Hoverfly Java Documentation

Release 0.12.2

SpectoLabs

Aug 26, 2019

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

1.1 Maven

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

```
<dependency>
    <groupId>io.specto</groupId>
    <artifactId>hoverfly-java</artifactId>
    <version>0.12.2</version>
    <scope>test</scope>
</dependency>
```

1.2 Gradle

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

```
testCompile ``io.specto:hoverfly-java:0.12.2``
```

1.3 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):

```
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;

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...
```

1.4 Contents

1.4.1 Quickstart

Maven

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

```
<dependency>
    <groupId>io.specto</groupId>
    <artifactId>hoverfly-java</artifactId>
    <version>0.12.2</version>
    <scope>test</scope>
</dependency>
```

Gradle

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

```
testCompile ``io.specto:hoverfly-java:0.12.2``
```

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):

```
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;

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...
```

1.4.2 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 up the simulation and journal logs by calling `reset` method.

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

    hoverfly.start();
    // can import or do some requests

    hoverfly.reset();
}
```

Spying

In Spy mode, Hoverfly calls the real service if a request is not matched, otherwise, it simulates.

This mode allows you to spy on some of the endpoints of the services your test depends on, while letting other traffic to pass through.

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

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

    // do some requests here
}
```

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"));
}
```

By default Hoverfly captures multiple identical requests once only, but you can set the following config to enable stateful capture which should capture all requests sequentially.

```
Hoverfly hoverfly = new Hoverfly(localConfigs().enableStatefulCapture(), CAPTURE)
```

Diffing

It is possible to set Hoverfly in Diff mode to detect the differences between a simulation and the the actual requests and responses. When Hoverfly is in Diff mode, it forwards the requests and serves responses from real service, and at the meantime, generates a diff report that stores in memory. You can later on call the `assertThatNoDiffIsReported` function to verify if any discrepancy is detected.

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

    hoverfly.start();
    hoverfly.importSimulation(classpath("simulation-to-compare.json"));

    // do some requests here

    hoverfly.assertThatNoDiffIsReported(false);

}
```

If you pass `true` to `assertThatNoDiffIsReported`, it will instruct Hoverfly to reset the diff logs after the assertion.

Simulation source

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
```

You can pass in multiple sources when importing simulations, for instance, if you need to combine simulations from previous capture session and ones that created via DSL:

```
hoverfly.simulate(
            classpath("test-service.json"),
            dsl(service("www.my-test.com")
                    .post("/api/bookings").body("{\"flightId\": \"1\"}")
                    .willReturn(created("http://localhost/api/bookings/1")))
    );
```

DSL

Hoverfly Java has a DSL which allows you to build request matcher to response mappings in Java instead of importing them as JSON.

The DSL 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`.

Simulate network delay

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))
    )
)
```

Request/response body conversion

There is currently an `HttpBodyConverter` interface which can be used to marshall Java objects into strings, and also set a content type header automatically.

It can be used for both request and response body, and supports JSON and XML data format out-of-the-box.

```
// For request body matcher
.body(equalsToJson(json(myObject)))     // with default objectMapper
.body(equalsToJson(json(myObject, myObjectMapper)))     // with custom objectMapper

// For response body
.body(xml(myObject))
.body(xml(myObject, myObjectMapper))
```

There is an implementation which lets you write inline JSON body efficiently with single quotes.

```
.body(jsonWithSingleQuotes("{'bookingId':'1'}"))
.body(jsonWithSingleQuotes("{'merchantName':'Jame\\'s'}"))  // escape single quote in␣
↪your data if necessary
```

Request field matchers

Hoverfly-Java abstracts away some of the complexity of building the request field matchers supported by Hoverfly.

By default, the DSL request builder assumes exact matching when you pass in a string.

You can also pass in a matcher created by the `HoverflyMatchers` factory class.

