTE_FILENAME = "a.txt";

  @Cube                                          <-- The container object is
  FtpContainer ftpContainer;                         annotated with @Cube.

  @Rule
  public TemporaryFolder folder = new TemporaryFolder();

  @Test
  public void should_upload_file_to_ftp_server() throws Exception {

    // Given
    final File file = folder.newFile(REMOTE_FILENAME);
    Files.write(file.toPath(), "Hello World".getBytes());

    // When
    FtpClient ftpClient = new FtpClient(ftpContainer.getIp(),   <-- Retrieves
              ftpContainer.getBindPort(),                           FTP properties
              ftpContainer.getUsername(),                           from the
              ftpContainer.getPassword());                          injected object
    try {
        ftpClient.uploadFile(file, REMOTE_FILENAME, ".");
    } finally {
        ftpClient.disconnect();
    }

    // Then
    final boolean filePresentInContainer = ftpContainer
      .isFilePresentInContainer(REMOTE_FILENAME);       <-- Encapsulates operations
    assertThat(filePresentInContainer, is(true));           related to the container
  }
}
```

Using a container object in an Arquillian Cube test is as simple as declaring it as such and annotating it with `@Cube`. At execution time, Arquillian Cube inspects all test classes for fields annotated with `Cube`, reads all metainformation, and starts the defined containers. After the test is executed, the container is stopped.

As you can see, the lifecycle of a container object isn't much different from defining it in a Docker Compose file. Note that in this case, there's no need for a Docker Compose file, although you could use both approaches together if you wanted to.

Updating default values

When you're working with a POJO, you can override any part of it using normal Java language conventions. In the following example, the test overrides the name of the container, as well as the port binding configuration:

```
@Cube(value = "myftp",
        portBinding =  "21->;21/tcp")
FtpContainer ftpContainer;
```

Updates default values provided by the container object

Another feature offered by the container-objects pattern in Arquillian Cube is the possibility of not depending on a specific image. You can build your own image from a Dockerfile using the `CubeDockerFile` annotation:

```
@Cube(value = "ftp",
        portBinding =  FtpContainer.BIND_PORT +  "->;21/tcp")
@CubeDockerFile("/docker")
@Environment(key = "USERNAME", value = FtpContainer.USERNAME)
@Environment(key = "PASSWORD", value = FtpContainer.PASSWORD)
public class FtpContainer {

}
```

Builds image from a configured Dockerfile

The `CubeDockerFile` annotation sets the location where the Dockerfile can be found, but it doesn't limit the contents of the Dockerfile. This location must be accessible by the runtime `ClassLoader`, so it must be present on the classpath.

You can also create the Dockerfile programmatically using the ShrinkWrap Descriptors domain-specific language (DSL). The following example shows how a Dockerfile can be defined using the DSL in a container object:

```
@Cube(value = "ftp", portBinding = "2121->;21/tcp")
public class FtpContainer {

  @CubeDockerFile
  public static Archive<?>; createContainer() {
    String dockerDescriptor = Descriptors.create(DockerDescriptor.class)
            .from("andrewvos/docker-proftpd")
            .expose(21)
            .exportAsString();
    return ShrinkWrap.create(GenericArchive.class)
            .add(new StringAsset(dockerDescriptor), "Dockerfile");
    }
}
```

Static method for defining a Dockerfile

Creates Dockerfile content using the ShrinkWrap Descriptors DSL

Builds an archive with all required content

The method that builds the Dockerfile must be annotated with `CubeDockerFile`, and it must be public and static with no arguments. In addition, the method needs to return a ShrinkWrap `Archive` instance. The Dockerfile isn't returned directly, because in certain circumstances you might need to add extra files required for building the Docker container. This is especially true when you need to add files during container creation.

The last feature offered by the container-objects pattern is the aggregation of containers. Aggregation allows you to define container objects in other container objects. Each aggregated object contains a link to its parent, so each of the involved parties can communicate with each other.

Here's how to define an inner container object:

```
@Cube
public class FirstContainerObject {
  @Cube("inner")
  LinkContainerObject linkContainerObject;
}
```

In addition to starting both containers, Arquillian Cube creates a link between them by setting the hostname for `LinkContainerObject` to `inner`. The link can be further configured by using a `@Link` annotation:

```
@Cube("inner")
@Link("db:db")
TestLinkContainerObject linkContainerObject;
```

8.7.1 Using a flexible container-object DSL

Arquillian Cube also provides a generic `Container` object to generate Cube instances. Writing definitions is more efficient when you use this approach, but it's a little more difficult to reuse the code or to provide the custom operations with the custom container-object approach.

Let's look at a simple example of how to declare and start a Docker container using the `Container` DSL:

```
@DockerContainer
Container pingpong = Container.withContainerName("pingpong")
                        .fromImage("jonmorehouse/ping-pong")
                        .withPortBinding(8080)
                        .build();

@Test
public void should_return_ok_as_pong() throws IOException {
   String response = ping(pingpong.getIpAddress(),
                          pingpong.getBindPort(8080));
   assertThat(response).containsSequence("OK");
}
```

The field is annotated with @DockerContainer.

The DSL starts with the withContainerName method.

Gets container information to connect

To create a generic container object, you only need to create a field of type `org.arquillian .cube.docker.impl.client.containerobject.dsl.Container` and annotate it with `@DockerContainer`.

You can also create a Docker network using the DSL approach:

```
@DockerNetwork
Network network = Network.withDefaultDriver("mynetwork").build();
```

The field is annotated with @DockerNetwork.

The DSL starts with the withDefaultDriver method.

To create a network using the DSL approach, you create a field of type `org.arquillian .cube.docker.impl.client.containerobject.dsl.Network` and annotate it with `@DockerNetwork`.

Container objects and DSL JUnit rules

You can define generic containers using a *JUnit rule*. This way, you can use any JUnit runner such as `SpringJUnit4ClassRunner` side by side with the container object DSL. Here's how to define a Redis container:

```
@ClassRule
public static ContainerDslRule redis = new ContainerDslRule("redis:3.2.6")
                                            .withPortBinding(6379);
```

Spring Data and Spring Boot

With Spring Data, you configure the database location using environment variables. To set them in the test, you need to use a custom `ApplicationContextInitializer`. Here's an example:

```
@RunWith(SpringJUnit4ClassRunner.class)
@SpringBootTest(classes = Application.class,
                webEnvironment = WebEnvironment.RANDOM_PORT)
@ContextConfiguration(initializers =
        SpringDataTest.Initializer.class)
public class SpringDataTest {

  @ClassRule
  public static ContainerDslRule redis =
      new ContainerDslRule("redis:3.2.6")
      .withPortBinding(6379);
```

Spring JUnit test runner with Boot configuration

Setup initializer to configure environment variables

Defines a Redis container

```
  public static class Initializer implements
              ApplicationContextInitializer
                <ConfigurableApplicationContext>; {
    @Override
    public void initialize(                                    Initializer
              ConfigurableApplicationContext                   implementation with
                configurableApplicationContext) {              container configuration
        EnvironmentTestUtils.addEnvironment("testcontainers",
                      configurableApplicationContext.getEnvironment(),
                      "spring.redis.host=" + redis.getIpAddress(),
                      "spring.redis.port=" + redis.getBindPort(6379)
        );
    }
  }
}
```

Note that you must add the `org.arquillian.cube:arquillian-cube-docker-junit-rule` dependency. You don't need to add any other Arquillian dependency.

So far, you've learned how to use Docker and Arquillian Cube to set up a complex testing environment. In the next section, we'll look at how to use and deploy Docker images in Kubernetes.

8.8 Deployment tests and Kubernetes

In this chapter, you've seen how you can use Docker for testing purposes, but perhaps you're also using Docker at the production level. Docker containers on their own can be difficult to manage and maintain. Complex applications typically require booting up multiple containers on multiple machines (note that the Docker host runs on a single host). You also need a way to orchestrate all these containers and offer other features such as fault tolerance, horizontal autoscaling, distribution of secrets, naming and discovery of services across all machines, rolling updates, and load balancing. One standout tool that offers these features is Kubernetes.

Kubernetes is an open source system for managing clusters of Docker containers. It was created by Google; several other companies have contributed to it, including Red Hat and Microsoft. Kubernetes provides tools for deploying and scaling applications, as well as managing changes to existing applications, such as updating to a new version or rolling back in the case of failure or health checks. Moreover, Kubernetes was created with two important features in mind: extensibility and fault tolerance.

Following are the primary Kubernetes concepts you need to understand (figure 8.7 summarizes these concepts):

- *Pod*—The minimal unit of organization in Kubernetes. A pod is composed of one or more containers that are run on the same host machine and can share resources.

- *Service*—A set of pods and the policy by which to access them. Kubernetes provides a stable IP address and DNS name to the service, which abstracts from the pods' location. Because pods are ephemeral, their IP address can change. Services react to this change by always forwarding to the location of the required pod. Services act as a load balancer among all pod instances.
- *Replication controller (RC)*—Maintains the desired state of the cluster. For example, if you need three instances of a pod, the RC will manage the given pod and always have three instances of it running on the cluster.
- *Namespace*—A way to create a virtual cluster backed by a physical cluster.

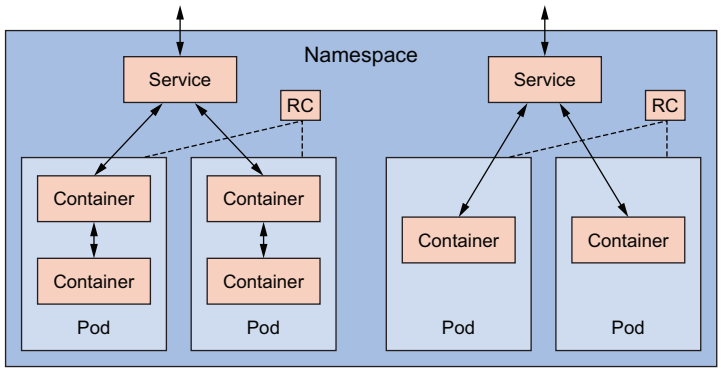


Figure 8.7 An example Kubernetes deployment

Usually, in Kubernetes, you define elements in a JSON or YAML file. For example, to define a pod, you could create the following pod-redis.json file:

```
{
  "kind": "Pod",
  "apiVersion": "v1",
  "metadata": {
    "name": "redis",
    "labels": {
      "app": "myredis"
    }
  },
  "spec": {
    "containers": [
      {
        "name": "key-value-store",
        "image": "redis",
        "ports": [
          {
            "containerPort": 6379
          }
        ]
      }
    ]
  }
}
```

This snippet defines a simple pod, using the `redis` container named `key-value-store`, exposing port number 6379 on the pod's IP address.

In Kubernetes, deploying an application/service isn't done manually but rather is programmable and automatic. This implies that you need to test that what's configured is what you expect to be deployed.

Arquillian Cube provides support for writing tests for Kubernetes. The idea behind the integration of Arquillian Cube and Kubernetes is to consume and test the provided services as well as validate that the environment is in the expected state. The integration lifecycle of Arquillian Cube and Kubernetes is summarized in figure 8.8.

WARNING Only Arquillian standalone mode is supported in the integration of Arquillian Cube and Kubernetes.

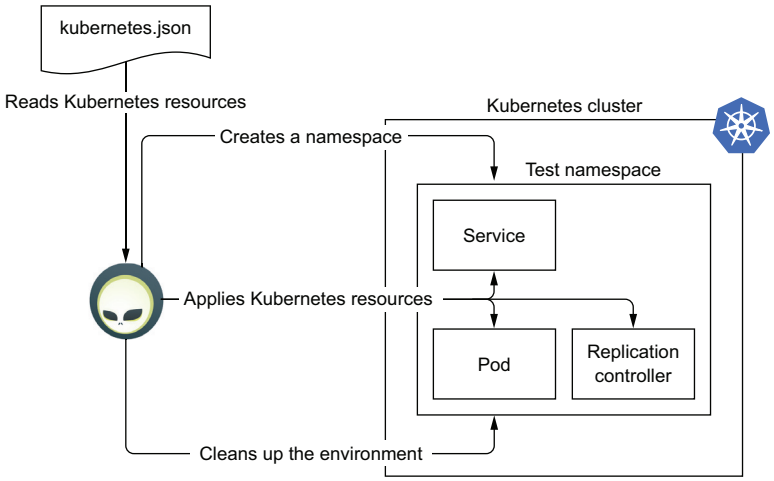


Figure 8.8 Lifecycle of Arquillian Cube and Kubernetes integration

Arquillian Cube Kubernetes creates a temporary namespace to deploy all Kubernetes resources in an isolated environment. Then it searches in the classpath for a file called kubernetes.json or kubernetes.yaml, and applies all Kubernetes resources required on that temporary namespace. Once everything is ready, it runs your tests (using a black box approach). And when testing is finished, it cleans up.

NOTE Arquillian Cube Kubernetes needs to authenticate into Kubernetes. To do so, Arquillian Cube reads user information (token and password) from ~/.kube/config.

You can configure Arquillian Cube Kubernetes parameters in arquillian.xml. Table 8.3 lists some of the most useful parameters.

Table 8.3 Arquillian Cube Kubernetes parameters

Attribute	Description	Default behavior
kubernetes.master	URL for the Kubernetes master	
env.config.url	URL for the Kubernetes JSON/YAML file	Defaults to a classpath resource kubernates.json
env.dependencies	Whitespace-separated list of URLs pointing to more than one Kubernetes definition file	
env.config.resource.name	Option to select a different classpath resource	
namespace.use.existing	Flag that specifies not to generate a new temporary namespace, but to reuse the one that's set	
env.init.enabled	Flag to initialize the environment with defined Kubernetes resources (goes hand to hand with namespace.use.existing)	By default, creates Kubernetes resources
namespace.cleanup.enabled	Instructs the extension to destroy the namespace after the end of the test suite	By default, destroys namespace to keep the cluster clean

NOTE Arquillian Cube Kubernetes can read properties from environment variables. The equivalent environment properties are property names in all caps, with the dot (.) symbol converted to _: for example, KUBERNETES_MASTER.

You can configure kubernetes.master (if the KUBERNETES_MASTER environment variable isn't set), by setting it in arquillian.xml:

```
<arquillian xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns="http://jboss.org/schema/arquillian"
  xsi:schemaLocation="http://jboss.org/schema/arquillian
  http://jboss.org/schema/arquillian/arquillian_1_0.xsd">;

  <extension qualifier="kubernetes">;
    <property name="kubernetes.master">;http://localhost:8443</property>;
  </extension>;

</arquillian>;
```

Any Arquillian Cube Kubernetes test can be enriched with the following elements:

- A Kubernetes client
- A session object containing test-session information such as the name of the temporarily created namespace
- A pod (by its ID) or a list of all pods started by the test
- An RC (by its ID) or a list of all RCs started by the test

- A service (by its ID) or a list of all services started by the test
- The URL of a service

The following test is enriched with some of these elements:

```
@RunWith(Arquillian.class)
public class ResourcesTest {

  @ArquillianResouce                    <-- Enriches the test with a service named my-service
  @Named("my-serivce")
  Service service;

  @ArquillianResouce                    <-- Enriches the test with all pods defined in the test
  PodList pods;

  @Test
  public void testStuff() throws Exception {
  }
}
```

In this test, all information of a Kubernetes service called `my-service` can be queried. You can also access all pods defined in the current test. In a similar way, you can get the list of services by using `ServiceList`, or a concrete pod by using `@Named` and the `Pod` object. The same goes for RC objects.

To inject a service URL, you do this:

```
@Named("hello-world-service")          <-- Service name
@PortForward
@ArquillianResource
URL url;
```

Moreover, as Arquillian Cube Docker does, Kubernetes integration provides tight integration with AssertJ, to provide a readable way of writing assertions about the environment. Here's a simple example of how to use this integration:

```
@RunWith(Arquillian.class)
public class RunningPodTest {

  @ArquillianResource                   <-- Kubernetes client enrichment
  KubernetesClient client;

