Using Arquillian, Hoverfly, AssertJ, JUnit, Selenium, and Mockito

Testing Java Microservices

Alex Soto Bueno
Andy Gumbrecht
Jason Porter

SAMPLE CHAPTER

MANNING

Testing Java Microservices

by Alex Soto Bueno,
Jason Porter,
and Andy Gumbrecht

Chapter 9

brief contents

1 ■ An introduction to microservices 1
2 ■ Application under test 13
3 ■ Unit-testing microservices 41
4 ■ Component-testing microservices 64
5 ■ Integration-testing microservices 100
6 ■ Contract tests 128
7 ■ End-to-end testing 164
8 ■ Docker and testing 190
9 ■ Service virtualization 233
10 ■ Continuous delivery in microservices 244

9 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.

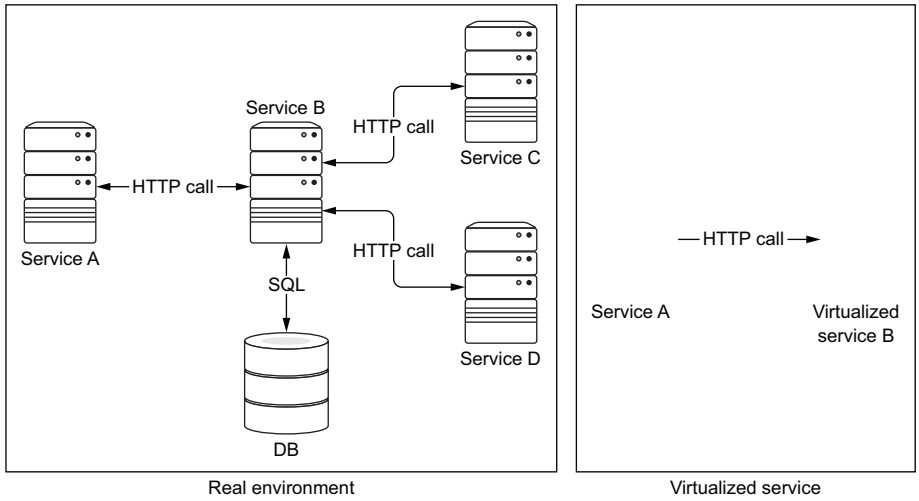

Figure 9.1 Real versus virtualized services

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.

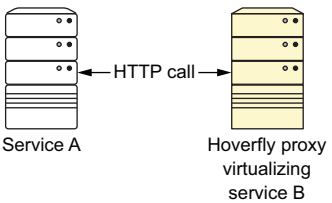

Figure 9.2 Hoverfly proxy

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.

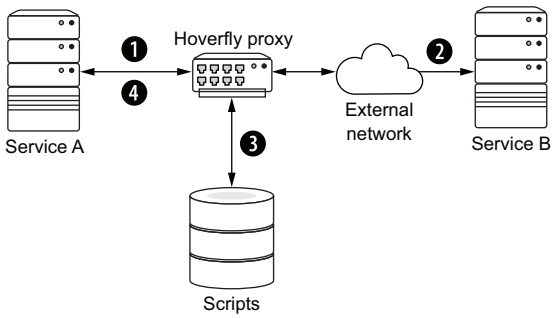

Figure 9.3 Hoverfly capture mode

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.

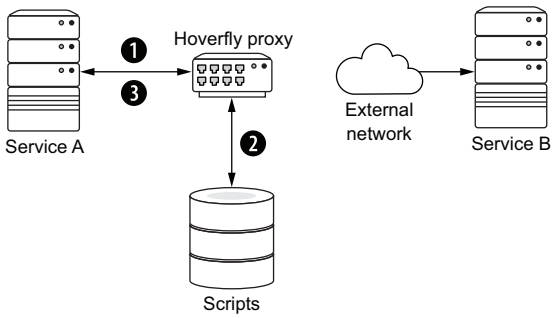

Figure 9.4 Hoverfly simulate mode

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.

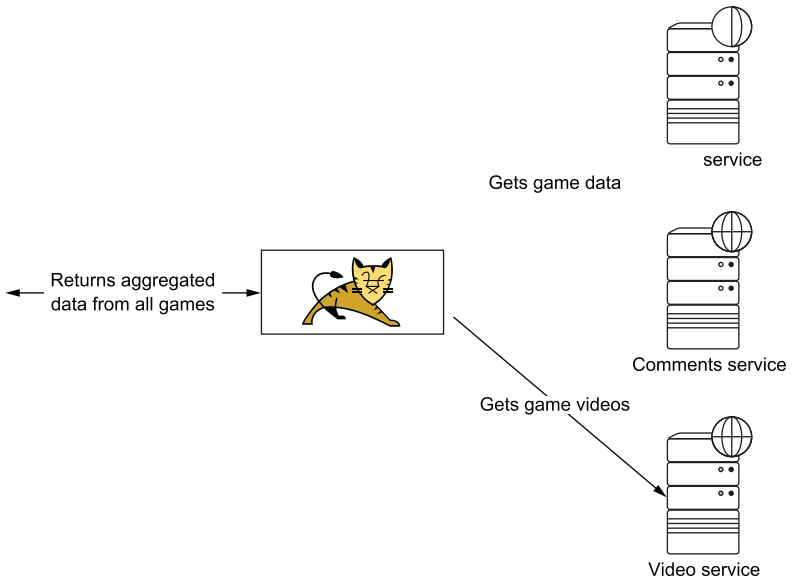

Figure 9.5 The aggregator service

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.

SOFTWARE DEVELOPMENT/JAVA

Testing Java Microservices

Soto Bueno • Gumbrecht • Porter

Microservice applications present special testing challenges. Even simple services need to handle unpredictable loads, and distributed message-based designs pose unique security and performance concerns. These challenges increase when you throw in asynchronous communication and containers.

Testing Java Microservices teaches you to implement unit and integration tests for microservice systems running on the JVM. You'll work with a microservice environment built using Java EE, WildFly Swarm, and Docker. You'll advance from writing simple unit tests for individual services to more-advanced practices like chaos or integration tests. As you move towards a continuous-delivery pipeline, you'll also master live system testing using technologies like the Arquillian, Wiremock, and Mockito frameworks, along with techniques like contract testing and over-the-wire service virtualization. Master these microservice-specific practices and tools and you'll greatly increase your test coverage and productivity, and gain confidence that your system will work as you expect.

What's Inside

- Test automation
- Integration testing microservice systems
- Testing container-centric systems
- Service virtualization

Written for Java developers familiar with Java EE, EE4J, Spring, or Spring Boot.

Alex Soto Bueno and **Jason Porter** are Arquillian team members. **Andy Gumbrecht** is an Apache TomEE developer and PMC. They all have extensive enterprise-testing experience.

To download their free eBook in PDF, ePub, and Kindle formats, owners of this book should visit manning.com/books/testing-java-microservices

"A great and invaluable gallery of test solutions, descriptions, and examples."
—Gualtiero Testa, Factor-y

"Covers the full spectrum of microservice-testing techniques. An indispensable book."
—Piotr Gliniewicz
netPR.pl

"Thorough explanations. Concrete examples. Real-world technology."
—Ethan Rivett, Powerley

"A beacon of light for Java developers."
—Yagiz Erkan, Motive Retail

ISBN-13: 978-1-61729-289-7
ISBN-10: 1-61729-289-3
54499
9 781617 292897

MANNING $44.99 / Can $59.99 [INCLUDING eBOOK]