Hoverfly Documentation

Release v0.10.2

SpectoLabs

Mar 27, 2017

Contents

Hoverfly is a lightweight, open source API simulation tool. Using Hoverfly, you can create realistic simulations of the APIs your application depends on.

- Replace slow, flaky API dependencies with realistic, re-usable simulations
- Simulate network latency, random failures or rate limits to test edge-cases
- Extend and customize with any programming language
- Export, share, edit and import API simulations
- CLI and native language bindings for Java and Python
- REST API
- Lightweight, high-performance, run anywhere
- Apache 2 license

CHAPTER 1

Source

The Hoverfly source code is available on GitHub.

Hoverfly is developed and maintained by SpectoLabs.

CHAPTER 2

Contents

Introduction

Motivation

Developing and testing interdependent applications is difficult. Maybe you're working on a mobile application that needs to talk to a legacy API. Or a microservice that relies on two other services that are still in development.

The problem is the same: how do you develop and test against external dependencies which you cannot control?

You could use mocking libraries as substitutes for external dependencies. But mocks are intrusive, and do not allow you to test all the way to the architectural boundary of your application.

Stubbed services are better, but they often involve too much configuration or may not be transparent to your application.

Then there is the problem of managing test data. Often, to write proper tests, you need fine-grained control over the data in your mocks or stubs. Managing test data across large projects with multiple teams introduces bottlenecks that impact delivery times.

Integration testing "over the wire" is problematic too. When stubs or mocks are swapped out for real services (in a continuous integration environment for example) new variables are introduced. Network latency and random outages can cause integration tests to fail unexpectedly.

Hoverfly was designed to provide you with the means to create your own "dependency sandbox": a simulated development and test environment that you control.

Hoverfly grew out of an effort to build "the smallest service virtualization tool possible".

Download and installation

Hoverfly comes with a command line interface called **hoverctl**. Archives containing the Hoverfly and hoverctl binaries are available for the major operating systems and architectures.

- MacOS 64bit
- Linux 32bit

- Linux 64bit
- Windows 32bit
- Windows 64bit

Download the correct archive, extract the binaries and place them in a directory on your PATH.

Homebrew (MacOS)

If you have homebrew, you can install Hoverfly using the `brew` command.

```
brew install SpectoLabs/tap/hoverfly
```

To upgrade your existing hoverfly to the latest release:

```
brew upgrade hoverfly
```

To show which versions are installed in your machine:

```
brew list --version hoverfly
```

You can switch to a previously installed version as well:

```
brew switch hoverfly <version>
```

To remove old versions :

```
brew cleanup hoverfly
```

Getting Started

Note

It is recommended that you keep Hoverfly and hoverctl in the same directory. However if they are not in the same directory, hoverctl will look in the current directory for Hoverfly, then in other directories on the PATH.

Hoverfly is composed of two binaries: **Hoverfly** and **hoverctl**.

hoverctl is a command line tool that can be used to configure and control Hoverfly. It allows you to run Hoverfly as a daemon.

Hoverfly is the application that does the bulk of the work. It provides the proxy server or webserver, and the API endpoints.

Once you have extracted both Hoverfly and hoverctl into a directory on your PATH, you can run hoverctl and Hoverfly.

```
hoverctl version
hoverfly -version
```

Both of these commands should return a version number. Now you can run an instance of Hoverfly:

```
hoverctl start
```

Check whether Hoverfly is running with the following command:

```
hoverctl logs
```

The logs should contain the string `serving proxy`. This indicates that Hoverfly is running.

Finally, stop Hoverfly with:

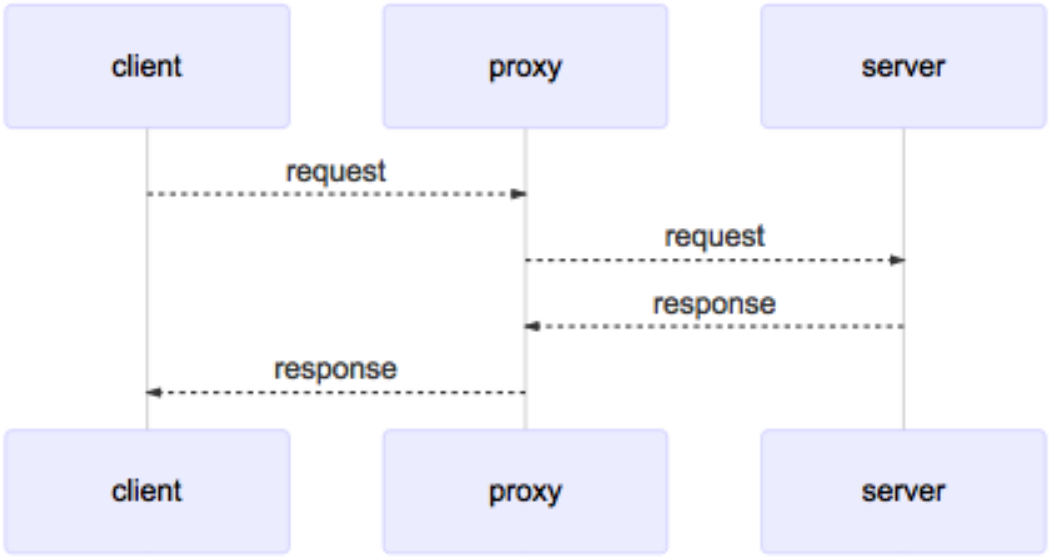
```
hoverctl stop
```

Key Concepts

Hoverfly's functionality is quite broad. You are encouraged to take the time to understand these key concepts before jumping into the *Tutorials*.

Hoverfly as a proxy server

A proxy server passes requests between a client and server.



It is sometimes necessary to use a proxy server to reach a network (as a security measure, for example). Because of this, all network-enabled software can be configured to use a proxy server.

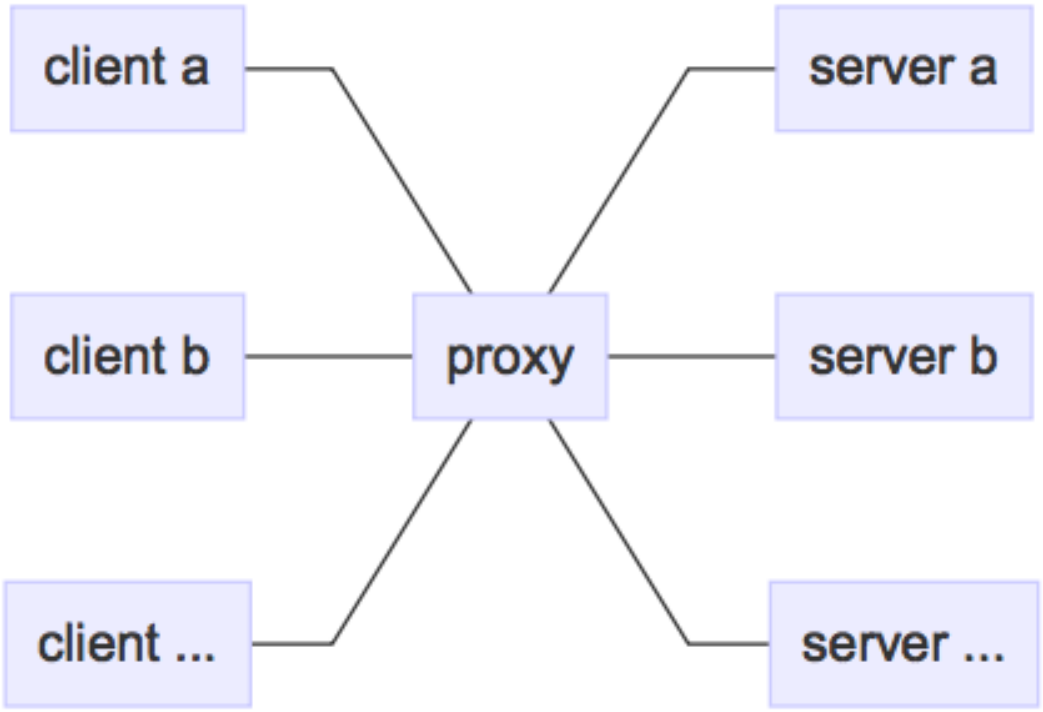
The relationship between clients and servers via a proxy server can be one-to-one, one-to-many, many-to-one, or many-to-many.

By default Hoverfly starts as a proxy server.

Using a proxy server

Applications can usually be configured to use a proxy server by setting environment variables:

```
export HTTP_PROXY="http://proxy-address:port"
export HTTPS_PROXY="https://proxy-address:port"
```


client a
client b
client ...
proxy
server a
server b
server ...

Launching network-enabled software within an environment containing these variables *should* make the application use the specified proxy server. The term *should* is used as not all software respects these environment variables for security reasons.

Alternatively, applications themselves can usually be configured to use a proxy. Curl can be configured to use a proxy via flags.

```
curl http://hoverfly.io --proxy http://proxy-ip:port
```

Note: The proxy configuration methods described here are intended to help you use the code examples in this documentation. The method of configuring an application or operating system to use a proxy varies depending on the environment.

- Windows Proxy Settings Explained
- Firefox Proxy Settings

The difference between a proxy server and a webserver

A proxy server is a type of webserver. The main difference is that when a webserver recieves a request from a client, it is expected to respond with whatever the intended response is (an HTML page, for example). The data it responds with is generally expected to reside on that server, or within the same network.

