Hoverfly Documentation

Release v0.17.0

SpectoLabs

Jun 04, 2018

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
```

Docker

If you have Docker, you can run Hoverfly using the `docker` command.

```
docker run -d -p 8888:8888 -p 8500:8500 spectolabs/hoverfly:latest
```

This will run the latest version of the Hoverfly Docker image. This Docker image does not contain hoverctl. Our recommendation is to have hoverctl on your host machine and then configure hoverctl to use the newly started Hoverfly Docker instance as a new target.

See also:

For a tutorial of creating a new target in hoverctl, see *Controlling a remote Hoverfly instance with hoverctl*.

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.

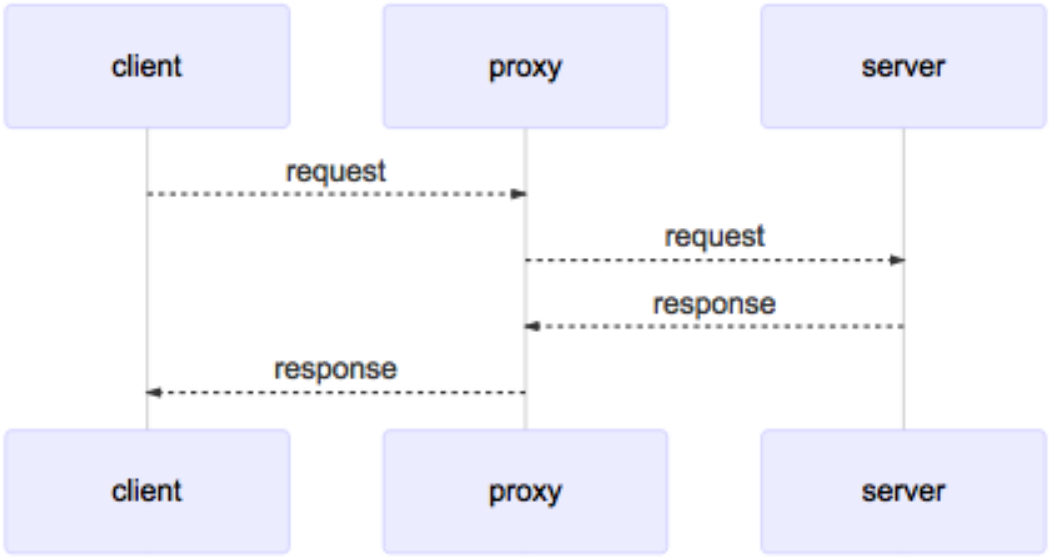
By default Hoverfly starts as a proxy server.

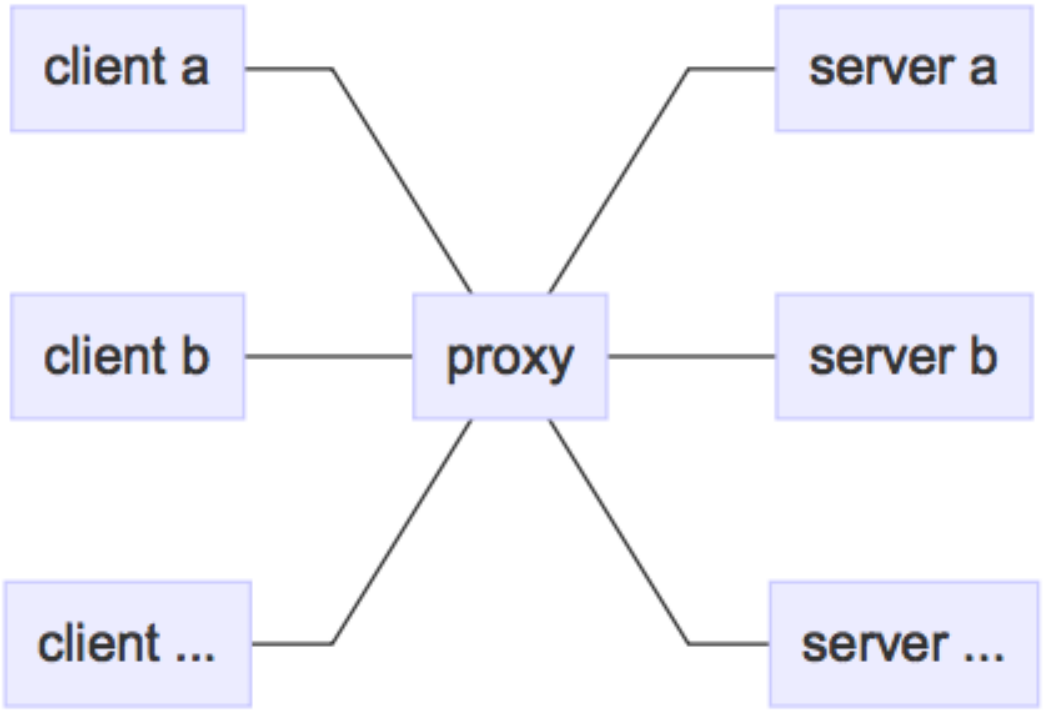
Using a proxy server

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"
```