Here are some examples:

```
SimulationSource.dsl(

    service(matches("www.*-test.com"))           // Matches url with wildcard
        .get(startsWith("/api/bookings/"))       // Matches request path that starts␣
↪with /api/bookings/
        .queryParam("page", any())               // Matches page query with any value
        .willReturn(success(json(booking)))

        .put("/api/bookings/1")
        .body(equalsToJson(json(booking)))       // Matches body which equals to a␣
↪JSON Object
        .willReturn(success())

        .put("/api/bookings/1")
        .body(matchesPartialJson(json(booking)))// Matches body which is a superset␣
↪of a JSON Object (Partial JSON matching)
        .willReturn(success())

        .post("/api/bookings")
        .body(matchesJsonPath("$.flightId"))     // Matches body with a JSON path␣
↪expression
        .willReturn(created("http://localhost/api/bookings/1"))

        .put("/api/bookings/1")
        .body(equalsToXml(xml(booking)))         // Matches body which equals to a XML␣
↪object
        .willReturn(success())

        // XmlPath Matcher
        .post("/api/bookings")
        .body(matchesXPath("/flightId"))         // Matches body with a xpath␣
↪expression
        .willReturn(created("http://localhost/api/bookings/1"))
)
```

`HoverflyMatchers` also provides the following matching methods:

```
HoverflyMatchers.contains("foo")
HoverflyMatchers.endsWith("foo")
HoverflyMatchers.startsWith("foo")

// Special matchers
HoverflyMatchers.matches("*foo*") // matches GLOB pattern
HoverflyMatchers.matchesGoRegex("[xyz]") // matches Golang regex pattern
```

Fuzzy matching is possible for request method, query and body with these simple built-in DSL methods:

```
SimulationSource.dsl(
    service("www.booking-is-down.com")
        .anyMethod(any())
        .anyQueryParams()
        .anyBody()
        .willReturn(serverError().body("booking is down"))
)
```

Headers are not used for matching unless they are specified. If you need to set a header to match on, use the `header` method:

```
SimulationSource.dsl(
    service("www.my-test.com")
        .post("/api/bookings")
        .body("{\"flightId\": \"1\"}")
        .header("Content-Type", any())      // Count as a match when request contains␣
↪this Content-Type header
        .willReturn(created("http://localhost/api/bookings/1"))
)
```

When you supply a simulation source to the `HoverflyRule`, you can enable the `printSimulationData` option to help debugging. This will print the simulation JSON data used by Hoverfly to `stdout`:

```
@ClassRule
public static HoverflyRule hoverflyRule = HoverflyRule.
↪inSimulationMode(simulationSource)
    .printSimulationData();
```

Response Templating

If you need to build a response dynamically based on the request data, you can do so using templating:

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

        // Using path in template
        .get("/api/bookings/1")
        .willReturn(success().body(jsonWithSingleQuotes(
                "{'id':{{ Request.Path.[2] }},'origin':'London','destination':
↪'Singapore','_links':{'self':{'href':'http://localhost/api/bookings/{{ Request.Path.
↪[2] }}'}}}"
        )))

        // Using query Param in template
        .get("/api/bookings")
            .queryParam("destination", "London")
            .queryParam("page", any())
        .willReturn(success().body(jsonWithSingleQuotes(
                "{'id':'1', 'destination':'{{ Request.QueryParam.destination }}','_
↪links':{'self':{'href':'http://localhost/api/bookings?page={{ Request.QueryParam.
↪page }}'}}}"
        )))));
```

The first example sets the id in response body based on the last path segment in the request using `{{ Request.Path.[2] }}`. The second example sets some response body data based on the request param using `{{ Request.QueryParam.destination }}` and `{{ Request.QueryParam.page }}`.

For more details, please check out the Hoverfly documentation.

Stateful simulation

Hoverfly Java supports stateful simulation which means a mocked service could return different responses for the same request depends on the current state(s). (for more details, please check out Hoverfly State).

Stateful simulation can be setup via DSL. A simple example is that you can simulate a GET booking API to return a good response if state is not set, but calling the DELETE API will trigger the state transition, and any subsequent requests to the GET API will results a 404 error.

```
SimulationSource.dsl(
    service("www.service-with-state.com")

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

        .delete("/api/bookings/1")
        .willReturn(success().andSetState("Booking", "Deleted"))

        .get("/api/bookings/1")
        .withState("Booking", "Deleted")
        .willReturn(notFound())
)
```

The following state control methods are available:

- `withState(key, value)` add a condition for the request to be matched based on the given state (which is a key-value pair)
- `andSetState(key, value)` applies to the response to set the state
- `andRemoveState(key)` can be applied to the response to remove a state by key.

You can also chain multiple state methods to create more complex scenarios.

Verification

This feature lets you verify that specific requests have been made to the external service endpoints.

You can do request verification by calling the `verify` method from `HoverflyRule`. It accepts two arguments. The first one is a `RequestMatcherBuilder` which is also used by the DSL for creating simulations. It lets you define your request pattern, and Hoverfly uses it to search its journal to find the matching requests. The second one is a `VerificationCriteria` which defines the verification criteria, such as the number of times a request was made. If the criteria are omitted, Hoverfly Java expects the request to have been made exactly once.

Here are some examples:

```
// Verify exactly one request
hoverfly.verify(
    service(matches("*.flight.*"))
        .get("/api/bookings")
        .anyQueryParams());

// Verify exactly two requests
hoverfly.verify(
    service("api.flight.com")
        .put("/api/bookings/1")
        .anyBody()
        .header("Authorization", "Bearer some-token"), times(2));
```

There are some useful `VerificationCriteria` static factory methods provided out-of-the-box. This will be familiar if you are a Mockito user.

```
times(1)
atLeastOnce(),
atLeast(2),
atMost(2),
never()
```

`VerificationCriteria` is a functional interface, meaning that you can provide your own criteria with a lambda expression. For example, you can create a more complex assertion on multiple request bodies, such as checking the transaction amount in a Charge object should keep increasing over time:

```
verify(service("api.payment.com").post("/v1/transactions").anyBody(),

    (request, data) -> {

        // Replace with your own criteria
        data.getJournal().getEntries().stream()
                .sorted(comparing(JournalEntry::getTimeStarted))
                .map(entry -> entry.getRequest().getBody())
                .map(body -> {
                    try {
                        return new ObjectMapper().readValue(body, Charge.class);
                    } catch (IOException e) {
                        throw new RuntimeException();
                    }
                })
                .reduce((c1, c2) -> {
                    if(c1.getTransaction() > c2.getTransaction()) {
                        throw new HoverflyVerificationError();
                    }
                    return c2;
                });
});
```

If you want to verify all the stubbed requests were made at least once, you can use `verifyAll`:

```
hoverfly.verifyAll();
```

You can also verify that an external service has never been called:

```
hoverfly.verifyZeroRequestTo(service(matches("api.flight.*")));
```

You can call `verify` as many times as you want, but requests are not verified in order by default. Support for verification in order will be added in a future release.

Resetting state

Verification is backed by a journal which logs all the requests made to Hoverfly. If multiple tests are sharing the same Hoverfly instance, for example when you are using `HoverflyRule` with `@ClassRule`, verification from one test might interfere with the requests triggered by another test.

In this case, you can reset the journal before each test to ensure a clean state for verifications:

```
@Before
public void setUp() throws Exception {

    hoverfly.resetJournal();

}
```

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:

```
localConfigs().proxyPort(8080)
```

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

```
localConfigs().destination("www.test.com") // only process requests to www.test.com
localConfigs().destination("api") // matches destination that contains api, eg. api.
↪test.com
localConfigs().destination(".+.test.com") // matches destination by regex, eg. dev.
↪test.com or stage.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.

```
localConfigs().proxyLocalHost()
```

You can configure Hoverfly to capture request headers which is turned off by default:

```
localConfigs().captureHeaders("Accept", "Authorization")
localConfigs().captureAllHeaders()
```

You can configure Hoverfly to run as a web server on default port 8500:

```
localConfigs().asWebServer()
```

You can configure Hoverfly to skip TLS verification. This option allows Hoverfly to perform "insecure" SSL connections to target server that uses invalid certificate (eg. self-signed certificate):

```
localConfigs().disableTlsVerification()
```

If you are developing behind a cooperate proxy, you can configure Hoverfly to use an upstream proxy:

```
localConfigs().upstreamProxy(new InetSocketAddress("127.0.0.1", 8900))
```

Logging

Hoverfly logs to SLF4J by default, meaning that you have control of Hoverfly logs using JAVA logging framework. Here is an example `logback.xml` that directs Hoverfly `WARN` logs to the console:

```
<?xml version="1.0" encoding="UTF-8"?>

<configuration scan="false" debug="false">

    <appender name="CONSOLE" class="ch.qos.logback.core.ConsoleAppender">
        <encoder>
            <charset>utf-8</charset>
            <Pattern>%date{ISO8601} [%-5level] %logger{10} %msg%n</Pattern>
        </encoder>
    </appender>

    <root level="INFO">
        <appender-ref ref="CONSOLE"/>
```

```
    </root>
    <logger name="hoverfly" level="WARN" additivity="false">
        <appender-ref ref="CONSOLE" />
    </logger>

</configuration>
```