A proxy server is expected to pass the incoming request on to another server (the "destination"). It is also expected to set some appropriate headers along the way, such as X-Forwarded-For, X-Real-IP, X-Forwarded-Proto etc. Once the proxy server receives a response from the destination, it is expected to pass it back to the client.

Hoverfly as a webserver

Sometimes you may not be able to configure your client to use a proxy, or you may want to explicitly point your application at Hoverfly. For this reason, Hoverfly can run as a webserver.

Note: When running as a webserver, Hoverfly cannot capture traffic (see *Capture mode*) - it can only be used to simulate and synthesize APIs (see *Simulate mode*, *Modify mode* and *Synthesize mode*). For this reason, when you use Hoverfly as a webserver, you should have Hoverfly simulations ready to be loaded.

When running as a webserver, Hoverfly strips the domain from the endpoint URL. For example, if you made requests to the following URL while capturing traffic with Hoverfly running as a proxy:

```
http://echo.jsontest.com/key/value
```

And Hoverfly is running in simulate mode as a webserver on:

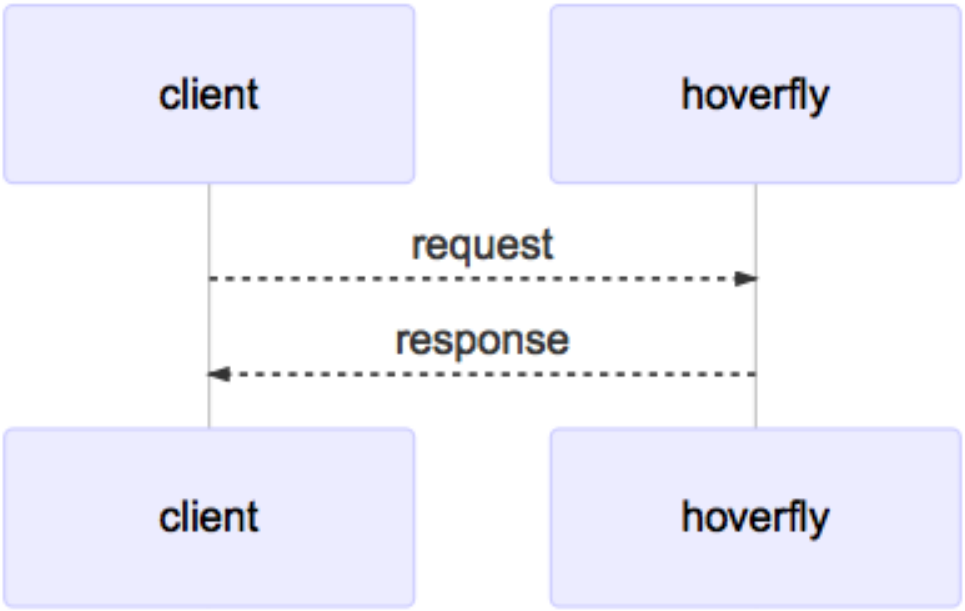
```
http://localhost:8888
```

Then the URL you would use to retrieve the data from Hoverfly would be:

```
http://localhost:8500/key/value
```

See also:

Please refer to the *Running Hoverfly as a webserver* tutorial for a step-by-step example.

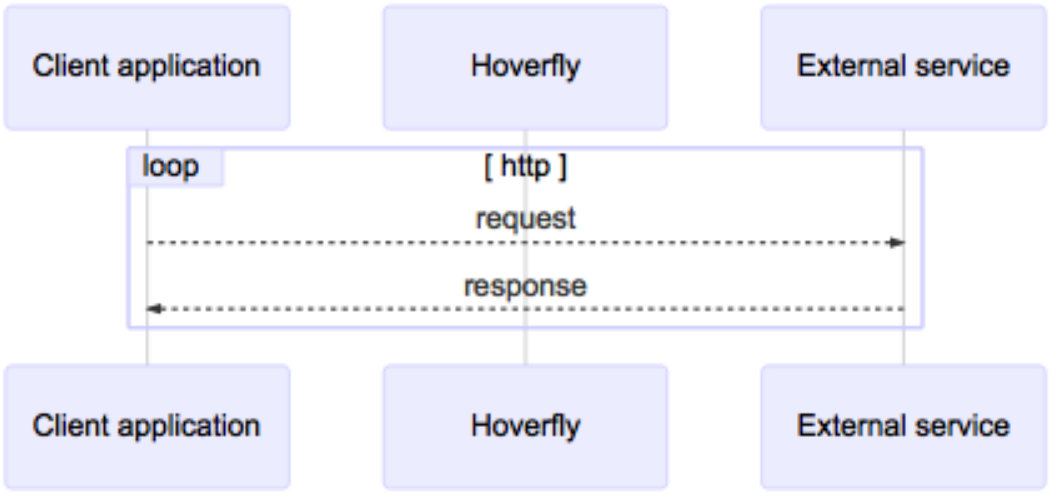


Hoverfly modes

Hoverfly has four different modes. It can only run in one mode at any one time.

Capture mode

Capture mode is used for creating API simulations.



In Capture mode, Hoverfly (running as a proxy server - see *Hoverfly as a proxy server*) intercepts communication between the client application and the external service. It transparently records outgoing requests from the client and the incoming responses from the service API.

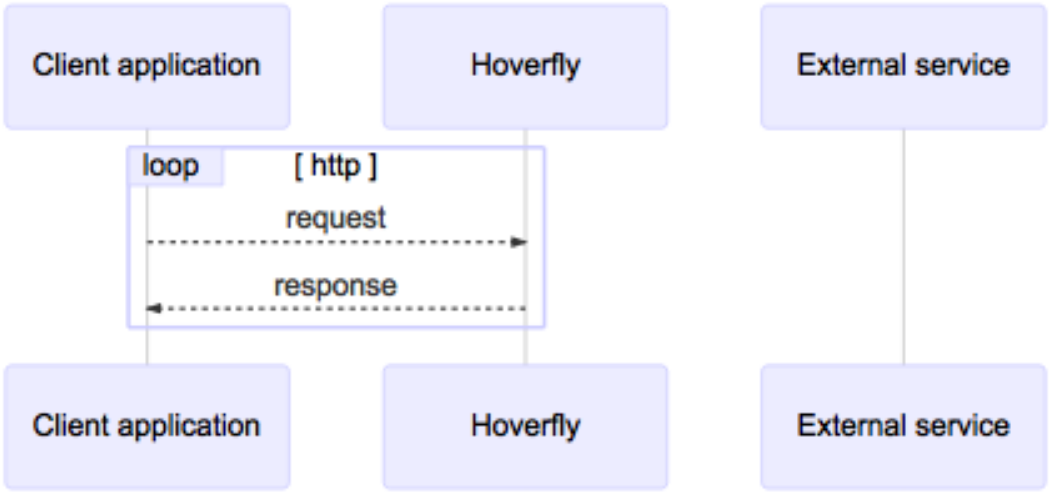
Most commonly, requests to the external service API are triggered by running automated tests against the application that consumes the API. During subsequent test runs, Hoverfly can be set to run in *Simulate mode*, removing the dependency on the real external service API. Alternatively, requests can be generated using a manual process.

Usually, Capture mode is used as the starting point in the process of creating an API simulation. Captured data is then exported and modified before being re-imported into Hoverfly for use as a simulation.

Note: Hoverfly cannot be set to Capture mode when running as a webserver (see *Hoverfly as a webserver*).

Simulate mode

In this mode, Hoverfly uses traffic captured using *Capture mode* (which may also have been manually edited) to mimic external APIs.



Each time Hoverfly receives a request from the application, instead of forwarding it to the intended destination, it looks in the simulation data for a matching response. If it finds a match, it returns the response to the application.

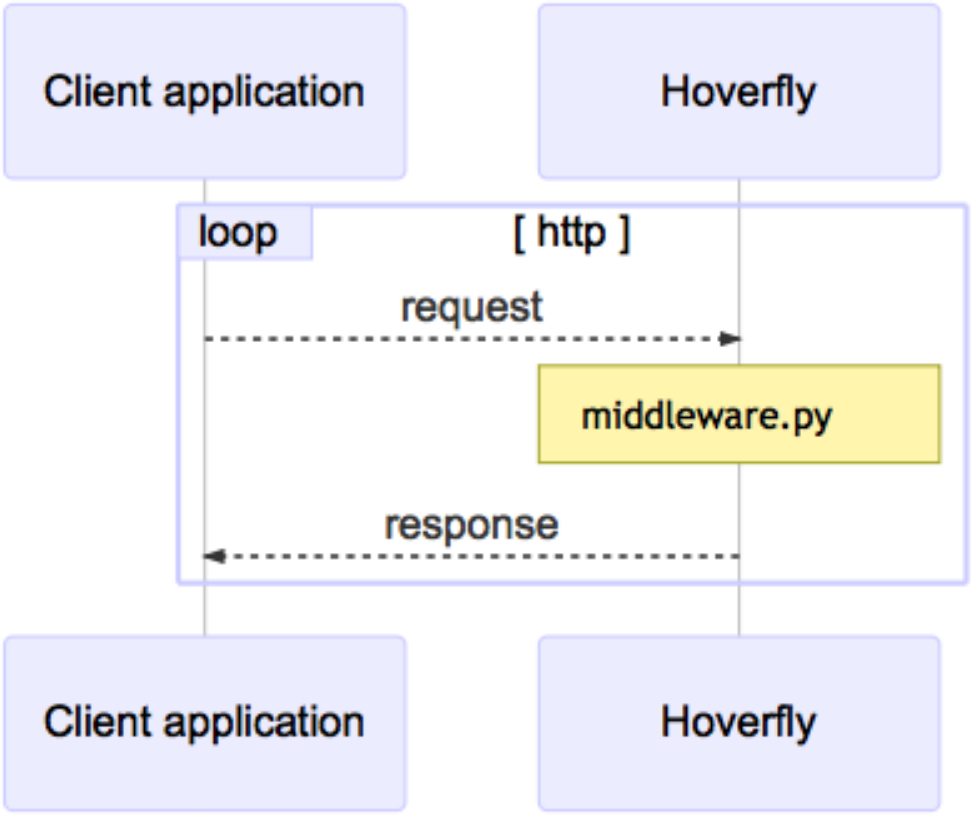
This simple matching strategy won't always be appropriate however. In some cases you may want Hoverfly to return a single response for a number of different possible requests. This can be done by editing the simulation data (see *Templates*).