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.


client
proxy
server
request
request
response
response
client
proxy
server


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

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

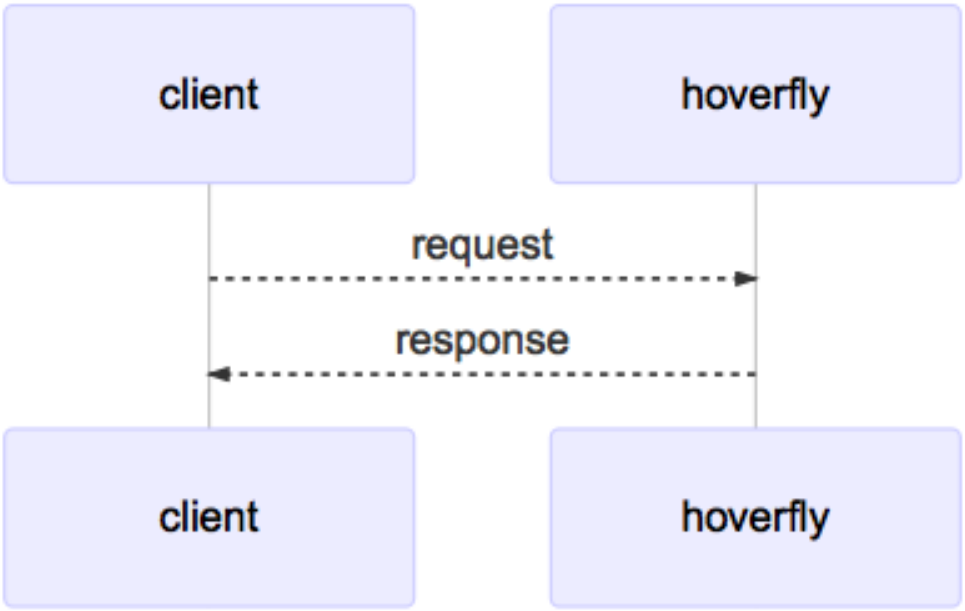
The difference between a proxy server and a webserver

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.

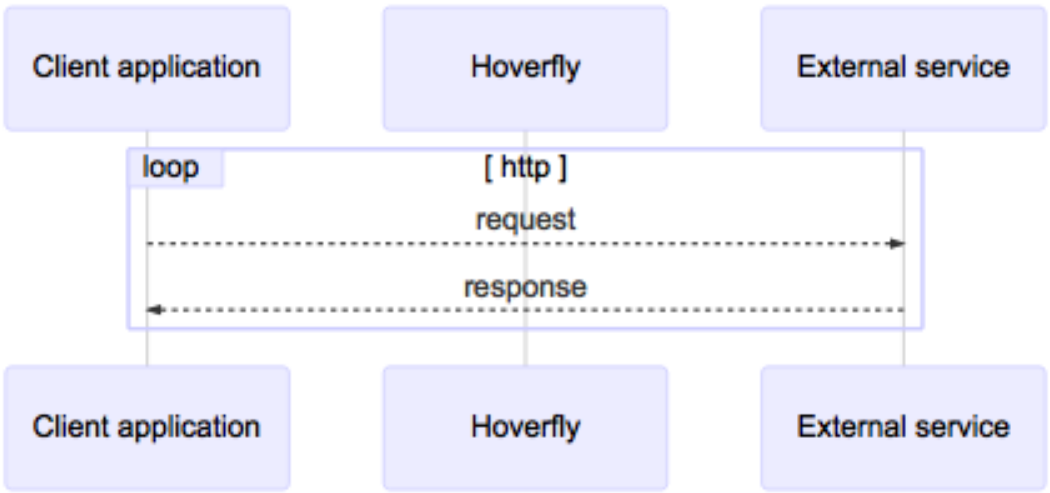
Hoverfly modes

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

Capture mode

Capture mode is used for creating API simulations.

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



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.

By default, Hoverfly will overwrite duplicate requests if the request has not changed. This can be a problem when trying to capture a stateful endpoint that may return a different response each time you make a request.