You can override the default `hoverfly` logger name:

```
localConfigs().logger("io.test.hoverfly")
```

Or change the log output directly to stdout:

```
localConfigs().logToStdOut()
```

Hoverfly by default generates `INFO` logs regardless of the external SLF4J logger configs. To get debug logging, you need to set the log level explicitly:

```
localConfigs().logLevel(LogLevel.DEBUG)
```

Middleware

You can configure Hoverfly to use a local middleware (for more details, please check out Hoverfly Middleware):

```
localConfigs().localMiddleware("python", "middleware/modify_response.py")
```

You should provide the absolute or relative path of the binary, in this case, `python` for running the python middleware. The second input is the middleware script file in the classpath (eg. `test/resources` folder)

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:

```
localConfigs()
    .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.

Simulation Preprocessor

The `SimulationPreprocessor` interface lets you apply custom transformation to the `Simulation` object before importing to Hoverfly. This can be useful if you want to batch add/remove matchers, or update matcher types, like weakening matching criteria of captured data. Here is an example of adding a glob matcher for all the paths:

```
HoverflyConfig configBuilder = new LocalHoverflyConfig().simulationPreprocessor(s ->
            s.getHoverflyData().getPairs()
                    .forEach(
                            p -> p.getRequest().getPath()
                                    .add(new RequestFieldMatcher<>
↪(RequestFieldMatcher.MatcherType.GLOB, "/preprocessed/*"))
                    )
    );
```

See *Hoverfly Extension Configurations* if you are using JUnit5.

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 `remoteConfigs()` fluent builder. The default settings point to localhost on default admin port 8888 and proxy port 8500.

You can point it to other host and ports

```
remoteConfigs()
    .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.

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

You can configure Hoverfly to use an HTTPS admin endpoint.

```
remoteConfigs()
    .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.

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

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

Admin API client

Consuming Hoverfly Admin API is easy with *HoverflyClient*. It allows you to control an external Hoverfly instance, such as changing mode, setting simulation data, etc.

You can create a default client that points to localhost:8888

```
HoverflyClient.createDefault();
```

You can customize the hostname and port. If the external Hoverfly requires authentication, you can provide an auth token from environment variable.

```
HoverflyClient.custom()
            .host("remote.host")
            .port(12345)
            .withAuthToken()        // this will try to get the auth token from an␣
↪environment variable named 'HOVERFLY_AUTH_TOKEN'
            .build();
```

1.4.3 JUnit 4

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`.

Incremental Capture

In capture mode, `HoverflyRule` by default exports the simulation and overwrites any existing content of the supplied file. This is not very helpful if you add a new test and you don't want to re-capture everything. In this case, you can enable the incremental capture option. `HoverflyRule` will check and import if the supplied simulation file exists. Any new request / response pairs will be appended to the existing file during capturing.

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

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 start 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.

One caveat is that if you need to have a clean state for verification, you will need to manually reset the journal before each test:

```
@Before
public void setUp() throws Exception {

    hoverflyRule.resetJournal();

}
```

However this is not required if you are calling `hoverflyRule.simulate` in each test to load a new set of simulations, as the journal reset is triggered automatically in this case.

1.4.4 JUnit 5

Hoverfly JUnit 5 extension takes advantage of the new *Extension* API to give finer control of Hoverfly throughout JUnit 5 tests life cycles. This section will show you how to get started with this new feature, and how to configure hoverfly via annotations in the JUnit 5 tests.

Quick Start

If your project is already using JUnit 5, simply import the *hoverfly-java-junit5* library.

For Maven, add the following dependency to your pom:

```
<dependency>
    <groupId>io.specto</groupId>
    <artifactId>hoverfly-java-junit5</artifactId>
    <version>0.12.2</version>
    <scope>test</scope>
</dependency>
```

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

```
testCompile ``io.specto:hoverfly-java-junit5:0.12.2``
```

If you haven't yet use JUnit 5, here is an example of Gradle configuration to get you up and running with Hoverfly and JUnit 5 in your project.