Synthesize mode

This mode is similar to *Simulate mode*, but instead of looking for a response in stored simulation data, the request is passed directly to a user-supplied executable file. These files are known as *Middleware*.

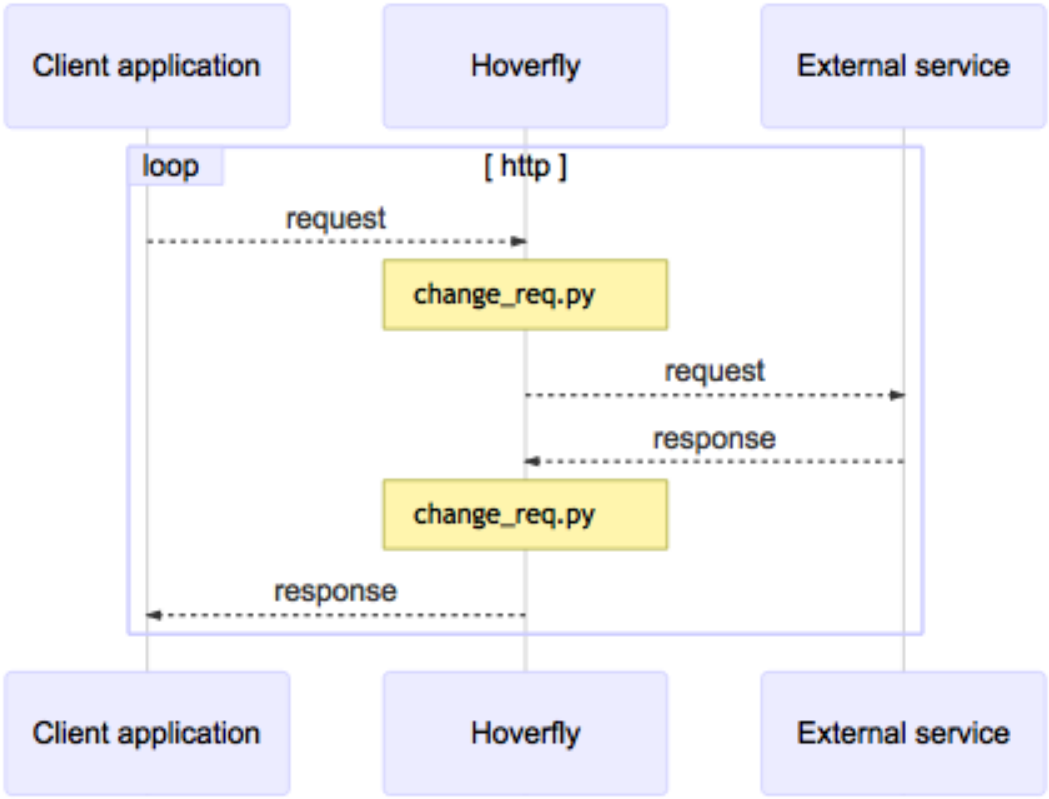
In Synthesize mode, the middleware executable is expected to generate a response to the incoming request "on the fly". Hoverfly will then return the generated response to the application. For Hoverfly to operate in Synthesize mode, a middleware executable must be specified.

Note: You might use this mode to simulate an API that may be too difficult to record correctly via *Capture mode*. An example would be an API that uses state to change the responses. You could create middleware that manages this state and produces the desired response based on the data in the request.


Client application
Hoverfly
loop
[http]
request
middleware.py
response
Client application
Hoverfly

Modify mode

Modify mode is similar to *Capture mode*, except it **does not save the requests and responses**. In Modify mode, Hoverfly will pass each request to a *Middleware* executable before forwarding it to the destination. Responses will also be passed to middleware before being returned to the client.



You could use this mode to "man in the middle" your own requests and responses. For example, you could change the API key you are using to authenticate against a third-party API.

Simulations

The core functionality of Hoverfly is to capture HTTP(S) traffic to create API simulations which can be used in testing. Hoverfly stores captured traffic as *simulations*.

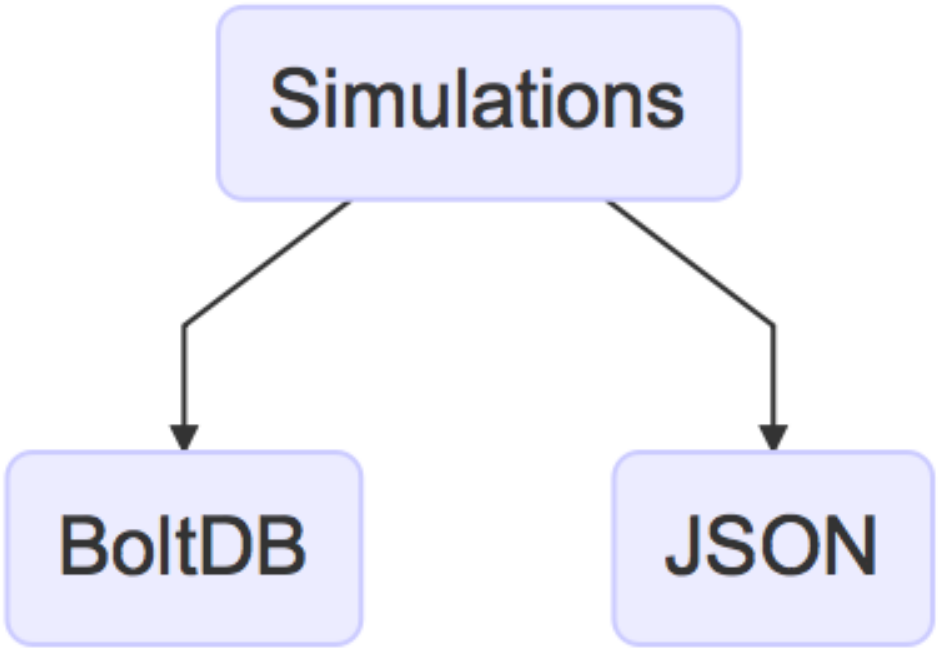
Simulations can be written to disk in two different formats: in a JSON, or a BoltDB database format.

<simulation>.json

Simulation JSON can be exported, edited and imported in and out of Hoverfly, and can be shared among Hoverfly users or instances. Simulation JSON files must adhere to the Hoverfly *Simulation schema*.

See also:

For a hands-on tutorial of creating and editing simulations, see *Creating and exporting a simulation*.


Simulations
BoltDB
JSON

requests.db

Hoverfly can also store simulation data on disk in a file called *requests.db*. This file is written to the current working directory. This means simulations do not need to be manually exported and re-imported between Hoverfly invocations.

> **Warning:** Please note although you can persist the Hoverfly data store to disk, Hoverfly does not store all of its internal state in this database currently. The only way to access, modify and share the full state of a running instance of Hoverfly is through the simulation JSON.

This mechanism uses a very high performance Golang database system: BoltDB.

Captured traffic

Hoverfly's core functionality is to capture requests and responses ("traffic") to create API simulations.

Request response pairs

When you capture traffic using Hoverfly's *Capture mode* and export resulting the simulation to JSON, you will see *request response pairs*:

```
{
  "response": {
    "status": 200,
    "body": "body here",
    "encodedBody": false,
    "headers": {
      "Content-Type": ["text/html; charset=utf-8"]
    }
  },
  "request": {
    "requestType": "recording",
    "path": "/",
    "method": "GET",
    "destination": "myhost.io",
    "scheme": "https",
    "query": "",
    "body": "",
    "headers": {
      "Accept": ["text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/
↪*;q=0.8"],
      "User-Agent": ["Mozilla/5.0 (Macintosh; Intel Mac OS X 10_11_6) AppleWebKit/537.
↪36 (KHTML, like Gecko) Chrome/52.0.2743.116 Safari/537.36"]
    }
  }
}
```

Notice the `"response"` and `"request"` key values in the document above. They both contain all of the fields from the original request and response. The headers are also stored.

Please also notice the `"requestType"` key, with its corresponding `"recording"` value: this denotes that the request response pair was created while using Hoverfly in *Capture mode*, and that in *Simulate mode*, Hoverfly will return this exact response when it receives this request.

Note: Since JSON does not support binary data, binary responses are base64 encoded. This is denoted by the encodedBody field. Hoverfly automatically encodes and decodes the data during the export and import phases.

Matching

Hoverfly simulates APIs by *matching* responses to incoming requests.

Imagine scanning through a dictionary for a word, and then looking up its definition. Hoverfly does exactly that, but the "word" is the URL that was "recorded" in *Capture mode*, plus all the fields in the `"request"` part of the document above, with the exception of the headers.

Note: The reason headers are not included in the match is because they can vary depending on the client.

This one-to-one matching strategy is extremely fast, but in some cases you may want Hoverfly to return a single response for more than one request. This is possible using *Templates*.

