Hoverfly Java Documentation

Release 0.5.0

SpectoLabs

May 18, 2017

Contents

Hoverfly is a lightweight service virtualisation tool which allows you to stub / simulate HTTP(S) services. It is a proxy written in Go which responds to HTTP(S) requests with stored responses, pretending to be it's real counterpart.

It enables you to get around common testing problems caused by external dependencies, such as non-deterministic data, flakiness, not yet implemented API's, licensing fees, slow tests and more.

Hoverfly Java is a native language binding which gives you an expressive API for managing Hoverfly in Java. It gives you a Hoverfly class which abstracts away the binary and API calls, a *DSL* for creating simulations, and a *JUnit* integration for using it within JUnit tests.

Hoverfly Java is developed and maintained by SpectoLabs.

CHAPTER 1

Quickstart

Maven

If using Maven, add the following dependency to your pom:

```
<dependency>
    <groupId>io.specto</groupId>
    <artifactId>hoverfly-java</artifactId>
    <version>0.5.0</version>
    <scope>test</scope>
</dependency>
```

Gradle

Or with Gradle add the dependency to your *.gradle file:

```
testCompile ``io.specto:hoverfly-java:0.5.0``
```

Code example

The simplest way is to get started is with the JUnit rule. Behind the scenes the JVM proxy settings will be configured to use the managed Hoverfly process, so you can just make requests as normal, only this time Hoverfly will respond instead of the real service (assuming your HTTP client respects JVM proxy settings):

```
public class HoverflyExample {

    @ClassRule
    public static HoverflyRule hoverflyRule = HoverflyRule.inSimulationMode(dsl(
        service("www.my-test.com")
            .get("/api/bookings/1")
```

```
            .willReturn(success("{\"bookingId\":\"1\"}", "application/json"))
    ));

    @Test
    public void shouldBeAbleToGetABookingUsingHoverfly() {
        // When
        final ResponseEntity<String> getBookingResponse = restTemplate.getForEntity(
↪"http://www.my-test.com/api/bookings/1", String.class);

        // Then
        assertThat(getBookingResponse.getStatusCode()).isEqualTo(OK);
        assertThatJSON(getBookingResponse.getBody()).isEqualTo("{\"bookingId\":\"1\"}
↪");
    }

// Continues...
```

Contents

Quickstart

Maven

If using Maven, add the following dependency to your pom:

```
<dependency>
    <groupId>io.specto</groupId>
    <artifactId>hoverfly-java</artifactId>
    <version>0.5.0</version>
    <scope>test</scope>
</dependency>
```

Gradle

Or with Gradle add the dependency to your *.gradle file:

```
testCompile ``io.specto:hoverfly-java:0.5.0``
```

Code example

The simplest way is to get started is with the JUnit rule. Behind the scenes the JVM proxy settings will be configured to use the managed Hoverfly process, so you can just make requests as normal, only this time Hoverfly will respond instead of the real service (assuming your HTTP client respects JVM proxy settings):

```
public class HoverflyExample {

    @ClassRule
    public static HoverflyRule hoverflyRule = HoverflyRule.inSimulationMode(dsl(
        service("www.my-test.com")
```

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

    @Test
    public void shouldBeAbleToGetABookingUsingHoverfly() {
        // When
        final ResponseEntity<String> getBookingResponse = restTemplate.getForEntity(
↪"http://www.my-test.com/api/bookings/1", String.class);

        // Then
        assertThat(getBookingResponse.getStatusCode()).isEqualTo(OK);
        assertThatJSON(getBookingResponse.getBody()).isEqualTo("{"\"bookingId\":\"1\"}
↪");
    }

// Continues...
```

Core functionality

This section describes the core functionality of Hoverfly Java.

Simulating

The core of this library is the Hoverfly class, which abstracts away and orchestrates a Hoverfly instance. A flow might be as follows:

```
try (Hoverfly hoverfly = new Hoverfly(configs(), SIMULATE)) {

    hoverfly.start();
    hoverfly.importSimulation(classpath("simulation.json"));

    // do some requests here
}
```

When running Hoverfly standalone you can clean it by calling *reset* method.

Capturing

The previous examples have only used Hoverfly in simulate mode. You can also run it in capture mode, meaning that requests will be made to the real service as normal, only they will be intercepted and recorded by Hoverfly. This can be a simple way of breaking a test's dependency on an external service; wait until you have a green test, then switch back into simulate mode using the simulation data recorded during capture mode.