```
buildscript {
    dependencies {
        classpath ``org.junit.platform:junit-platform-gradle-plugin:1.0.1''
    }

    repositories {
        mavenCentral()
    }
}

apply plugin: ``org.junit.platform.gradle.plugin''

dependencies {
    testCompile ``io.specto:hoverfly-java-junit5:0.12.2''

    testRuntime(``org.junit.platform:junit-platform-launcher:1.0.1'')
    testRuntime(``org.junit.jupiter:junit-jupiter-engine:5.0.1'')
    testRuntime(``org.junit.vintage:junit-vintage-engine:4.12.1'')
}
```

Hoverfly Extension

Enabling Hoverfly is as easy as registering the `HoverflyExtension` in your test class:

```
@ExtendWith(HoverflyExtension.class)
class MyTests {
    // ...
}
```

Declaring `HoverflyExtension` at the top of the class will start Hoverfly in simulation mode with default configurations, and stop it after all the tests are done.

You can add Hoverfly as an argument to constructor or methods which will be dynamically resolved by JUnit 5 runtime. It gives you the ability to configure hoverfly in a per-test basis, and use Hoverfly API to simulate using *DSL* or perform *Verification*.

```
@Test
void shouldDoSomethingWith(Hoverfly hoverfly) {
    // ...
}
```

As you can see, injecting Hoverfly into your test gives you a lot of flexibility, and `HoverflyExtension` takes care of resetting Hoverfly before each test to prevent interference.

Configurations

You can override Hoverfly configuration via the `@HoverflyCore` annotation, for example to change the ports of Hoverfly:

```
@HoverflyCore(mode = HoverflyMode.CAPTURE, config = @HoverflyConfig(adminPort = 9000,␣
↪proxyPort = 9001))
@ExtendWith(HoverflyExtension.class)
class CustomHoverflyTests {
    // ...
}
```

All the existing Hoverfly *Configuration* options are available via the `@HoverflyConfig` interface.

To set a *SimulationPreprocessor* for `@HoverflyConfig`, you can create a static nested class that implements the interface:

```
static class CustomSimulationPreprocessor implements SimulationPreprocessor {
    @Override
    public void accept(Simulation simulation) {
        // your transformation goes here
    }
}
```

Then pass your custom `SimulationPreprocessor` class to the `@HoverflyConfig` annotation like this:

```
@HoverflyCore(mode = HoverflyMode.SIMULATE, config = @HoverflyConfig(
        simulationPreprocessor = CustomSimulationPreprocessor.class
))
```

Simulate

With `@HoverflySimulate` annotation, you can declare a global simulation source that applies to all the tests in a test class. If one test loads a different simulation, `HoverflyExtension` is able to import the global source declared in `@HoverflySimulate` before the next test run.

```
package com.x.y.z;

@HoverflySimulate(source = @HoverflySimulate.Source(value = "test-service-https.json",
↪ type = HoverflySimulate.SourceType.CLASSPATH))
@ExtendWith(HoverflyExtension.class)
class SimulationTests {
    // ...
}
```

The current supported source type is `CLASSPATH`, `FILE`, and `DEFAULT_PATH``(which is ``src/test/resources/hoverfly`)

If no source is provided, it will try to locate a file called with fully qualified name of test class, replacing dots (.) and dollar signs ($) to underlines (_) in the Hoverfly default path. In this example, the file path that will be looked for is `src/test/resources/hoverfly/com_x_y_z_SimulationTests.json`

Note: To simulate using DSL, you can inject the Hoverfly object into your test method or constructor, and call the Hoverfly APIs directly.

As `HoverflyExtension` implements JUnit 5 `ParameterResolver`, you can serve up the configured Hoverfly object in your tests for further customization. You can refer to JUnit 5 User Guide here

```
@ExtendWith(HoverflyExtension.class)
class SimulationTests {

    @Test
    void shouldDoSomethingWith(Hoverfly hoverfly) {
        hoverfly.simulate(dsl(
            service("www.my-test.com")
                .post("/api/bookings").body("{\"flightId\": \"1\"}")
                .willReturn(created("http://localhost/api/bookings/1"))));

        // ...
    }
}
```

Capture

You can declare `@HoverflyCapture` to run Hoverfly in capture mode (see *Capturing*). You can customize the path and the filename for exporting the simulations.