Templates

Sometimes simple one-to-one matching of responses to requests is not enough.

Request templates are defined in the *Simulation schema* by setting the `"requestType"` property for a request to `"template"` and including only the information in the request that you want Hoverfly to use in the match.

In the example below, Hoverfly will return the same response for any request with the path `/template`:

```
{
    "data": {
        "pairs": [{
            "response": {
                "status": 200,
                "body": "<h1>Matched on template</h1>",
                "encodedBody": false,
                "headers": {
                    "Content-Type": ["text/html; charset=utf-8"]
                }
            },
            "request": {
                "requestType": "template",
                "path": "/template"
            }
        }]
    }
}
```

For looser matching on URL paths, it is possible to use a wildcard to substitute characters. This is achieved by using the `*` symbol. This will match any number of characters, and is case sensitive.

In the next example, Hoverfly will return the same response for requests with the path `/api/v1/template` or `/api/v2/template`.

```
{
    "data": {
        "pairs": [{
```

```
            "response": {
                "status": 200,
                "body": "<h1>Matched on template</h1>",
                "encodedBody": false,
                "headers": {
                    "Content-Type": ["text/html; charset=utf-8"]
                }
            },
            "request": {
                "requestType": "template",
                "path": "/api/*/template"
            }
        }]
    }
}
```

The *Simulation schema* can contain both request recordings and request templates:

```
{
    "data": {
        "pairs": [{
            "response": {
                "status": 200,
                "body": "<h1>Matched on recording</h1>",
                "encodedBody": false,
                "headers": {
                    "Content-Type": [
                        "text/html; charset=utf-8"
                    ]
                }
            },
            "request": {
                "requestType": "recording",
                "path": "/",
                "method": "GET",
                "destination": "myhost.io",
                "scheme": "https",
                "query": "",
                "body": "",
                "headers": {
                    "Accept": [
                        "text/html,application/xhtml+xml,application/xml;q=0.9,image/
↪webp,*/*;q=0.8"
                    ],
                    "Content-Type": [
                        "text/plain; charset=utf-8"
                    ],
                    "User-Agent": [
                        "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_11_6) AppleWebKit/
↪537.36 (KHTML, like Gecko) Chrome/52.0.2743.116 Safari/537.36"
                    ]
                }
            }
        }, {
            "response": {
                "status": 200,
                "body": "<h1>Matched on template</h1>",
                "encodedBody": false,
```

```
                "headers": {
                    "Content-Type": [
                        "text/html; charset=utf-8"
                    ]
                }
            },
            "request": {
                "requestType": "template",
                "path": "/template",
                "method": null,
                "destination": null,
                "scheme": null,
                "query": null,
                "body": null,
                "headers": null
            }
        }],
        "globalActions": {
            "delays": []
        }
    }
}
```

A standard workflow might be:

1. Capture some traffic
2. Export it to JSON
3. Edit the JSON to set certain requests to templates, removing the properties for these requests that should be excluded from the match or substituting characters in the URL path for wildcards
4. Re-import the JSON to Hoverfly

Note: If the `"requestType"` property is not defined or not recognized, Hoverfly will treat a request as a `"recording"`.

See also:

Templating is best understood with a practical example, so please refer to *Adding templates to a simulation* to get hands on experience with templating.

Destination filtering

By default, Hoverfly will process every request it receives. However, you may wish to control which URLs Hoverfly processes.

This is done by *filtering* the *destination* URLs using either a string or a regular expression. The *destination* string or regular expression will be compared against the host and the path of a URL.

For example, specifying `hoverfly.io` as the destination value will tell Hoverfly to process only URLs on the `hoverfly.io` host.

```
hoverctl destination "hoverfly.io"
```

Specifying `api` as the *destination* value during *Capture mode* will tell Hoverfly to capture only URLs that contain the string `api`. This would include both `api.hoverfly.io/endpoint` and `hoverfly.io/api/endpoint`.

See also:

This functionality is best understood via a practical example: see *Capturing or simulating specific URLs* in the *Tutorials* section.

Note: The destination setting applies to all Hoverfly modes. If a destination value is set while Hoverfly is running in *Simulate mode*, requests that are excluded by the destination setting will be passed through to the real URLs. This makes it possible to return both real and simulated responses.

Meta

The last part of the simulation schema is the meta object. Its purpose is to store metadata that is relevant to your simulation. This includes the simulation schema version, the version of Hoverfly used to export the simulation and the date and time at which the simulation was exported.

```
"meta": {
  "schemaVersion": "v1",
  "hoverflyVersion": "v0.9.0",
  "timeExported": "2016-11-11T11:53:52Z"
}
```

Delays

Once you have created a simulated service by capturing traffic between your application and an external service, you may wish to make the simulation more "realistic" by applying latency to the responses returned by Hoverfly.

Hoverfly can be configured to apply delays to responses based on URL pattern matching or HTTP method. This is done using a regular expression to match against the URL, a delay value in milliseconds, and an optional HTTP method value.

See also:

This functionality is best understood via a practical example: see *Adding delays to a simulation* in the *Tutorials* section.

You can also apply delays to simulations using *Middleware* (see the *Using middleware to simulate network latency* tutorial). Using middleware to apply delays sacrifices performance for flexibility.

Middleware

Middleware intercepts traffic between the client and the API (whether real or simulated), and allowing you to manipulate it.

You can use middleware to manipulate data in simulated responses, or to inject unpredictable performance characteristics into your simulation.

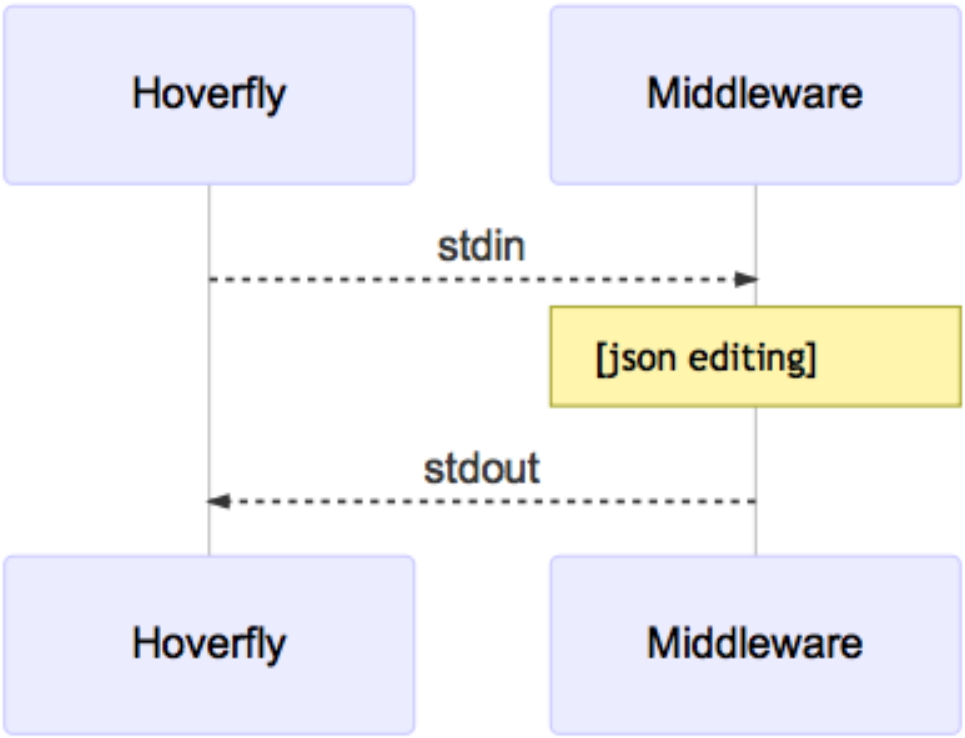
Middleware works differently depending on the Hoverfly mode.

- Capture mode: middleware affects only **outgoing requests**
- Simulate mode: middleware affects only **incoming responses** (cache contents remain untouched)
- Synthesize mode: middleware **creates responses**
- Modify mode: middleware affects **requests and responses**

You can write middleware in any language as long as it can be executed by the shell on the Hoverfly host. Middleware could be a binary file, or a script.

Middleware Interface

When middleware is called by Hoverfly, it expects to receive and return JSON (see *Simulation schema*). Middleware can be used to modify the values in the JSON but **must not** modify the schema itself.



Hoverfly will send the JSON object to middleware via the standard input stream. Hoverfly will then listen to the standard output stream and wait for the JSON object to be returned.

See also:

Middleware examples are covered in the tutorials section. See *Using middleware to simulate network latency* and *Using middleware to modify response payload and status code*.

Native language bindings

Native language bindings are available for Hoverfly to make it easy to integrate into different environments.

HoverPy

To get started:

```
sudo pip install hoverpy
python
```

And in Python you can simply get started with:

```
import hoverpy
import requests

# capture mode
with hoverpy.HoverPy(capture=True) as hp:
    data = requests.get("http://time.jsontest.com/").json()

# simulation mode
with hoverpy.HoverPy() as hp:
    simData = requests.get("http://time.jsontest.com/").json()
    print(simData)

assert(data["milliseconds_since_epoch"] == simData["milliseconds_since_epoch"])
```

For more information, read the HoverPy documentation.

Hoverfly Java

- Strict or loose HTTP request matching based on URL, method, body and header combinations
- Fluent and expressive DSL for easy generation of simulated APIs
- Automatic marshalling of objects into JSON during request/response body generation
- HTTPS automatically supported, no extra configuration required
- Download via Maven or Gradle

To get started, read the Hoverfly Java documentation.