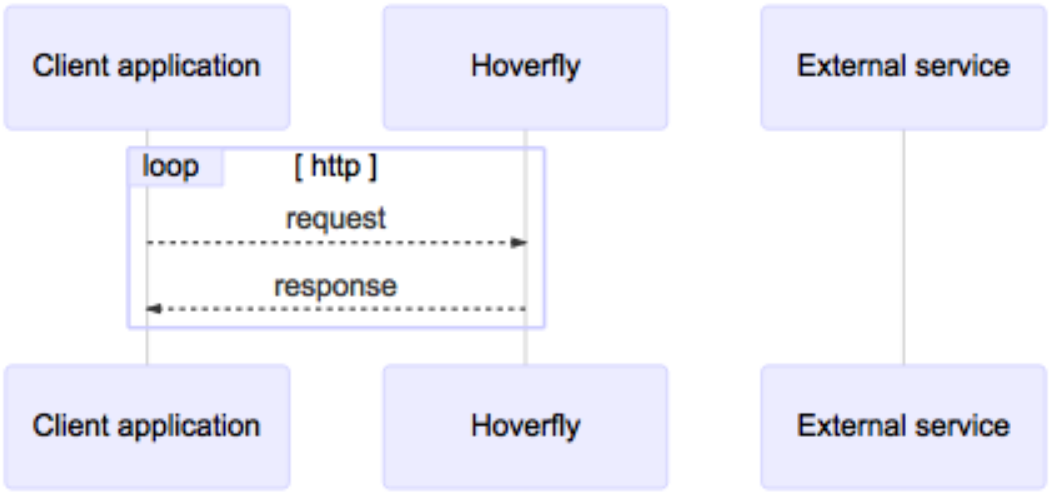
Using the stateful mode argument when setting Hoverfly to capture mode will disable this duplicate overwrite feature, enabling you to capture sequences of responses and play them back in *Simulate mode* in order.

See also:

This functionality is best understood via a practical example: see *Capturing a sequence of responses* in the *Tutorials* section.

Simulate mode

In this mode, Hoverfly uses its simulation data in order to simulate external APIs. Each time Hoverfly receives a request, rather than forwarding it on to the real API, it will respond instead. No network traffic will ever reach the real external API.



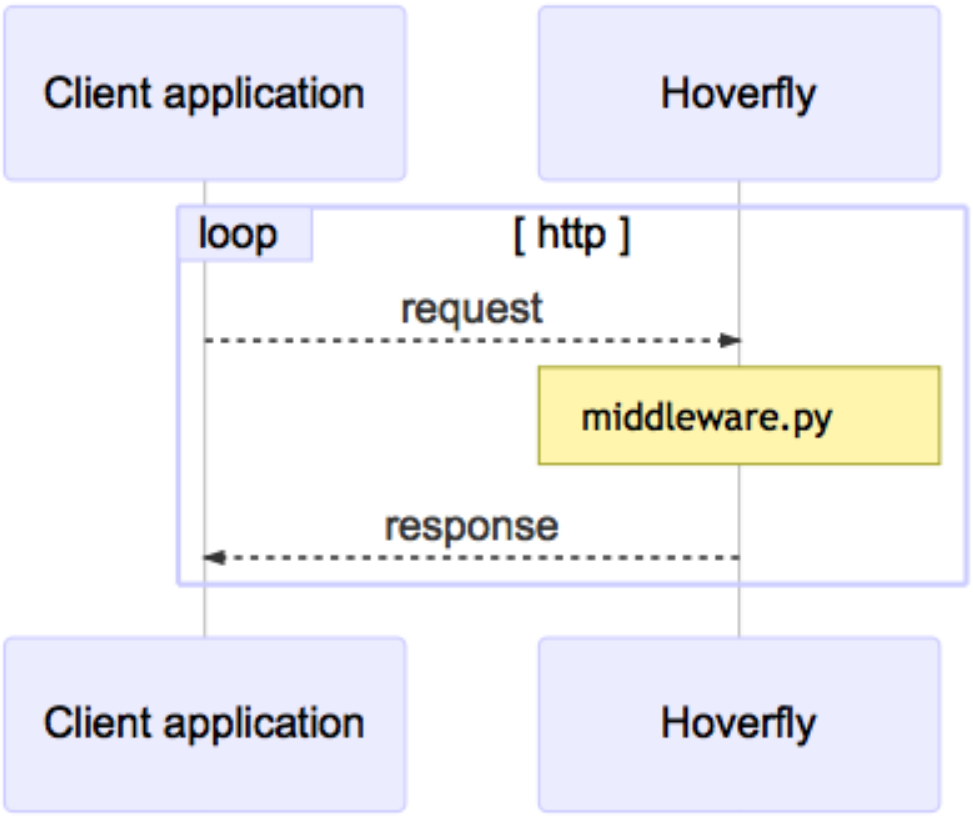
The simulation can be produced automatically via by running Hoverfly in *Capture mode*, or created manually. See *Simulations* for information.

Spy mode

In this mode, Hoverfly simulates external APIs if a request match is found in simulation data (See *Simulate mode*), otherwise, the request will be passed through to the real API.

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*.