  @Test
  public void should_deploy_all_pods() {
    assertThat(client).deployments().pods().isPodReadyForPeriod();
  }                                     <-- AssertJ Kubernetes integration's assertThat method
}
```

This test asserts that the current `Deployment` creates at least one pod, which becomes available within a time period (by default, 30 seconds), and that it stays in the `Ready` state for a time period (by default, 1 second). This test is simple, but in our experi-

ence it catches most errors that may occur during deployment time on Kubernetes. Of course, this is just a start; you can add as many assertions as you need to validate that the application is deployed as required. Moreover, AssertJ Kubernetes provides custom assertions not only for `KubernetesClient` but also for pods, services, and RCs.

> **Arquillian Cube Kubernetes and OpenShift**
>
> Arquillian Cube Kubernetes implements some extra features to help with testing through OpenShift:
>
> - Automatic setup of connecting to non-exported routes.
> - Triggering the build job/pipeline directly from the test. This pushes the `@Deployment` artifact to a local repository and triggers an OS build for deployment.
>
> Because OpenShift version 3 is a Kubernetes system, everything that's valid in Arquillian Cube Kubernetes is valid in OpenShift.

After this introduction to using Docker for testing purposes and which tools you can use to automate tests using Docker, let's see what you need to do to begin using them.

8.9 Build-script modifications

You've seen that Arquillian Cube has integrations with different technologies such as Docker, Kubernetes, and OpenShift. Each has its own dependencies, and the following sections cover how to add those dependencies to your tests.

8.9.1 Arquillian Cube Docker

To use Cube Docker integration, you need to add the following dependency:

```
dependencies {
  testCompile group: 'org.arquillian.cube',
              name: 'arquillian-cube-docker',
              version: '1.2.0'
}
```

To use Drone/Graphene integration, you also need to add this:

```
dependencies {
  testCompile group: 'org.arquillian.cube',
              name: 'arquillian-cube-docker-drone',
              version: '1.2.0'
}
```

Notice that in this case, you need to add Selenium, Arquillian Drone, or Arquillian Graphene dependencies, as you did in chapter 7.

To use REST Assured integration, you also need to add the REST Assured dependency:

```
dependencies {
  testCompile group: 'org.arquillian.cube',
              name: 'arquillian-cube-docker-restassured',
              version: '1.2.0'
}
```

Finally, to use AssertJ integration, add the AssertJ dependency:

```
dependencies {
  testCompile group: 'org.arquillian.cube',
              name: 'assertj-docker-java',
              version: '1.2.0'
}
```

8.9.2 Arquillian Cube Docker JUnit rule

To use container DSL with JUnit rule support, add the following dependency:

```
dependencies {
  testCompile group: 'org.arquillian.cube',
              name: 'arquillian-cube-docker-junit-rule,
              version: '1.2.0'
}
```

8.9.3 Arquillian Cube Kubernetes

To use Arquillian Cube with Kubernetes support, add this dependency:

```
dependencies {
  testCompile group: 'org.arquillian.cube',
              name: 'arquillian-cube-kubernetes',
              version: '1.2.0'
}
```

To use AssertJ integration, you also need to add this:

```
dependencies {
  testCompile group: 'io.fabric8',
              name: 'kubernetes-assertions',
              version: '2.2.101'
}
```

8.9.4 Arquillian Cube OpenShift

To use Arquillian Cube with specific features of OpenShift (not the Kubernetes part), add the following dependency:

```
dependencies {
  testCompile group: 'org.arquillian.cube',
              name: 'arquillian-cube-openshift',
              version: '1.2.0'
}
```

8.10 Testing the Dockerfile for the video service

The video service is packaged into a Docker image. To create this image, use the following Dockerfile (code/video/Dockerfile).

Listing 8.14 Video-service Dockerfile

```
FROM java:8-jdk

ADD build/libs/video-service-*.jar /video-service.jar

EXPOSE 8080
RUN bash -c 'touch /video-service.jar'
ENTRYPOINT ["java", "-Djava.security.egd=file:/dev/./urandom",
            "-jar","/video-service.jar"]
```

As you can see, nothing special happens here. The output of the Spring Boot artifact is copied inside the image, and when the image is instantiated, the service is also started.

Next, you create a Docker Compose file to automate the building of the image (code/video/c-tests/src/test/resources/arquillian.xml).

Listing 8.15 Docker Compose file

```
<?xml version="1.0"?>;
<arquillian xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
            xmlns="http://jboss.org/schema/arquillian"
            xsi:schemaLocation="http://jboss.org/schema/arquillian
  http://jboss.org/schema/arquillian/arquillian_1_0.xsd">;


    <extension qualifier="docker">;
        <property name="machineName">;dev</property>;
        <property name="dockerContainers">;
            videoservice:
              build: ../.
              environment:
                - SPRING_REDIS_HOST=redis
                - SPRING_REDIS_PORT=6379
                - YOUTUBE_API_KEY=${YOUTUBE_API_KEY}
              ports:
                - "8080:8080"
              links:
                - redis:redis
            redis:
              image: redis:3.2.6
        </property>;
    </extension>;

</arquillian>;
```

Builds and defines dependencies of the service

Finally, you can write the test to validate that the image can be built and the container instantiated (code/video/c-tests/src/test/java/book/video/VideoServiceContainerTest.java).

Listing 8.16 Validating the image and the container

```
@RunWith(Arquillian.class)
public class VideoServiceContainerTest {


    @ArquillianResource
    DockerClient docker;

    @Test
    public void should_create_valid_dockerfile() {
        DockerJavaAssertions.assertThat(docker).container
                ("videoservice").hasExposedPorts("8080/tcp")
                .isRunning();
    }