```
try(Hoverfly hoverfly = new Hoverfly(configs(), CAPTURE)) {

    hoverfly.start();

    // do some requests here

    hoverfly.exportSimulation(Paths.get("some-path/simulation.json"));
}
```

Sources

There are a few different potential sources for Simulations:

```
SimulationSource.classpath("simulation.json"); //classpath
SimulationSource.defaultPath("simulation.json"); //default hoverfly resource path␣
↪which is src/test/resources/hoverfly
SimulationSource.url("http://www.my-service.com/simulation.json"); // URL
SimulationSource.url(new URL("http://www.my-service.com/simulation.json")); // URL
SimulationSource.file(Paths.get("src", "simulation.json")); // File
SimulationSource.dsl(service("www.foo.com").get("/bar).willReturn(success())); //␣
↪Object
SimulationSource.simulation(new Simulation()); // Object
SimulationSource.empty(); // None
```

DSL

The rule now has fluent DSL which allows you to build request matcher to response mappings in Java opposed to importing them as JSON.

The rule is fluent and hierarchical, allowing you to define multiple service endpoints as follows:

```
simulationSource.dsl(
    service("www.my-test.com")

        .post("/api/bookings").body("{\"flightId\": \"1\"}")
        .willReturn(created("http://localhost/api/bookings/1"))

        .get("/api/bookings/1")
        .willReturn(success("{\"bookingId\":\"1\"\}", "application/json")),

    service("www.anotherService.com")

        .put("/api/bookings/1").body(json(new Booking("foo", "bar")))
        .willReturn(success())

        .delete("/api/bookings/1")
        .willReturn(noContent())
)
```

The entry point for the DSL is *HoverflyDSL.service*. After calling this you can provide a *method* and *path*, followed by optional request components. You can then use *willReturn* to state which response you want when there is a match, which takes *responseBuilder* object that you can instantiate directly, or via the helper class *ResponseCreators*.

You can also simulate fixed network delay using DSL.

Global delays can be set for all requests or for a particular HTTP method:

```
simulationSource.dsl(
    service("www.slow-service.com")
        .andDelay(3, TimeUnit.SECONDS).forAll(),

    service("www.other-slow-service.com")
        .andDelay(3, TimeUnit.SECONDS).forMethod("POST")
)
```

Per-request delay can be set as follows:

```
simulationSource.dsl(
    service("www.not-so-slow-service.com")
        .get("/api/bookings")
        .willReturn(success().withDelay(1, TimeUnit.SECONDS))
    )
)
```

Conversion

There is currently a *BodyConverter* interface which can be used to serialise Java objects into strings, and also set a content type header automatically

```
.body(json(new JsonObject("foo", "bar"))) // default
.body(json(new JsonObject("foo", "bar"), myObjectMapper)) // Object mapper configured
```

Configuration

Hoverfly takes a config object, which contains sensible defaults if not configured. Ports will be randomised to unused ones, which is useful on something like a CI server if you want to avoid port clashes. You can also set fixed port:

```
configs().proxyPort(8080)
```

You can configure Hoverfly to process requests to certain destinations / hostnames

```
configs().destination("www.test.com") // only process requests to www.test.com
configs().destination("api") // matches destination that contains api, eg. api.test.
↪com
```

You can configure Hoverfly to proxy localhost requests. This is useful if the target server you are trying to simulate is running on localhost.

```
configs().proxyLocalHost()
```

SSL

When requests pass through Hoverfly, it needs to decrypt them in order for it to persist them to a database, or to perform matching. So you end up with SSL between Hoverfly and the external service, and then SSL again between your client and Hoverfly. To get this to work, Hoverfly comes with it's own self-signed certificate which has to be trusted by your client. To avoid the pain of configuring your keystore, Hoverfly's certificate is trusted automatically when you instantiate it.

Alternatively, you can override the default SSL certificate by providing your own certificate and key files via the HoverflyConfig object, for example:

```
configs()
    .sslCertificatePath("ssl/ca.crt")
    .sslKeyPath("ssl/ca.key");
```

The input to these config options should be the file path relative to the classpath. Any PEM encoded certificate and key files are supported.

If the default SSL certificate is overridden, hoverfly-java will not automatically set it trusted, and it is the users' responsibility to configure SSL context for their HTTPS client.

Using externally managed instance

It is possible to configure Hoverfly to use an existing API simulation managed externally. This could be a private Hoverfly cluster for sharing API simulations across teams, or a publicly available API sandbox powered by Hoverfly.

You can enable this feature easily with the configs fluent builder. The default settings point to localhost on default admin port 8888 and proxy port 8500.