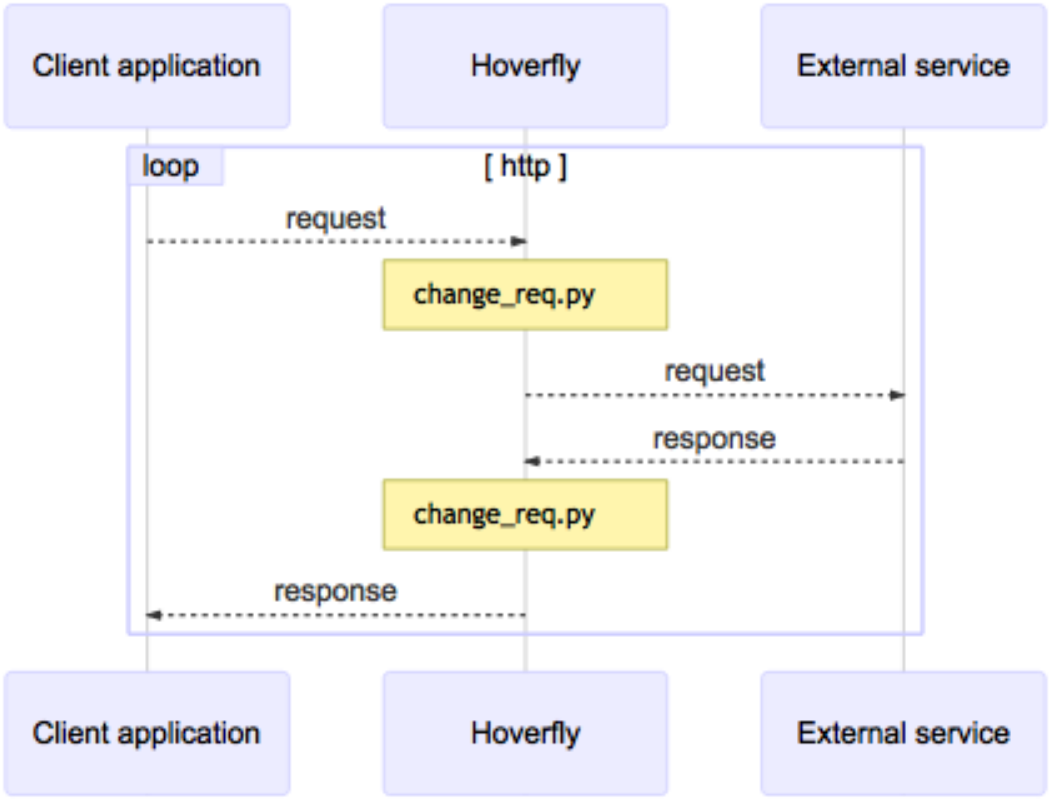
In Synthesize mode, the middleware executable is expected to generate a response to the incoming request "on the fly". Hoverfly will then return the generated response to the application. For Hoverfly to operate in Synthesize mode, a middleware executable must be specified.

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.

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.


Client application
Hoverfly
External service
loop
[http]
request
change_req.py
request
response
change_req.py
response
Client application
Hoverfly
External service

Diff mode

In this mode, Hoverfly forwards a request to an external service and compares a response with currently stored simulation. With both the stored simulation response and the real response from the external service, Hoverfly is able to detect differences between the two. When Hoverfly has finished comparing the two responses, the difference is stored and the incoming request is served the real response from the external service.

The differences can be retrieved from Hoverfly using the API (*GET /api/v2/diff*). The response contains a list of differences, containing the request and the differences of the response.


```
{
  "diff": [{
    "request": {
      "method": "GET",
      "host": "time.jsontest.com",
      "path": "/",
      "query": ""
    },
    "diffReports": [{
      "timestamp": "2018-03-16T17:45:40Z",
      "diffEntries": [{
        "field": "header/X-Cloud-Trace-Context",
        "expected": "[ec6c455330b682c3038ba365ade6652a]",
        "actual": "[043c9bb2eafa1974bc09af654ef15dc3]"
      }, {
        "field": "header/Date",
        "expected": "[Fri, 16 Mar 2018 17:45:34 GMT]",
        "actual": "[Fri, 16 Mar 2018 17:45:41 GMT]"
      }, {
        "field": "body/time",
        "expected": "05:45:34 PM",
        "actual": "05:45:41 PM"
      }, {
        "field": "body/milliseconds_since_epoch",
        "expected": "1.521222334104e+12",
        "actual": "1.521222341017e+12"
      }]
    }]
  }]
}
```


This data is stored and kept until the Hoverfly instance is stopped or the the storage is cleaned by calling the API (*DELETE /api/v2/diff*).

See also:

For more information on the API to retrieve differences, see *REST 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*.

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*.

Simulations consist of **Request Matchers and Responses**, **Delays** and **Metadata** ("Meta").

Request Responses Pairs

Hoverfly simulates APIs by matching **incoming requests** from the client to **stored requests**. Stored requests have an associated **stored response** which is returned to the client if the match is successful.

The matching logic that Hoverfly uses to compare incoming requests with stored requests can be configured using **Request Matchers**.

Request Matchers

When Hoverfly captures a request, it creates a Request Matcher for each field in the request. A Request Matcher consists of the request field name, the type of match which will be used to compare the field in the incoming request to the field in the stored request, and the request field value.

By default, Hoverfly will set the type of match to `exact` for each field.

See also:

There are many types of Request Matcher. Please refer to *Request matchers* for a list of the types available, and examples of how to use them.

There alse are two different matching strategies: **strongest match** (default) and **first match** (legacy). Please refer to *Matching strategies* for more information.