Tutorials

In these examples, we will use the hoverctl (CLI tool for Hoverfly) to interact with Hoverfly.

Hoverfly can also be controlled via its *REST API*, or via *Native language bindings*.

Basic tutorials

Creating and exporting a simulation

Note: If you are running Hoverfly on a machine that accesses the internet via a proxy (for example if you are on a corporate network), please follow the *Using Hoverfly behind a proxy* tutorial before proceeding.

Start Hoverfly and set it to Capture mode

```
hoverctl start
hoverctl mode capture
```

Make a request with cURL, using Hoverfly as a proxy server:

```
curl --proxy http://localhost:8500 http://time.jsontest.com
```

View the Hoverfly logs

```
hoverctl logs
```

Export the simulation to a JSON file

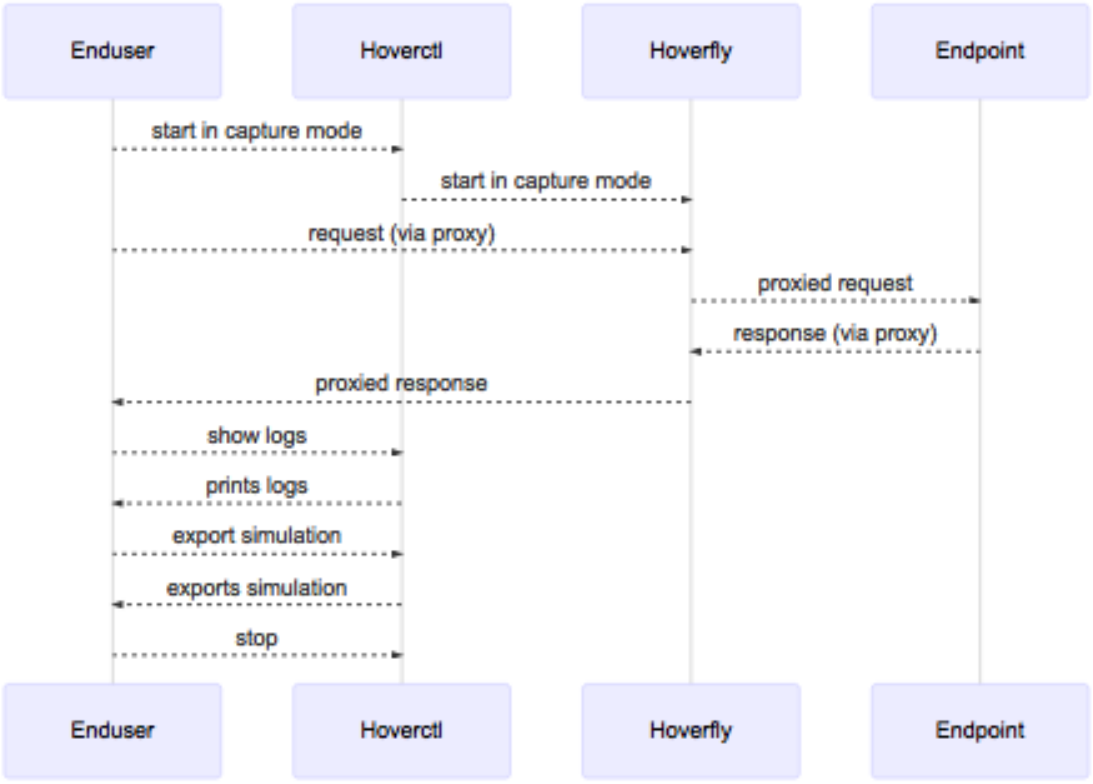
```
hoverctl export simulation.json
```

Stop hoverfly

```
hoverctl stop
```

You'll now see a `simulation.json` file in your current working directory, which contains all your simulation data.

In case you are curious, the sequence diagram for this process looks like this:



Importing and using a simulation

In this tutorial we are going to import the simulation we created in the previous tutorial.

```
hoverctl start
hoverctl import simulation.json
```

Hoverfly can also import simulation data that is stored on a remote host via HTTP:

```
hoverctl import https://example.com/example.json
```

Make a request with cURL, using Hoverfly as a proxy.

```
curl --proxy localhost:8500 http://time.jsontest.com
```

This outputs the time at the time the request was captured.

```
{
   "time": "02:07:28 PM",
   "milliseconds_since_epoch": 1482242848562,
   "date": "12-20-2016"
}
```

Stop Hoverfly:

```
hoverctl stop
```

Adding delays to a simulation

Simulating API latency during development allows you to write code that will deal with it gracefully.

In Hoverfly, this is done by applying "delays" to responses in a simulation.

Delays are applied by editing the Hoverfly simulation JSON file. Delays can be applied selectively according to request URL pattern and/or HTTP method.

Applying a delay to all responses

Let's apply a 2 second delay to all responses. First, we need to create and export a simulation.

```
hoverctl start
hoverctl mode capture
curl --proxy localhost:8500 http://time.jsontest.com
curl --proxy localhost:8500 http://date.jsontest.com
hoverctl export simulation.json
hoverctl stop
```

Take a look at the `"globalActions"` property within the `simulation.json` file you exported. It should look like this:

```
                "globalActions": {
                        "delays": []
                }
```

Edit the file so the `"globalActions"` property looks like this:

```
                "globalActions": {
                        "delays": [
                                {
                                        "urlPattern": ".",
                                        "delay": 2000
                                }
```

```
                ]
        }
```

Hoverfly will apply a delay of 2000ms to all URLs that match the `"urlPattern"` value. We want the delay to be applied to **all URLs**, so we set the `"urlPattern"` value to the regular expression `".*"`.

Now import the edited `simulation.json` file, switch Hoverfly to Simulate mode and make the requests again.

```
hoverctl start
hoverctl import simulation.json
curl --proxy localhost:8500 http://time.jsontest.com
curl --proxy localhost:8500 http://date.jsontest.com
hoverctl stop
```

The responses to both requests are delayed by 2 seconds.

Applying different delays based on host

Now let's apply a delay of 1 second on responses from `time.jsontest.com` and a delay of 2 seconds on responses from `date.jsontest.com`.

Run the following to create and export a simulation.

```
hoverctl start
hoverctl mode capture
curl --proxy localhost:8500 http://time.jsontest.com
curl --proxy localhost:8500 http://date.jsontest.com
hoverctl export simulation.json
hoverctl stop
```

Edit the `simulation.json` file so that the `"globalActions"` property looks like this:

```
        "globalActions": {
                "delays": [
                        {
                        "urlPattern": "time\\.jsontest\\.com",
                        "delay": 1000
                        },
                        {
                        "urlPattern": "date\\.jsontest\\.com",
                        "delay": 2000
                        }
                ]
        }
```

Now run the following to import the edited `simulation.json` file and run the simulation:

```
hoverctl start
hoverctl import simulation.json
curl --proxy localhost:8500 http://time.jsontest.com
curl --proxy localhost:8500 http://date.jsontest.com
hoverctl stop
```

You should notice a 1 second delay on responses from `time.jsontest.com`, and a 2 second delay on responses from `date.jsontest.com`.

Note: You can easily get into a situation where your request URL has multiple matches. In this case, the first successful match wins.

Applying different delays based on URI

Now let's apply different delays based on location. Run the following to create and export a simulation.

```
hoverctl start
hoverctl mode capture
curl --proxy localhost:8500 http://echo.jsontest.com/a/b
curl --proxy localhost:8500 http://echo.jsontest.com/b/c
curl --proxy localhost:8500 http://echo.jsontest.com/c/d
hoverctl export simulation.json
hoverctl stop
```

Edit the `simulation.json` file so that the `"globalActions"` property looks like this:

```
                "globalActions": {
                        "delays": [
                                {
                                "urlPattern": "echo\\.jsontest\\.com\\/a\\/b",
                                "delay": 2000
                                },
                                {
                                "urlPattern": "echo\\.jsontest\\.com\\/b\\/c",
                                "delay": 2000
                                },
                                {
                                "urlPattern": "echo\\.jsontest\\.com\\/c\\/d",
                                "delay": 3000
                                }
                        ]
                }
```

Now run the following to import the edited `simulation.json` file and run the simulation:

```
hoverctl start
hoverctl import simulation.json
hoverctl mode simulate
curl --proxy localhost:8500 http://echo.jsontest.com/a/b
curl --proxy localhost:8500 http://echo.jsontest.com/b/c
curl --proxy localhost:8500 http://echo.jsontest.com/c/d
hoverctl stop
```

You should notice a 2 second delay on responses from `echo.jsontest.com/a/b` and `echo.jsontest.com/b/c`, and a 3 second delay on the response from `echo.jsontest.com/c/d`.

Applying different delays based on HTTP method

Let's apply a delay of 2 seconds on responses to **GET requests only** made to `echo.jsontest.com/b/c`.

Run the following to create and export a simulation.

```
hoverctl start
hoverctl mode capture
curl --proxy localhost:8500 http://echo.jsontest.com/b/c
curl --proxy localhost:8500 -X POST http://echo.jsontest.com/b/c
hoverctl export simulation.json
hoverctl stop
```

Edit the `simulation.json` file so that the `"globalActions"` property looks like this:

```
                "globalActions": {
                        "delays": [
                                {
                                "urlPattern": "echo\\.jsontest\\.com\\/b\\/c",
                                "delay": 2000,
                                "httpMethod": "GET"
                                }
                        ]
                }
```

Now run the following to import the edited `simulation.json` file and run the simulation:

```
hoverctl start
hoverctl import simulation.json
curl --proxy localhost:8500 http://echo.jsontest.com/b/c
curl -X POST --proxy localhost:8500 http://echo.jsontest.com/b/c
hoverctl stop
```

You should notice a 2 second delay on the response to the GET request and no delay on the response to the POST request.