}
```

Validates the Docker container properties

This test uses Docker AssertJ integration. Although it isn't mandatory, we recommend using it to maintain your test's readability.

When executing this test, Arquillian Cube instructs the Docker host to build and run the given image. If the image can be built, the test verifies that ports that should be exposed are still exposed (nobody has changed them in the Dockerfile) and finally verifies that the container is running.

TIP You don't need to build the image every time. Instead, you can create the image once in the CD build and then reuse it for each kind of test.

Exercise

You should now be able to use Docker to set up a testing environment to run your tests. Using the video-service example, write a simple end-to-end test using REST Assured.

Summary

- You can use Docker for testing purposes by using it to set up test environments. You can also use it to test applications that use Docker in production.
- You can write tests for the UI using Docker and Selenium/Arquillian Graphene so that everything, including the browser, is containerized.
- The container-object pattern lets you create containers programmatically.
- Docker doesn't force you to use any specific language or framework. This means you can use Arquillian Cube to test any application written with any language, as long as it's Dockerized. This is a perfect match for a microservices architecture, where each microservice may be coded in a different language.

- If a microservice depends on an external service, you shouldn't set up the testing environment using the real external service (doing so could make your tests flaky); you can use service virtualization to simulate external services. Because WireMock is an HTTP server, you can containerize it and use it in Docker. This way, you can use Docker to test microservice(s) and cut dependencies at the level you need, and simulate the responses with a WireMock/Hoverfly container.

Chapter 9 from *Testing Java Microservices*
by Alex Soto Bueno, Andy Gumbrecht
and Jason Porter

Service virtualization

This chapter covers

- Appreciating service virtualization
- Simulating internal and external services
- Understanding service virtualization and Java

In a microservices architecture, the application as a whole can be composed of many interconnected services. These services can be either internal services, as in members of the same application domain, or external, third-party services that are totally out of your control.

As you've seen throughout the book, this approach implies that some changes are required when continuous application testing is part of your delivery pipeline. Back in chapter 7, we observed that one of the biggest challenges faced when testing a microservices architecture is having a clean test environment readily available. Getting multiple services up, running, and prepared is no trivial task. It takes time to prepare and execute tests, and it's highly likely that you'll end up with several flaky tests—tests that fail not due to code issues, but because of a failures within the testing environment. One of the techniques you can adopt to fix this is service virtualization.

9.1 What is service virtualization?

Service virtualization is a technique used to emulate the behavior of dependencies of component-based applications. Generally speaking, in a microservices architecture, these dependencies are usually REST API–based services, but the concept can also be applied to other kinds of dependencies such as databases, enterprise service buses (ESBs), web services, Java Message Service (JMS), or any other system that communicates using messaging protocols.

9.1.1 Why use service virtualization?

Following are several situations in which you might want to use service virtualization:

- When the current service (consumer) depends on another service (provider) that hasn't been developed yet or is still in development.
- When provisioning a new instance of the required service (provider) is difficult or too slow for testing purposes.
- When configuring the service (provider) isn't a trivial task. For example, you might need to prepare a huge number of database scripts for running the tests.
- When services need to be accessed in parallel by different teams that have completely different setups.
- When the provider service is controlled by a third party or partner, and you have a rate-limited quota on daily requests. You don't want to consume the quota with tests!
- When the provider service is available only intermittently or at certain times of the day or night.

Service virtualization can resolve all these challenges by simulating the behavior of the required service. With service virtualization, you model and deploy a *virtual asset* that represents the provider service, simulating the parts required for your test.

Figure 9.1 shows the difference between provisioning a real environment and a virtualized one for running tests. On the left, you can see that writing a test for service A requires that you boot up service B, including its database. At the same time, you might also need to start *transitive* services such as services C and D. On the right side, you can see that service B and all of its dependencies are replaced by a virtualized version that emulates the behavior of service B.

It's important to note that this diagram isn't much different than the one you saw when you learned about mocking and stubbing in chapter 3; but rather than simulating classes, you're simulating service calls. Streamlining that thought, you can imagine service virtualization as mocking at the enterprise level.