An example Request Matcher Set might look like this:

Field	Matcher Type	Value
scheme	exact	"https"
method	exact	"GET"
destination	exact	"docs.hoverfly.io"
path	exact	"/pages/keyconcepts/templates.html"
query	exact	"query=true"
body	exact	""
headers	exact	

In the Hoverfly simulation JSON file, this Request Matcher Set would be represented like this:

```
            "request": {
               "path": [
                  {
                     "matcher": "exact",
                     "value": "/pages/keyconcepts/templates.html"
                  }
               ],
               "method": [
                  {
                     "matcher": "exact",
                     "value": "GET"
                  }
               ],
               "destination": [
                  {
                     "matcher": "exact",
                     "value": "docs.hoverfly.io"
                  }
               ],
               "scheme": [
                  {
```

```
                    "matcher": "exact",
                    "value": "http"
                }
            ],
            "body": [
                {
                    "matcher": "exact",
                    "value": ""
                }
            ],
            "query": {
                "query": [
                    {
                        "matcher": "exact",
                        "value": "true"
                    }
                ]
            }
        },
```

View entire simulation file

The matching logic that Hoverfly uses to compare an incoming request to a stored request can be changed by editing the Request Matchers in the simulation JSON file.

It is not necessary to have a Request Matcher for every request field. By omitting Request Matchers, it is possible to implement **partial matching** - meaning that Hoverfly will return one stored response for multiple incoming requests.

For example, this Request Matcher will match any incoming request to the `docs.hoverfly.io` destination:

```
            "destination": [
                {
                    "matcher": "exact",
                    "value": "docs.hoverfly.io"
                }
            ]
```

View entire simulation file

In the example below, the `globMatch` Request Matcher type is used to match any subdomain of `hoverfly.io`:

```
            "destination": [
                {
                    "matcher": "glob",
                    "value": "*.hoverfly.io"
                }
            ]
```

View entire simulation file

It is also possible to use more than one Request Matcher for each field.

In the example below, a `regexMatch` **and** a `globMatch` are used on the `destination` field.

This will match on any subdomain of `hoverfly.io` which begins with the letter `d`. This means that incoming requests to `docs.hoverfly.io` and `dogs.hoverfly.io` will be matched, but requests to `cats.hoverfly.io` will not be matched.

```
            "destination": [
                {
```

```
                    "matcher": "glob",
                    "value": "*.hoverfly.io"
                },
                {
                    "matcher": "regex",
                    "value": "(\\Ad)"
                }
            ]
```

View entire simulation file

See also:

There are many types of Request Matcher. Please refer to *Request matchers* for a list of the types available, and examples of how to use them.

For a practical example of how to use a Request Matcher, please refer to *Loose request matching using a Request Matcher* in the tutorials section.

There alse are two different matching strategies: **strongest match** (default) and **first match** (legacy). Please refer to *Matching strategies* for more information.

Responses

Each Request Matcher Set has a response associated with is. If the request match is successful, Hoverfly will return the response to the client.

```
        "response": {
            "status": 200,
            "body": "Response from docs.hoverfly.io/pages/keyconcepts/templates.
↪html",
            "encodedBody": false,
            "headers": {
                "Hoverfly": [
                    "Was-Here"
                ]
            },
            "templated": false
        }
```

View entire simulation file

Editing the fields in response, combined with editing the Request Matcher set, makes it possible to configure complex request/response logic.

Binary data in responses

JSON is a text-based file format so it has no intrinsic support for binary data. Therefore if Hoverfly a response body contains binary data (images, gzipped, etc), the response body will be base64 encoded and the *encodedBody* field set to true.

```
            "body": "YmFzZTY0IGVuY29kZWQ=",
            "encodedBody": true,
```

View entire simulation file

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.

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.

```
1  "meta": {
2     "schemaVersion": "v5",
3     "hoverflyVersion": "v0.17.0",
4     "timeExported": "2018-06-04T10:29:57+01:00"
5  }
```

See also:

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

Matching strategies

Hoverfly has two matching strategies. Each has advantages and trade-offs.

Note: In order to fully understand Hoverfly's matching strategies, it is recommended that you read the *Simulations* section first.

Strongest Match

This is the default matching strategy for Hoverfly. If Hoverfly finds multiple Request Response Pairs that match an incoming request, it will return the Response from the pair which has the highest **matching score**.

To set "strongest" as the matching strategy, simply run:

```
hoverctl mode simulate
```

Or to be explicit run:

```
hoverctl mode simulate --matching-strategy=strongest
```

Matching scores

This example shows how matching scores are calculated.

Let's assume Hoverfly is running in simulate mode, and the simulation data contains four *Request Responses Pairs*. Each Request Response Pair contains one or more *Request Matchers*.

Hoverfly then receives a `GET` request to the destination `www.destination.com`. The incoming request contains the following fields.