Adding templates to a simulation

See also:

Please carefully read through *Templates* alongside this tutorial to gain a high-level understanding of what we are about to cover.

In this tutorial, we are going to go through the steps required to generate and use a matching template.

Let's begin by capturing some traffic and exporting a simulation.

```
hoverctl start
hoverctl mode capture
curl --proxy http://localhost:8500 http://echo.jsontest.com/foo/baz/bar/spam
hoverctl export simulation.json
hoverctl stop
```

Which gives us this output:

```
time="2016-12-22T08:56:24Z" level=info msg="Hoverfly is now running" admin-
↪port=8888 proxy-port=8500
time="2016-12-22T08:56:24Z" level=info msg="Hoverfly has been set to capture␣
↪mode"
  % Total    % Received % Xferd  Average Speed   Time    Time     Time ␣
↪Current
                                 Dload  Upload   Total   Spent    Left  Speed

  0     0    0     0    0     0      0      0 --:--:-- --:--:-- --:--:--    ␣
↪0
```

```
  0     0    0     0    0     0       0      0 --:--:-- --:--:-- --:--:--
↪0
100    38  100    38    0     0    167      0 --:--:-- --:--:-- --:--:--
↪168
{
   "foo": "baz",
   "bar": "spam"
}
time="2016-12-22T08:56:25Z" level=info msg="Successfully exported to
↪simulation.json"
time="2016-12-22T08:56:25Z" level=info msg="Hoverfly has been stopped"
```

If you take a look at your `simulation.json` you should notice these lines in your request.

```
"request": {
        "requestType": "recording",
        "path": "/foo/baz/bar/spam",
        "method": "GET",
```

Modify them to:

```
    "request": {
      "requestType": "template",
      "path": "/foo/*/bar/spam",
      "method": "GET",
```

Save the file as `simulationimport.json` and run the following command to import it and cURL the simulated endpoint:

```
hoverctl start
hoverctl mode simulate
hoverctl import simulationimport.json
curl --proxy http://localhost:8500 http://echo.jsontest.com/foo/QUX/bar/spam
hoverctl stop
```

The same response is returned, even though we created our simulation with a request to `http://echo.jsontest.com/foo/baz/bar/spam` in Capture mode and then sent a request to `http://echo.jsontest.com/foo/QUX/bar/spam` in Simulate mode.

As you can see, templating allows us to match URLs using globbing.

Note: Key points:

- To do templating, capture a simulation, export it, edit it
- While editing, change `"requestTypes"` value to `"template"`
- Substitute strings in URLs with the wildcard `*` to return one response for more than one request
- Re-import the simulation

Using middleware to simulate network latency

See also:

Please carefully read through *Middleware* alongside these tutorials to gain a high-level understanding of what we are about to cover.

We will use a Python script to apply a random delay of less than one second to every response in a simulation.

Before you proceed, please ensure that you have Python installed.

Let's begin by writing our middleware. Save the following as `middleware.py`:

```
#!/usr/bin/env python
import sys
import logging
import random
from time import sleep

logging.basicConfig(filename='random_delay_middleware.log', level=logging.DEBUG)
logging.debug('Random delay middleware is called')

# set delay to random value less than one second

SLEEP_SECS = random.random()

def main():

    data = sys.stdin.readlines()
    # this is a json string in one line so we are interested in that one line
    payload = data[0]
    logging.debug("sleeping for %s seconds" % SLEEP_SECS)
    sleep(SLEEP_SECS)


    # do not modifying payload, returning same one
    print(payload)

if __name__ == "__main__":
    main()
```

The middleware script delays each response by a random value of less than one second.

```
hoverctl start
hoverctl mode capture
curl --proxy http://localhost:8500 http://time.jsontest.com
hoverctl mode simulate
hoverctl middleware --binary python --script middleware.py
curl --proxy http://localhost:8500 http://time.jsontest.com
hoverctl stop
```

Middleware gives you control over the behaviour of a simulation, as well as the data.

Note: Middleware gives you flexibility when simulating network latency - allowing you to randomize the delay value for example - but a new process is spawned every time the middleware script is executed. This can impact Hoverfly's performance under load.

If you need to simulate latency during a load test, it is recommended that you use Hoverfly's native *Delays* functionality to simulate network latency (see *Adding delays to a simulation*) instead of writing middleware. The delays functionality sacrifices flexibility for performance.

Using middleware to modify response payload and status code

See also:

Please carefully read through *Middleware* alongside these tutorials to gain a high-level understanding of what we are about to cover.

We will use a python script to modify the body of a response and randomly change the status code.

Let's begin by writing our middleware. Save the following as `middleware.py`:

```
#!/usr/bin/env python

import sys
import json
import logging
import random

logging.basicConfig(filename='middleware.log', level=logging.DEBUG)
logging.debug('Middleware "modify_request" called')

def main():
    payload = sys.stdin.readlines()[0]

    logging.debug(payload)

    payload_dict = json.loads(payload)
    payload_dict['response']['status'] = random.choice([200, 201])

    if "response" in payload_dict and "body" in payload_dict["response"]:
        payload_dict["response"]["body"] = "{'foo': 'baz'}\n"

    print(json.dumps(payload_dict))

if __name__ == "__main__":
    main()
```

The middleware script randomly toggles the status code between 200 and 201, and changes the response body to a dictionary containing `{'foo':'baz'}`.

```
hoverctl start
hoverctl mode capture
curl --proxy http://localhost:8500 http://time.jsontest.com
hoverctl mode simulate
hoverctl middleware --binary python --script middleware.py
curl --proxy http://localhost:8500 http://time.jsontest.com
hoverctl stop
```

As you can see, middleware allows you to completely modify the content of a simulated HTTP response.

Simulating HTTPS APIs

To capture HTTPS traffic, you need to use Hoverfly's SSL certificate.

First, download the certificate:

```
wget https://raw.githubusercontent.com/SpectoLabs/hoverfly/master/core/cert.pem
```

We can now run Hoverfly with the standard `capture` then `simulate` workflow.

```
hoverctl start
hoverctl mode capture
```

```
curl --proxy https://localhost:8500 https://example.com --cacert cert.pem
hoverctl mode simulate
curl --proxy https://localhost:8500 https://example.com --cacert cert.pem
hoverctl stop
```

Curl was able to make the HTTPS request using an HTTPS proxy because we provided it with Hoverfly's SSL certificate.

Note: This example uses cURL. If you are using Hoverfly in another environment, you will need to add the certificate to your trust store. This is done automatically by the Hoverfly Java library (see *Hoverfly Java*).

See also:

This example uses Hoverfly's default SSL certificate. Alternatively, you can use Hoverfly to generate a new certificate. For more information, see *Configuring SSL in Hoverfly*.

Running Hoverfly as a webserver

See also:

Please carefully read through *Hoverfly as a webserver* alongside this tutorial to gain a high-level understanding of what we are about to cover.

Below is a complete example how to capture data with Hoverfly running as a proxy, and how to save it in a simulation file.

```
hoverctl start
hoverctl mode capture
curl --proxy http://localhost:8500 http://echo.jsontest.com/a/b
hoverctl export simulation.json
hoverctl stop
```

Now we can use Hoverfly as a **webserver** in Simulate mode.

```
hoverctl start webserver
hoverctl import simulation.json
curl http://localhost:8500/a/b
hoverctl stop
```

Hoverfly returned a response to our request while running as a webserver, not as a proxy.

Notice that we issued a cURL command to `http://localhost:8500/a/b` instead of `http://echo.jsontest.com/a/b`. This is because when running as a webserver, Hoverfly strips the domain from the endpoint URL in the simulation.

This is explained in more detail in the *Hoverfly as a webserver* section.

Note: Hoverfly starts in Simulate mode by default.

Capturing or simulating specific URLs

We can use the `hoverctl destination` command to specify which URLs to capture or simulate. The destination setting can be tested using the `--dry-run` flag. This makes it easy to check whether the destination setting will filter out URLs as required.

```
hoverctl start
hoverctl destination "ip" --dry-run http://ip.jsontest.com
hoverctl destination "ip" --dry-run http://time.jsontest.com
hoverctl stop
```

This tells us that setting the destination to `ip` will allow the URL `http://ip.jsontest.com` to be captured or simulated, while the URL `http://time.jsontest.com` will be ignored.

Now we have checked the destination setting, we can apply it to filter out the URLs we don't want to capture.

```
hoverctl start
hoverctl destination "ip"
hoverctl mode capture
curl --proxy http://localhost:8500 http://ip.jsontest.com
curl --proxy http://localhost:8500 http://time.jsontest.com
hoverctl logs
hoverctl export simulation.json
hoverctl stop
```

If we examine the logs and the `simulation.json` file, we can see that *only* a request response pair to the `http://ip.jsontest.com` URL has been captured.

The destination setting can be either a string or a regular expression.

```
hoverctl start
hoverctl destination "^.*api.*com" --dry-run https://api.github.com
hoverctl destination "^.*api.*com" --dry-run https://api.slack.com
hoverctl destination "^.*api.*com" --dry-run https://github.com
hoverctl stop
```

Here, we can see that setting the destination to `^.*api.*com` will allow the `https://api.github.com` and `https://api.slack.com` URLs to be captured or simulated, while the `https://github.com` URL will be ignored.