Service virtualization shouldn't be used only for testing optimal path cases, but also for testing edge cases, so that you test the entire application (negative testing). Sometimes it's difficult to test edge cases against real services. For example, you might want to test how a client behaves with low-latency responses from the provider, or how it acts when characters are sent with a different character encoding than expected.

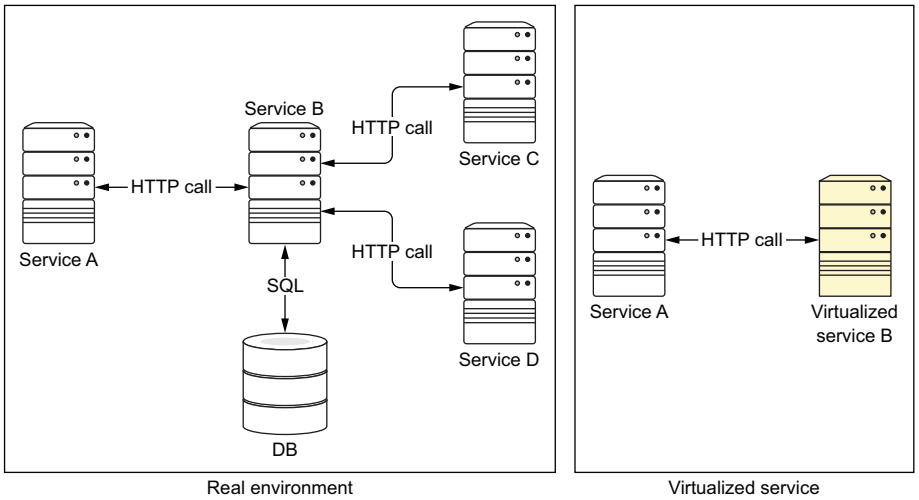


Figure 9.1 Real versus virtualized services

Think back to the Gamer application—you can't ask igdb.com and youtube.com to shut down their APIs for an afternoon while you perform negative testing. (Well, you could, but don't hold your breath waiting for an answer!) In such cases, it should be apparent why service virtualization is so useful.

9.1.2 When to use service virtualization

The book has presented many different kinds of tests, from unit tests to end-to-end tests. When is service virtualization useful?

- *Unit tests*—You're unlikely to need service virtualization for unit tests. In 99% of cases, using traditional mock, dummy, and stub techniques will be enough.
- *Component tests*—This is where service virtualization shines: you can test how components interact with each other without relying on external services.
- *Integration tests*—By their nature, integration tests are run against real services. In test cases, this might be a problem (such as edge cases, third-party services, and so on), so you might opt for service virtualization.
- *Contract tests*—When testing a contract against a provider, you might need service virtualization to simulate dependencies of the provider service.
- *End-to-end tests*—By definition, end-to-end tests shouldn't rely on service virtualization, because you're testing against the real system. In some rare cases where you relay on flaky third-party services, service virtualization might still be a viable solution.

As you can see, virtual assets are replaced by progressively more real services as you move to more functional tests.

In chapter 4, we discussed the concept of simulating external services with WireMock. In this chapter, we'll introduce a new tool called *Hoverfly*, which is designed specifically for service virtualization.

9.2 Mimicking service responses with Hoverfly

Hoverfly (https://hoverfly.readthedocs.io) is an open source, lightweight, service-virtualization proxy written in the Go programming language. It allows you to emulate HTTP and HTTPS services. As you can see in figure 9.2, Hoverfly starts a proxy that responds to requests with stored (canned) responses. These responses should be exactly the same as the ones the real service would generate for the provided requests. If this process is performed correctly, and if the stored responses are accurate for the real service, Hoverfly will mimic the real service responses perfectly, and your tests will be accurate.

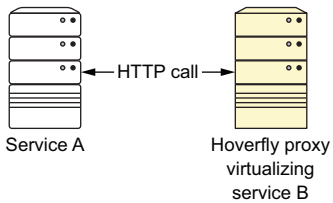


Figure 9.2 Hoverfly proxy

NOTE *Hoverfly Java* (https://hoverfly-java.readthedocs.io) is a Java wrapper for Hoverfly that abstracts you away from the actual binary and API calls, and also provides tight integration with JUnit. From this point on, when we talk about Hoverfly, we mean the Java wrapper.

9.2.1 Hoverfly modes

Hoverfly has three working modes:

- *Capture*—Makes requests against real services as normal. Requests and responses are intercepted and recorded by the Hoverfly proxy so they can be used later.
- *Simulate*—Returns simulated responses for the provided requests. Simulations might be loaded from different kinds of sources such as files, classpath resources, or URLs, or programmatically defined using the Hoverfly domain-specific language (DSL). This is the preferred mode for services under development.
- *Capture or simulate*—A combination of the other two modes. The proxy starts in capture mode if the simulation file doesn't exist, or in simulate mode otherwise. This mode is preferred when already developed services or third-party services are available.

Figure 9.3 shows a schema for capture mode:

1. A request is performed using a real service, which is probably deployed outside of the machine where the test is running.
2. The Hoverfly proxy redirects traffic to the real host, and the response is returned.
3. The Hoverfly proxy stores a script file for the matching request and response that were generated by the real service interaction.
4. The real response is returned to the caller.

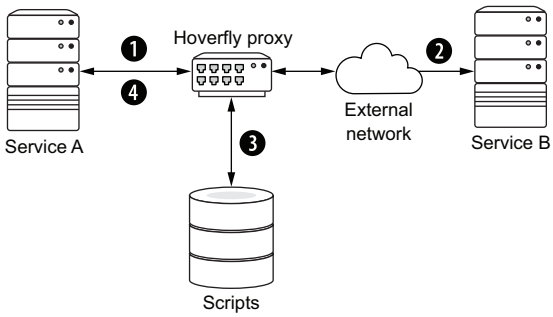


Figure 9.2 Hoverfly capture mode

Figure 9.4 illustrates simulate mode:

1. A request is performed, but instead the call being routed to the real service, it's routed to the Hoverfly proxy.
2. The Hoverfly proxy checks the corresponding response script for the provided request.
3. A canned response is replayed back to the caller.

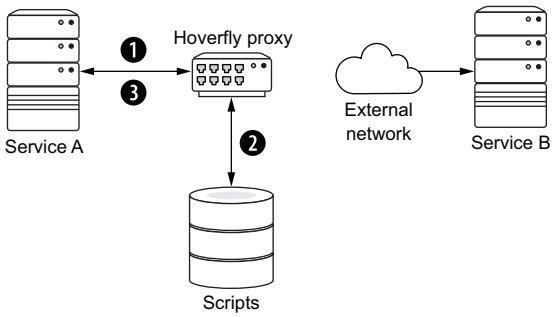


Figure 9.3 Hoverfly simulate mode

Hoverfly and JVM proxy settings

Hoverfly Java sets the network Java system properties to use the Hoverfly proxy. This means if the client API you're using to communicate with other services honors these properties, you don't need to change anything to work with Hoverfly. If that isn't the case, you need to set `http.proxyHost`, `http.proxyPort`, `https.proxyHost`, `https.proxyPort`, and, optionally, `http.nonProxyHosts` to your client proxy configuration.

When this override is in place, all communication between the Java runtime and the physical network (except `localhost` by default) will pass through the Hoverfly proxy. For example, when using the `okhttp` client, which honors network system properties, you might do this:

```
URL url = new URL("http", "www.myexample.com", 8080, "/" + name);
Request request = new Request.Builder().url(url).get().build();
final Response response = client.newCall(request).execute();
```

(continued)

Because the proxy settings are now overridden, the request is performed through the Hoverfly proxy. Depending on the selected configuration mode, the request will either be sent to www.myexample.com or simulated.

9.2.2 JUnit Hoverfly

Let's look at some examples of how to use Hoverfly with JUnit.

JUnit Hoverfly simulate mode

Hoverfly comes in the form of a JUnit rule. You can use either `@ClassRule` for static initialization, or `@Rule` for each test. We recommend using `@ClassRule`, to avoid the overhead of starting the Hoverfly proxy for each test method execution. Here's an example:

```
import static io.specto.hoverfly.junit.core.SimulationSource.defaultPath;
import io.specto.hoverfly.junit.rule.HoverflyRule;

@ClassRule
public static HoverflyRule hoverflyRule = HoverflyRule
                    .inSimulationMode(defaultPath("simulation.json"));
```

Reads simulation.json from the default Hoverfly resource path

Here, the Hoverfly proxy is started, and it then loads the simulation.json simulation file from the default Hoverfly resource path, src/test/resources/hoverfly. After that, all tests are executed, and the Hoverfly proxy is stopped.

In addition to loading simulations from a file, you can specify request matchers and responses using the DSL, as shown here:

```
import static io.specto.hoverfly.junit.core.SimulationSource.dsl;
import static io.specto.hoverfly.junit.dsl.HoverflyDsl.service;
import static io.specto.hoverfly.junit.dsl.ResponseCreators.success;
import static io.specto.hoverfly.junit.dsl.ResponseCreators.created;

import io.specto.hoverfly.junit.rule.HoverflyRule;

@ClassRule
public static HoverflyRule hoverflyRule =
HoverflyRule.inSimulationMode(dsl(
    service("www.myexample.com")
            .post("/api/games").body("{\"gameId\": \"1\"}")
            .willReturn(created("http://localhost/api/game/1"))

            .get("/api/games/1")
            .willReturn(success("{\"gameId\":\"1\"\}", "application/json"))
    ));
```

Starts Hoverfly using the DSL method

Creates a request and a response for a POST method

Sets the host where the connection is to be made

Creates a request and a response for a GET method

Request-field matchers

Hoverfly has the concept of *request-field matchers*, which let you use different kinds of matchers in the DSL elements. Here's an example:

```
service(matches("www.*-test.com"))
        .get(startsWith("/api/games/"))
        .queryParam("page", any())
```

Matches the URL using a wildcard

Matches the request path that starts with /api/games/

Matches the page query parameter with any value

JUNIT HOVERFLY CAPTURE MODE

Starting Hoverfly in capture mode is the same as it is in simulate mode, but you use `inCaptureMode` to indicate that you want to store the interaction:

```
@ClassRule
public static HoverflyRule hoverflyRule
    = HoverflyRule.inCaptureMode("simulation.json");
```

Starts Hoverfly in capture mode, and records the result

In this example, Hoverfly is started in capture mode. This effectively means the traffic is redirected/routed to the real service, but now these interactions are recorded in a file located by default at src/test/resources/hoverfly/simulation.json.

JUNIT HOVERFLY CAPTURE OR SIMULATE MODE

This mode is the combination of both previous modes, using capture mode if no previously recorded file is present. The generated files can then be added to your version control to complete the test case for others to use without the real service. Here's an example:

```
@ClassRule
public static HoverflyRule hoverflyRule
    = HoverflyRule.inCaptureOrSimulationMode("simulation.json");
```

9.2.3 Configuring Hoverfly

Hoverfly ships with defaults that may work in all cases, but you can override them by providing an `io.specto.hoverfly.junit.core.HoverflyConfig` instance to the previous methods. For example, you can change the proxy port where the Hoverfly proxy is started by setting `inCaptureMode("simulation.json", HoverflyConfig.configs() .proxyPort(8080))`.

By default, all hostnames are proxied, but you can also restrict this behavior to specific hostnames. For example, `configs().destination("www.myexample.com")` configures the Hoverfly proxy to only process requests to www.myexample.com.

Localhost calls are *not* proxied by default. But if your provider service is running on localhost, you can configure Hoverfly to proxy localhost calls by using `configs().proxyLocalHost()`.

Configuring SSL

If your service uses Secure Sockets Layer (SSL), Hoverfly needs to decrypt the messages in order to persist them to a file in capture mode, or to perform the matching in simulate mode. Effectively, you have one SSL connection between the client and the Hoverfly proxy, and another between the Hoverfly proxy and the real service.

To make things simple, Hoverfly comes with its own self-signed certificate that must be trusted by your client. The good news is that Hoverfly's certificate is trusted automatically when you instantiate it.

You can override this behavior and provide your own certificate and key using the `HoverflyConfig` class: for example, `configs().sslCertificatePath("ssl/ca.crt").sslKeyPath("ssl/ca.key")`. Note that these files are relative to the classpath.

Configuring an external instance

It's possible to configure Hoverfly to use an existing Hoverfly proxy instance. This situation might arise when you're using a Docker image hosting a Hoverfly proxy. Again, you can configure these parameters easily by using the `HoverflyConfig` class: for example, `configs().remote().host("192.168.99.100").proxyPort(8081)`.

9.3 Build-script modifications

Now that you've learned about service virtualization and Hoverfly, let's look at the involved dependencies. Hoverfly requires only a single group, artifact, version (GAV) dependency definition:

```
dependencies {
  testCompile "io.specto:hoverfly-java:0.6.2"
}
```

This pulls in all the required transient dependencies to the test scope.

9.4 Using service virtualization for the Gamer application

As you've seen throughout the book, in the Gamer application, the aggregator service communicates with three services to compose the final request to the end user with all information about games, as shown in figure 9.5. Let's write a component test for the code that connects to the comments service.

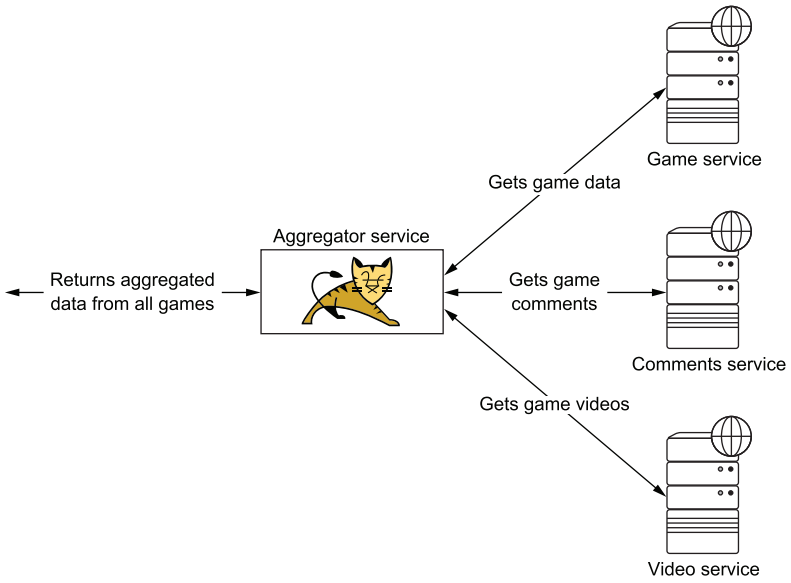


Figure 9.4 The aggregator service

In the following listing (code/aggregator/cp-tests/src/test/java/book/aggr/CommentsGatewayTest.java), the comments service is deployed in (pre)production at comments.gamers.com, and you'll use capture or simulate mode so that initial requests are sent to the real service. All subsequent calls will be simulated.

Listing 9.1 Testing the `CommentsGateway` class

```
public class CommentsGatewayTest {

    @ClassRule
    public static HoverflyRule hoverfly = HoverflyRule
        .inCaptureOrSimulationMode("simulation.json"); /

    @Test
    public void shouldInsertComments()
        throws ExecutionException, InterruptedException {

        final JsonObject commentObject = Json.createObjectBuilder()
            .add("comment", "This Game is Awesome").add("rate",
                5).add("gameId", 1234).build();

        final CommentsGateway commentsGateway = new CommentsGateway
            ();
        commentsGateway.initRestClient("http://comments.gamers.com")
        ;

        final Future<Response>; comment = commentsGateway
            .createComment(commentObject);
```

Instantiates the Hoverfly rule

Makes the call to the real host

```
        final Response response = comment.get();
        final URI location = response.getLocation();

        assertThat(location).isNotNull();
        final String id = extractId(location);
        assertThat(id).matches("[0-9a-f]+");
    }
```

Asserts that the location is valid

The big difference between this and the other test cases is that the first time you run this test, the request is sent to the comments service deployed at comments.gamers.com via the Hoverfly proxy, and requests and responses are recorded. The src/test/resources/hoverfly/simulation.json file is created because it doesn't yet exist. The next time you run the test, communication is still proxied through the Hoverfly proxy, but because the file now exists, the canned responses are returned.

In case you're curious (we know you are), the recorded file looks like the next listing (src/test/resources/hoverfly/simulation.json).

Listing 9.2 Simulation file with canned responses

```
{
  "data" : {
    "pairs" : [ {
      "request" : {
        "path" : {
          "exactMatch" : "/comments"
        },
        "method" : {
          "exactMatch" : "POST"
        },
        "destination" : {"exactMatch" : "comments.gamers.com"},
        "scheme" : {
          "exactMatch" : "http"
        },
        "query" : {"exactMatch" : ""},
        "body" : {
          "jsonMatch" : "{\"comment\":\"This Game is Awesome\",
                        \"rate\":5,\"gameId\":1234}"
        }
      },
      "response" : {
        "status" : 201,
        "encodedBody" : false,
        "headers" : {
          "Content-Length" : [ "0" ],
          "Date" : [ "Thu, 15 Jun 2017 17:51:17 GMT" ],
          "Hoverfly" : [ "Was-Here" ],
          "Location" : [ "comments.gamers.com/5942c915c9e77c0001454df1" ],
          "Server" : [ "Apache TomEE" ]
        }
      }
    } ],
```

```
    "globalActions" : {
      "delays" : [ ]
    }
  },
  "meta" : {
    "schemaVersion" : "v2"
  }
}
```

Summary

- Service virtualization isn't a substitute for contract tests but something to use with them, mostly in provider-validation scenarios.
- Service virtualization is used for removing the flakiness of tests that depend on external and potentially unreliable services.
- A virtual asset is the service you're simulating.
- You can use service virtualization to emulate unfinished services in addition to existing services, thus allowing rapid development in parallel teams.
- Hoverfly Java takes care of all network redirections and lets you get on with writing the test.

index

A

B

C

D

E

F

G

H

T

U

V

W

X


manning