Request

Field	Value
method	GET
destination	www.destination.com

Request Response Pair 1

Field	Matcher Type	Value	Score	Total Score	Matched?
method	exact	DELETE	+0	1	false
destination	exact	www.destination.com	+1		

This pair contains two Request Matchers. The **method** value in the incoming request (`GET`) does not match the value for the **method** matcher (`DELETE`). However the **destination** value does match.

This gives the Request Response Pair a total score of 1, but since one match failed, it is treated as unmatched (**Matched?** = `false`).

Request Response Pair 2

Field	Matcher Type	Value	Score	Total Score	Matched?
method	exact	GET	+1	1	true

This pair contains one Request Matcher. The **method** value in the incoming request (`GET`) matches the value for the **method** matcher. This gives the pair a total score of 1, and since no matches failed, it is treated as matched.

Request Response Pair 3

Field	Matcher Type	Value	Score	Total Score	Matched?
method	exact	GET	+1	2	true
destination	exact	www.destination.com	+1		

In this pair, the **method** and **destination** values in the incoming request both match the corresponding Request Matcher values. This gives the pair a total score of 2, and it treated as matched.

Request Response Pair 4

Field	Matcher Type	Value	Score	Total Score	Matched?
method	exact	GET	+1	1	false
destination	exact	www.miss.com	+0		

This pair is treated as unmatched because the **destination** matcher failed.

Request Response Pair 3 has the highest score, and is therefore the **strongest match**.

This means that Hoverfly will return the Response contained within Request Response Pair 3.

Note: When there are multiple matches all with the same score, Hoverfly will pick the last one in the simulation.

The strongest match strategy makes it much easier to identify why Hoverfly has not returned a Response to an incoming Request. If Hoverfly is not able to match an incoming Request to a Request Response Pair, it will return the closest match. For more information see *Troubleshooting*.

However, the additional logic required to calculate matching scores does affect Hoverfly's performance.

First Match

First match is the alternative (legacy) mechanism of matching. There is no scoring, and Hoverfly simply returns the first match it finds in the simulation data.

To set first match as the matching strategy, run:

```
hoverctl mode simulate --matching-strategy=first
```

The main advantage of this strategy is performance - although it makes debugging matching errors harder.

Caching

In *Simulate mode*, Hoverfly uses caching in order to retain strong performance characteristics, even in the event of complex matching. The cache is a key-value store of request hashes (hash of all the request excluding headers) to responses.

Caching matches

When Hoverfly receives a request in simulate mode, it will first hash it and then look for it in the cache. If a cache entry is found, it will send the cached response back to Hoverfly. If it is not found, it will look for a match in the list of matchers. Whenever a new match is found, it will be added to the cache.

Caches misses

Hoverfly also caches misses. This means that repeating a request which was not matched will return a cached miss, avoiding the need to perform matching. The closest miss is also cached, so Hoverfly will not lose any useful information about which matchers came closest.

Header caching

Currently, headers are not included in the hash for a request. This is because headers tend to change across time and clients, impacting ordinary and eager caching respectively.

Eager caching

The cache is automatically pre-populated whenever switching to simulate mode. This only works on certain matchers (such as matchers where every field is an "exactMatch"), but it means the initial cache population does not happen **during** simulate mode.

Cache invalidation

Cache invalidation is a straightforward process in Hoverfly. It only occurs when a simulation is modified.

Templating

Hoverfly can build responses dynamically through templating. This is particularly useful when combined with loose matching, as it allows a single matcher to represent an unlimited combination of responses.

Enabling Templating

By default templating is disabled. In order to enable it, set the flag to true in the response of a simulation.

Available Data

Currently, the following data is available through templating:

Field	Example	Request	Result
Request scheme	{{ Request.Scheme }}	http://www.foo.com	http
Query parameter value	{{ Request.QueryParam.myParam }}	http://www.foo.com?myParam=bar	bar
Query parameter value (list)	{{ Request.QueryParam.NameOfParameter.[1] }}	http://www.foo.com?myParam=bar1&myParam=bar2	bar2
Path parameter value	{{ Request.Path.[1] }}	http://www.foo.com/zero/one/two	one
State	{{ State.basket }}	State Store = {"basket":"eggs"}	eggs

Helper Methods

Additional data can come from helper methods. Current we only have some for the current data, but this list is likely to expand:

Description	Example	Result
The current date, formatted in iso8601	{{ iso8601DateTime }}	2006-01-02T15:04:05Z07:00
The current date, formatted in iso8601, with days added	{{ iso8601DateTimePlusDays Request.QueryParam.plusDays }}	2006-02-02T15:04:05Z07:00

Conditional Templating, Looping and More

Hoverfly uses the https://github.com/aymerick/raymond library for templating, which is based on http://handlebarsjs.com/

To learn about more advanced templating functionality, such as looping and conditionals, read the documentation for these projects.

State

Hoverfly contains a map of keys and values which it uses to store it's internal state. Some *Request matchers* can be made to only match when Hoverfly is in a certain state, and other matchers can be set to mutate Hoverfly's state.