Advanced tutorials

Using Hoverfly behind a proxy

In some environments, you may only be able to access the internet via a proxy. For example, your organization may route all traffic through a proxy for security reasons.

If this is the case, you will need to configure Hoverfly to work with the 'upstream' proxy.

This configuration value can be easily set when starting an instance of Hoverfly.

For example, if the 'upstream' proxy is running on port `8080` on host `corp.proxy`:

```
hoverctl start --upstream-proxy http://corp.proxy:8080
```

Upstream proxy authentication

If the proxy you are using uses HTTP basic authentication, you can provide the authentication credentials as part of the upstream proxy configuration setting.

For example:

```
hoverctl start --upstream-proxy http://my-user:my-pass@corp.proxy:8080
```

Currently, HTTP basic authentication is the only supported authentication method for an authenticated proxy.

Controlling a remote Hoverfly instance with hoverctl

So far, the tutorials have shown how hoverctl can be used to control an instance of Hoverfly running on the same machine.

In some cases, you may wish to use hoverctl to control an instance of Hoverfly running on a remote host.

In this example, we assume that the remote host is reachable at `hoverfly.example.com`, and that ports `8880` and `8555` are available. We will also assume that the Hoverfly binary is installed on the remote host.

On the **remote host**, start Hoverfly using flags to override the default admin port (`-ap`) and proxy port (`-pp`).

```
hoverfly -ap 8880 -pp 8555
```

See also:

For a full list of all Hoverfly flags, please refer to *Hoverfly commands* in the *Reference* section.

On your **local machine**, edit your `~/.hoverfly/config.yml` so it looks like this:

```
hoverfly.host: hoverfly.example.com
hoverfly.admin.port: "8880"
hoverfly.proxy.port: "8555"
hoverfly.db.type: memory
hoverfly.username: ""
hoverfly.password: ""
hoverfly.webserver: false
hoverfly.tls.certificate: ""
hoverfly.tls.key: ""
hoverfly.tls.disable: false
```

Now that hoverctl knows the location of the remote Hoverfly instance, run the following commands **on your local machine** to capture and simulate a URL using the remote Hoverfly:

```
hoverctl mode capture
curl --proxy http://hoverfly.example.com:8555 http://ip.jsontest.com
hoverctl mode simulate
curl --proxy http://hoverfly.example.com:8555 http://ip.jsontest.com
```

Note: The `hoverfly.host` value in the `config.yml` file allows hoverctl to interact with the **admin API** of the remote Hoverfly instance.

The application that is making the request (in this case, cURL), **also** needs to be configured to use the remote Hoverfly instance as a proxy. In this example, it is done using cURL's `--proxy` flag.

If you are running Hoverfly on a remote host, you may wish to enable authentication on the Hoverfly proxy and admin API. This is described in the *Enabling authentication for the Hoverfly proxy and API* tutorial.

Enabling authentication for the Hoverfly proxy and API

If you are running Hoverfly on a remote host (see *Controlling a remote Hoverfly instance with hoverctl*), you may wish to enable authentication on the Hoverfly proxy and API.

Setting Hoverfly authentication credentials

In this example, we assume that the steps in the *Controlling a remote Hoverfly instance with hoverctl* tutorial have been followed, and that the Hoverfly binary is installed **but not running** on a remote host.

On the **remote host**, run the following command to start Hoverfly with authentication credentials, and the default admin and proxy ports overridden.

```
hoverfly -auth -username my-user -password my-pass -ap 8880 -pp 8555
```

> **Warning:** By default, Hoverfly starts with authentication disabled. If you require authentication you must make sure the `-auth`, `-username` and `-password` flags are supplied every time Hoverfly is started.

Configuring hoverctl

On your **local machine**, edit your `~/.hoverfly/config.yml` so it looks like this:

```
hoverfly.host: hoverfly.example.com
hoverfly.admin.port: "8880"
hoverfly.proxy.port: "8555"
hoverfly.db.type: memory
hoverfly.username: "my-user"
hoverfly.password: "my-pass"
hoverfly.webserver: false
hoverfly.tls.certificate: ""
hoverfly.tls.key: ""
hoverfly.tls.disable: false
```

The `config.yml` file will now tell hoverctl the location of the remote Hoverfly instance, and provide the credentials required to authenticate.

Run the following commands **on your local machine** to capture and simulate a URL using the remote Hoverfly:

```
hoverctl mode capture
curl --proxy http://my-user:my-pass@hoverfly.example.com:8555 http://ip.jsontest.com
hoverctl mode simulate
curl --proxy http://my-user:my-pass@hoverfly.example.com:8555 http://ip.jsontest.com
```

Note: The `hoverfly.username` and `hoverfly.password` values in the `config.yml` file allow hoverctl to authenticate against the admin API of a remote Hoverfly instance.

When using authentication, the Hoverfly proxy port is also authenticated using basic HTTP authentication. This means that any application using Hoverfly (in this example, cURL) must include the authentication credentials as part of the proxy URL.

Configuring SSL in Hoverfly

In some cases, you may not wish to use Hoverfly's default SSL certificate. Hoverfly allows you to generate a new certificate and key.

The following command will start a Hoverfly process and create new `cert.pem` and `key.pem` files in the current working directory. These newly-created files will be loaded into the running Hoverfly instance.

```
hoverfly -generate-ca-cert
```

Optionally, you can provide a custom certificate name and authority:

```
hoverfly -generate-ca-cert -cert-name tutorial.cert -cert-org "Tutorial Certificate␣
↪Authority"
```

Once you have generated `cert.pem` and `key.pem` files with Hoverfly, you can use hoverctl to start an instance of Hoverfly using these files.

```
hoverctl start --certificate cert.pem --key key.pem
```

Note: Both a certificate and a key file must be supplied. The files must be in unencrypted PEM format.

Reference

These reference documents contain information regarding invoking the hoverctl command, hoverfly command, and interacting with the APIs.

hoverctl commands

This page contains the output of:

```
hoverctl --help
```

The command's help content has been placed here for convenience.

```
hoverctl is the command line tool for Hoverfly

Usage:
  hoverctl [command]

Available Commands:
  config      Show hoverctl configuration information
  delete      Delete Hoverfly simulation
  destination Get and set Hoverfly destination
  export      Export a simulation from Hoverfly
  import      Import a simulation into Hoverfly
  logs        Get the logs from Hoverfly
  middleware  Get and set Hoverfly middleware
  mode        Get and set the Hoverfly mode
  start       Start Hoverfly
  stop        Stop Hoverfly
  version     Get the version of hoverctl

Flags:
      --admin-port string       A port number for the Hoverfly API/GUI. Overrides the␣
↪default Hoverfly admin port (8888)
      --certificate string      A path to a certificate file. Overrides the default␣
↪Hoverfly certificate
      --database string         A database type [memory|boltdb]. Overrides the␣
↪default Hoverfly database type (memory)
```

```
      --disable-tls             Disables TLS verification
  -h, --help                    help for hoverctl
      --host string             A host on which a Hoverfly instance is running.␣
↪Overrides the default Hoverfly host (localhost)
      --key string              A path to a key file. Overrides the default Hoverfly␣
↪TLS key
      --proxy-port string       A port number for the Hoverfly proxy. Overrides the␣
↪default Hoverfly proxy port (8500)
      --upstream-proxy string   A host for which Hoverfly will proxy its requests to
  -v, --verbose                 Verbose logging from hoverctl

Use "hoverctl [command] --help" for more information about a command.
```

Hoverfly commands

This page contains the output of:

```
hoverfly --help
```

The command's help content has been placed here for convenience.

```
Usage of hoverfly:
    -add
        add new user '-add -username hfadmin -password hfpass'
    -admin
        supply '-admin false' to make this non admin user (defaults to 'true') ␣
↪(default true)
    -ap string
        admin port - run admin interface on another port (i.e. '-ap 1234' to run␣
↪admin UI on port 1234)
    -auth
        enable authentication, currently it is disabled by default
    -capture
        start Hoverfly in capture mode - transparently intercepts and saves requests/
↪response
    -cert string
        CA certificate used to sign MITM certificates
    -cert-name string
        cert name (default "hoverfly.proxy")
    -cert-org string
        organisation name for new cert (default "Hoverfly Authority")
    -db string
        Persistance storage to use - 'boltdb' or 'memory' which will not write␣
↪anything to disk (default "boltdb")
    -db-path string
        database location - supply it to provide specific database location (will be␣
↪created there if it doesn't exist)
    -dest value
        specify which hosts to process (i.e. '-dest fooservice.org -dest barservice.
↪org -dest catservice.org') - other hosts will be ignored will passthrough'
    -destination string
        destination URI to catch (default ".")
    -dev
        supply -dev flag to serve directly from ./static/dist instead from statik␣
↪binary
    -generate-ca-cert
```

```
        generate CA certificate and private key for MITM
    -httptest.serve string
        if non-empty, httptest.NewServer serves on this address and blocks
    -import value
        import from file or from URL (i.e. '-import my_service.json' or '-import␣
↪http://mypage.com/service_x.json'
    -key string
        private key of the CA used to sign MITM certificates
    -metrics
        supply -metrics flag to enable metrics logging to stdout
    -middleware string
        should proxy use middleware
    -modify
        start Hoverfly in modify mode - applies middleware (required) to both␣
↪outgoing and incomming HTTP traffic
    -password string
        password for new user
    -pp string
        proxy port - run proxy on another port (i.e. '-pp 9999' to run proxy on port␣
↪9999)
    -synthesize
        start Hoverfly in synthesize mode (middleware is required)
    -test.bench string
        regular expression per path component to select benchmarks to run
    -test.benchmem
        print memory allocations for benchmarks
    -test.benchtime duration
        approximate run time for each benchmark (default 1s)
    -test.blockprofile string
        write a goroutine blocking profile to the named file after execution
    -test.blockprofilerate int
        if >= 0, calls runtime.SetBlockProfileRate() (default 1)
    -test.count n
        run tests and benchmarks n times (default 1)
    -test.coverprofile string
        write a coverage profile to the named file after execution
    -test.cpu string
        comma-separated list of number of CPUs to use for each test
    -test.cpuprofile string
        write a cpu profile to the named file during execution
    -test.memprofile string
        write a memory profile to the named file after execution
    -test.memprofilerate int
        if >=0, sets runtime.MemProfileRate
    -test.outputdir string
        directory in which to write profiles
    -test.parallel int
        maximum test parallelism (default 8)
    -test.run string
        regular expression to select tests and examples to run
    -test.short
        run smaller test suite to save time
    -test.timeout duration
        if positive, sets an aggregate time limit for all tests
    -test.trace string
        write an execution trace to the named file after execution
    -test.v
        verbose: print additional output
```

```
    -tls-verification
        turn on/off tls verification for outgoing requests (will not try to verify␣
↪certificates) - defaults to true (default true)
    -username string
        username for new user
    -v        should every proxy request be logged to stdout
    -version
        get the version of hoverfly
    -webserver
        start Hoverfly in webserver mode (simulate mode)
```