```
configs().remote()
```

You can point it to other host and ports

```
configs()
    .remote()
    .host("10.0.0.1")
    .adminPort(8080)
    .proxyPort(8081)
```

Depends on the set up of the remote Hoverfly instance, it may require additional security configurations.

You can provide a custom CA certificate for the proxy.

```
configs()
    .remote()
    .proxyCaCert("ca.pem") // the name of the file relative to classpath
```

You can configure Hoverfly to use an HTTPS admin endpoint.

```
configs()
    .remote()
    .withHttpsAdminEndpoint()
```

You can provide the token for the custom Hoverfly authorization header, this will be used for both proxy and admin endpoint authentication without the need for username and password.

```
configs()
    .remote()
    .withAuthHeader() // this will get auth token from an environment variable named
↪'HOVERFLY_AUTH_TOKEN'

configs()
    .remote()
    .withAuthHeader("some.token") // pass in token directly
```

JUnit

An easier way to orchestrate Hoverfly is via the JUnit Rule. This is because it will create destroy the process for you automatically, doing any cleanup work and auto-importing / exporting if required.

Simulate

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

Capture

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

File is relative to `src/test/resources/hoverfly`.

Multi-Capture

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

public void someTest() {
    hoverflyRule.capture("firstScenario.json");

    // test
}

public void someTest() {
    hoverflyRule.capture("someOtherScenario.json");

    // test
}
```

File is relative to `src/test/resources/hoverfly`.

Capture or Simulate

You can create a Hoverfly Rule that is started in capture mode if the simulation file does not exist and in simulate mode if the file does exist. File is relative to `src/test/resources/hoverfly`.

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

Use @ClassRule

It is recommended to boot Hoverfly once and share it across multiple tests by using a *@ClassRule* rather than *@Rule*. This means you don't have the overhead of starting one process per test, and also guarantees that all your system properties are set correctly before executing any of your test code.

Miscellaneous

Apache HttpClient

This doesn't respect JVM system properties for things such as the proxy and truststore settings. Therefore when you build one you would need to:

```
HttpClient httpClient = HttpClients.createSystem();
// or
HttpClient httpClient = HttpClientBuilder.create().useSystemProperties().build();
```

Or on older versions you may need to:

```
HttpClient httpClient = new SystemDefaultHttpClient();
```

In addition, Hoverfly should be initialized before Apache HttpClient to ensure that the relevant JVM system properties are set before they are used by Apache library to configure the HttpClient.

There are several options to achieve this:

- Use *@ClassRule* and it guarantees that *HoverflyRule* is executed at the very start and end of the test case
- If using *@Rule* is inevitable, you should initialize the HttpClient inside your *@Before* setUp method which will be executed after *@Rule*
- As a last resort, you may want to manually configured Apache HttpClient to use custom proxy or SSL context, please check out HttpClient examples

Legacy Schema Migration

If you have recorded data in the legacy schema generated before hoverfly-junit v0.1.9, you will need to run the following commands using Hoverfly to migrate to the new schema:

```
$ hoverctl start
$ hoverctl delete simulations
$ hoverctl import --v1 path-to-my-json/file.json
$ hoverctl export path-to-my-json/file.json
$ hoverctl stop
```

V1 to V2 Schema Migration

Starting from Hoverfly-java v0.5.0, the simulation schema is upgraded to v2 which supports matchers. Although it is backward compatible with v1, upgrading to v2 is recommended:

```
$ hoverctl start
$ hoverctl delete simulations
$ hoverctl import path-to-my-json/file.json
$ hoverctl export path-to-my-json/file.json
$ hoverctl stop
```

Using Snapshot Version

To use snapshot version, you should include the OSS snapshot repository in your build file.

If using Maven, add the following repository to your pom:

```
<repositories>
    <repository>
        <id>oss-snapshots</id>
            <name>OSS Snapshots</name>
        <url>https://oss.sonatype.org/content/repositories/snapshots</url>
        <snapshots>
            <enabled>true</enabled>
            </snapshots>
    </repository>
</repositories>
```

Or with Gradle add the repository to your build.gradle file:

```
repositories {
    maven {
        url 'https://oss.sonatype.org/content/repositories/snapshots'
    }
}
```