```
@HoverflyCapture(path = "build/resources/test/hoverfly",
            filename = "captured-simulation.json",
            config = @HoverflyConfig(captureAllHeaders = true, proxyLocalHost = true))
@ExtendWith(HoverflyExtension.class)
class CaptureTests {
    // ...
}
```

If `path` and `filename` are not provided, the simulation will be exported to a file with fully-qualified name of the test class in the default Hoverfly path.

Capture or simulate

You can set `HoverflyExtension` to switch between simulate and capture mode automatically. If a source is not found, it will capture, otherwise, simulate. This is previously known as `inCaptureOrSimulateMode` in JUnit 4 `HoverflyRule` (see *Capture or Simulate*).

This feature can be enabled easily by setting `enableAutoCapture` to `true` in `@HoverflySimulate`:

```
@HoverflySimulate(source = @Source(value = "build/resources/test/hoverfly/missing-
↪simulation.json", type = SourceType.FILE),
    enableAutoCapture = true)
@ExtendWith(HoverflyExtension.class)
class CaptureIfFileNotPresent {
    // ...
}
```

Diff

You can declare `@HoverflyDiff` to run Hoverfly in diff mode (see *Diffing*). You can customize the location of the simulation data as well as the Hoverfly configuration parameters.

```
@HoverflyDiff(
    source = @HoverflySimulate.Source(value = "hoverfly/diff/captured-wrong-
↪simulation-for-diff.json",
        type = HoverflySimulate.SourceType.CLASSPATH))
)
```

Also you can use `@HoverflyValidate` at class or method level, to assert automatically that there is no difference between simulated and captured traffic.

Nested tests

If your test class contains several groups of tests that require different Hoverfly configurations, you can do so by registering `HoverflyExtension` with nested tests:

```
@Nested
@HoverflySimulate
@ExtendWith(HoverflyExtension.class)
class MyNestedTestsOne {
    // ...
}

@Nested
@HoverflyCapture
@ExtendWith(HoverflyExtension.class)
class MyNestedTestsTwo {
    // ...
}
```

1.4.5 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

OkHttpClient

If you are using OkHttpClient to make HTTPS requests, you will need to configure it to use the custom `SSLContext` and `TrustManager` that supports Hoverfly CA cert:

```
SslConfigurer sslConfigurer = hoverflyRule.getSslConfigurer();
OkHttpClient okHttpClient = new OkHttpClient.Builder()
        .sslSocketFactory(sslConfigurer.getSslContext().getSocketFactory(),␣
↪sslConfigurer.getTrustManager())
        .build();
```

Spock Framework

If you are testing with BDD and Spock Framework, you could also use Hoverfly-Java JUnit Rule. Just initialize a `HoverflyRule` in the Specification, and annotate it with `@ClassRule` and `@Shared` which indicates the `HoverflyRule` is shared among all the feature methods:

```
class MySpec extends Specification {

    @Shared
    @ClassRule
    HoverflyRule hoverflyRule = HoverflyRule.inSimulationMode()

    // Feature methods

    def setup() {
        // Reset the journal before each feature if you need to do a verification
        hoverflyRule.resetJournal()
    }
}
```

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

Migration to the latest (V5) schema

Starting from Hoverfly-java v0.11.0, the simulation schema is upgraded to v5 which is a big leap in terms of the maturity of header and query matchers, and the possibility to introduce more request matchers without any breaking changes in the future. Although Hoverfly is designed to be backward compatible with all the previous schemas, upgrading to v5 is highly recommended:

```
$ hoverctl start
$ 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'
    }
}
```

Trusting Hoverfly certificate

Your HTTP client need to trust Hoverfly's self-signed certificate in order for Hoverfly to intercept and decrypt HTTPS traffic.

You will get an error like this if the certificate is not trusted.

```
javax.net.ssl.SSLHandshakeException: PKIX path building failed: sun.
security.provider.certpath.SunCertPathBuilderException: unable to find valid
certification path to requested target
```

Hoverfly sets its certificate to be trusted in the the default `SSLContext`. If your HTTP client uses the default `SSLContext`, then you don't need to do anything.

Otherwise, you should refer to your HTTP client documentation to find out how to customize the trusted certificates.

Hoverfly provides the following methods to return the `SSLContext` and `TrustManager` if you ever need to configure your HTTP client:

```
hoverflyRule.getSslConfigurer().getSslContext();
hoverflyRule.getSslConfigurer().getTrustManager();
```

As a last resort, you can still trust Hoverfly certificate by adding it to the global Java keystore:

```
$ wget https://raw.githubusercontent.com/SpectoLabs/hoverfly/master/core/cert.pem
$ sudo $JAVA_HOME/bin/keytool -import -alias hoverfly -keystore $JAVA_HOME/jre/lib/
↪security/cacerts -file cert.pem
```