REST API

GET /api/v2/simulation

Gets all simulation data. The simulation JSON contains all the information Hoverfly can hold; this includes recordings, templates, delays and metadata.

Example response body:

```
{
  "data": {
    "pairs": [
      {
        "response": {
          "status": 200,
          "body": "<h1>Matched on recording</h1>",
          "encodedBody": false,
          "headers": {
            "Content-Type": [
              "text/html; charset=utf-8"
            ]
          }
        },
        "request": {
          "requestType": "recording",
          "path": "/",
          "method": "GET",
          "destination": "myhost.io",
          "scheme": "https",
          "query": "",
          "body": "",
          "headers": {
            "Accept": [
              "text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;
↪q=0.8"
            ],
            "Content-Type": [
              "text/plain; charset=utf-8"
            ],
            "User-Agent": [
              "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_11_6) AppleWebKit/537.36␣
↪(KHTML, like Gecko) Chrome/52.0.2743.116 Safari/537.36"
            ]
          }
        }
```

```
      },
      {
        "response": {
          "status": 200,
          "body": "<h1>Matched on template</h1>",
          "encodedBody": false,
          "headers": {
            "Content-Type": [
              "text/html; charset=utf-8"
            ]
          }
        },
        "request": {
          "requestType": "template",
          "path": "/template",
          "method": null,
          "destination": null,
          "scheme": null,
          "query": null,
          "body": null,
          "headers": null
        }
      }
    ],
    "globalActions": {
      "delays": []
    }
  },
  "meta": {
    "schemaVersion": "v1",
    "hoverflyVersion": "v0.9.0",
    "timeExported": "2016-11-11T11:53:52Z"
  }
```

PUT /api/v2/simulation

This puts the supplied simulation JSON into Hoverfly, overwriting any existing simulation data.

Example request body:

```
{
  "data": {
    "pairs": [
      {
        "response": {
          "status": 200,
          "body": "<h1>Matched on recording</h1>",
          "encodedBody": false,
          "headers": {
            "Content-Type": [
              "text/html; charset=utf-8"
            ]
          }
        },
        "request": {
          "requestType": "recording",
          "path": "/",
```

```
          "method": "GET",
          "destination": "myhost.io",
          "scheme": "https",
          "query": "",
          "body": "",
          "headers": {
            "Accept": [
              "text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;
↪q=0.8"
            ],
            "Content-Type": [
              "text/plain; charset=utf-8"
            ],
            "User-Agent": [
              "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_11_6) AppleWebKit/537.36␣
↪(KHTML, like Gecko) Chrome/52.0.2743.116 Safari/537.36"
            ]
          }
        }
      },
      {
        "response": {
          "status": 200,
          "body": "<h1>Matched on template</h1>",
          "encodedBody": false,
          "headers": {
            "Content-Type": [
              "text/html; charset=utf-8"
            ]
          }
        },
        "request": {
          "requestType": "template",
          "path": "/template",
          "method": null,
          "destination": null,
          "scheme": null,
          "query": null,
          "body": null,
          "headers": null
        }
      }
    ],
    "globalActions": {
      "delays": []
    }
  },
  "meta": {
    "schemaVersion": "v1",
    "hoverflyVersion": "v0.9.0",
    "timeExported": "2016-11-11T11:53:52Z"
  }
```

GET /api/v2/hoverfly

Gets configuration information from the running instance of Hoverfly.

Example response body:

```
{
    "destination": ".",
    "middleware": {
            "binary": "python",
            "script": "# a python script would go here",
            "remote": ""
    },
    "mode": "simulate",
    "usage": {
        "counters": {
            "capture": 0,
            "modify": 0,
            "simulate": 0,
            "synthesize": 0
        }
    }
}
```

GET /api/v2/hoverfly/destination

Gets the current destination setting for the running instance of Hoverfly.

Example response body:

```
{
    destination: "."
}
```

PUT /api/v2/hoverfly/destination

Sets a new destination for the running instance of Hoverfly, overwriting the existing destination setting.

Example request body:

```
{
    destination: "new-destination"
}
```

GET /api/v2/hoverfly/middleware

Gets the middleware settings for the running instance of Hoverfly. This could be either an executable binary, a script that can be executed with a binary or a URL to remote middleware.

Example response body:

```
{
    "binary": "python",
    "script": "#python code goes here",
    "remote": ""
}
```

PUT /api/v2/hoverfly/middleware

Sets new middleware, overwriting the existing middleware for the running instance of Hoverfly. The middleware being set can be either an executable binary located on the host, a script and the binary to execute it or the URL to a remote middleware.

Example request body:

```
{
    "binary": "python",
    "script": "#python code goes here",
    "remote": ""
}
```

GET /api/v2/hoverfly/mode

Gets the mode for the running instance of Hoverfly.

Example response body:

```
{
    mode: "simulate"
}
```

PUT /api/v2/hoverfly/mode

Changes the mode of the running instance of Hoverfly.

Example request body:

```
{
    mode: "simulate"
}
```

GET /api/v2/hoverfly/usage

Gets metrics information for the running instance of Hoverfly.

Example response body:

```
{
    "metrics": {
        "counters": {
            "capture": 0,
            "modify": 0,
            "simulate": 0,
            "synthesize": 0
        }
    }
}
```

GET /api/v2/hoverfly/version

Gets the version of Hoverfly.

Example response body:

```
{
    "version": "v0.10.1"
}
```

GET /api/v2/hoverfly/upstream-proxy

Gets the upstream proxy configured for Hoverfly.

Example response body:

```
{
    "upstream-proxy": "proxy.corp.big-it-company.org:8080"
}
```

Simulation schema

```
{
  "data": {
    "pairs": [
      {
        "response": {
          "status": 200,
          "body": "<h1>Matched on recording</h1>",
          "encodedBody": false,
          "headers": {
            "Content-Type": [
              "text/html; charset=utf-8"
            ]
          }
        },
        "request": {
          "requestType": "recording",
          "path": "/",
```

```
          "method": "GET",
          "destination": "myhost.io",
          "scheme": "https",
          "query": "",
          "body": "",
          "headers": {
            "Accept": [
              "text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;
↪q=0.8"
            ],
            "Content-Type": [
              "text/plain; charset=utf-8"
            ],
            "User-Agent": [
              "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_11_6) AppleWebKit/537.36␣
↪(KHTML, like Gecko) Chrome/52.0.2743.116 Safari/537.36"
            ]
          }
        }
      },
      {
        "response": {
          "status": 200,
          "body": "<h1>Matched on template</h1>",
          "encodedBody": false,
          "headers": {
            "Content-Type": [
              "text/html; charset=utf-8"
            ]
          }
        },
        "request": {
          "requestType": "template",
          "path": "/template",
          "method": null,
          "destination": null,
          "scheme": null,
          "query": null,
          "body": null,
          "headers": null
        }
      }
    ],
    "globalActions": {
      "delays": []
    }
  },
  "meta": {
    "schemaVersion": "v1",
    "hoverflyVersion": "v0.9.0",
    "timeExported": "2016-11-11T11:53:52Z"
  }
```


Contributing

Contributions are welcome! To contribute, please:

1. Fork the repository
2. Create a feature branch on your fork
3. Commit your changes, and create a pull request against Hoverfly's master branch

In your pull request, please include details regarding your change (why you made the change, how to test it etc).

Learn more about the forking workflow here.

Building, running & testing

You will need Go 1.8 . Instructions on how to set up your Go environment can be found here.

```
cd $GOPATH/src
mkdir -p github.com/SpectoLabs/
cd github.com/SpectoLabs/
git clone https://github.com/SpectoLabs/hoverfly.git
# or: git clone https://github.com/<your_username>/hoverfly.git
cd hoverfly
make build
```

Notice the binaries are in the `target` directory.

Finally, to test your build:

```
make test
```

Community

- Chat on the Hoverfly Gitter channel
- Joing the Hoverfly mailing list
- Raise a GitHub Issue
- Get in touch with SpectoLabs on Twitter