Setting State when Performing a Match

A response includes two fields, *transitionsState* and *removesState* which alter Hoverflies internal state during a match:

```
"request": {
    "path": [
        {
            "matcher": "exact",
            "value": "/pay"
        }
    ]
},
"response": {
    "status": 200,
    "body": "eggs and large bacon",
    "transitionsState" : {
        "payment-flow" : "complete",
    },
    "removesState" : [
        "basket"
    ]
}
```

In the above case, the following changes to Hoverflies internal state would be made on a match:

Current State of Hoverfly	New State of Hoverfly?	reason
payment-flow=pending,basket=full	payment-flow=complete	Payment value transitions, basket deleted by key
basket=full	payment-flow=complete	Payment value created, basket deleted by key
	payment-flow=complete	Payment value created, basket already absent

Requiring State in order to Match

A matcher can include a field *requiresState*, which dictates the state Hoverfly must be in for there to be a match:

```
"request": {
    "path": [
        {
            "matcher": "exact",
            "value": "/basket"
        }
    ]
    "requiresState": {
        "eggs": "present",
        "bacon" : "large"
    }
},
"response": {
    "status": 200,
    "body": "eggs and large bacon"
}
```

In the above case, the following matches results would occur when making a request to */basket*:

Current State of Hoverfly	matches?	reason
eggs=present,bacon=large	true	Required and current state are equal
eggs=present,bacon=large,f=x	true	Additional state 'f=x' is not used by this matcher
eggs=present	false	Bacon is missing
eggs=present,bacon=small	false	Bacon is has the wrong value

Managing state via Hoverctl

It could be tricky to reason about the current state of Hoverfly, or to get Hoverfly in a state that you desire for testing. This is why Hoverctl comes with commands that let you orchestrate it's state. Some useful commands are:

```
$ hoverctl state --help
$ hoverctl state get-all
$ hoverctl state get key
$ hoverctl state set key value
$ hoverctl state delete-all
```

Sequences

Using state, it is possible to recreate a sequence of different responses that may come back given a single request. This can be useful when trying to test stateful endpoints.

When defining state for request response pairs, if you prefix your state key with the string `sequence:`, Hoverfly will acknowledge the pair as being part of a stateful sequence. When simulating this sequence, Hoverfly will keep track of the user's position in the sequence and move them forwards.

Once Hoverfly has reached the end of the sequence, it will continue to return the final response.

```
{
    "request": {
        "requiresState": {
            "sequence:1": "1"
        }
    },
    "response": {
        "status": 200,
        "body": "First response",
        "transitionsState" : {
            "sequence:1" : "2",
        }
    }
    "request": {
        "requiresState": {
            "sequence:1": "2"
        }
    },
    "response": {
        "status": 200,
        "body": "Second response",
    }
}
```

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.

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. There are two different types of middleware.

Local Middleware

Hoverfly has the ability to invoke middleware by executing a script or binary file on a host operating system. The only requires are that the provided middleware can be executed and sends the Middleware JSON schema to stdout when the Middleware JSON schema is received on stdin.

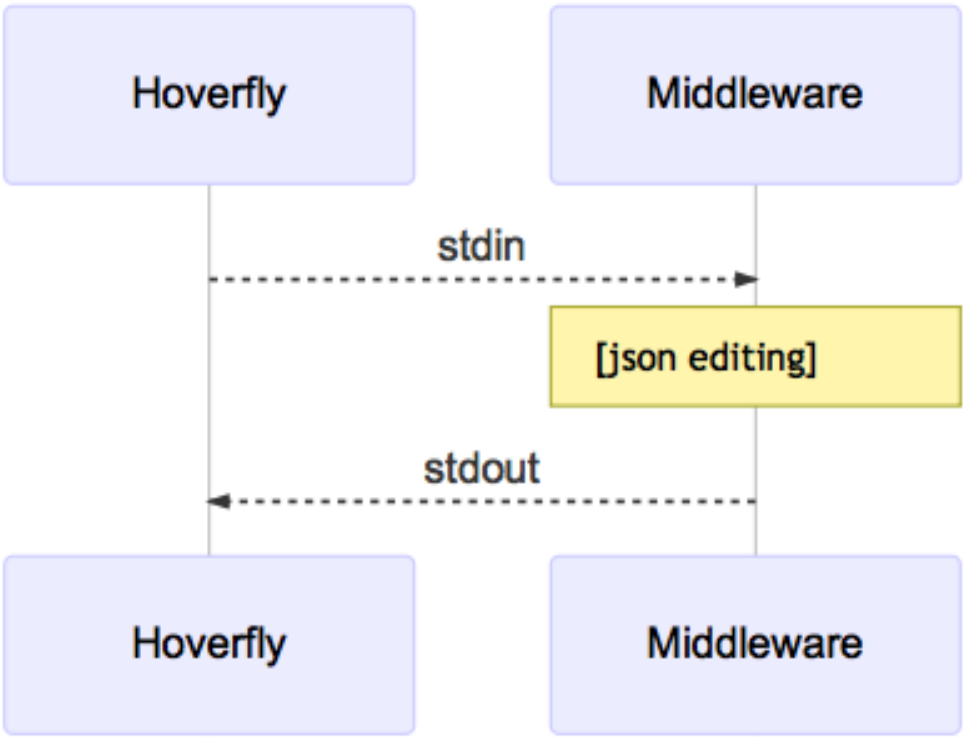
HTTP Middleware

Hoverfly can also send middleware requests to a HTTP server instead of running a process locally. The benefits of this are that Hoverfly does not initiate the process, giving more control to the user. The only requirements are that Hoverfly

can POST the Middleware JSON schema to middleware URL provided and the middleware HTTP server respones with a 200 and the Middleware JSON schema is in the response.

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*.

Hoverctl

Hoverctl is a command line utility that is shipped with Hoverfly. Its purpose is to make it easier to interact with Hoverfly APIs and your local filesystem, while also providing a way to start and stop instances of Hoverfly.

Hoverctl also has the ability to work with multiple instances of Hoverfly, through the use of the target option. Configuration is stored against each target meaning it is possible to start an instance of Hoverfly locally and remotely and still be able to interact with each separately.

See also:

Please refer to *hoverctl commands* for more information about hoverctl.

See also:

Please refer to the *Controlling a remote Hoverfly instance with hoverctl* tutorial for a step by step example of using targets.

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:

Note: By default, request headers are not captured. If you want to capture headers, you will need to specify them when setting capture mode.

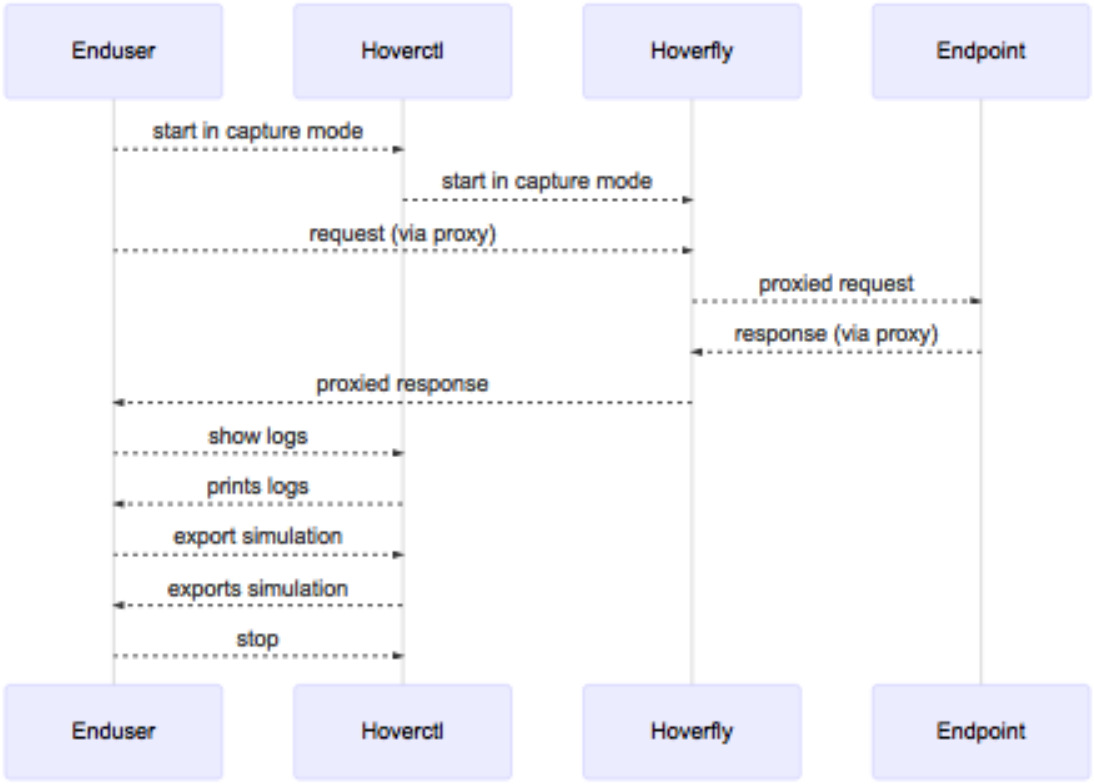
```
hoverctl mode capture --headers "User-Agent,Content-Type,Authorization"
hoverctl mode capture --all-headers
```

Note: It is possible to filter and export simulation to separate JSON files by providing a plain text or regex string to the `--url-pattern` flag:

```
hoverctl export echo.json --url-pattern "echo.jsontest.com"     // export simulations␣
↪for echo.jsontest.com only
hoverctl export api.json --url-pattern "(.+).jsontest.com"      // export simulations␣
↪for all jsontest.com subdomains
```

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:

```
1       "globalActions": {
2          "delays": []
3       }
```

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

```
1       "globalActions": {
2          "delays": [
3             {
4                "urlPattern": ".",
5                "httpMethod": "",
6                "delay": 2000
7             }
8          ]
9       }
```

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",
               "httpMethod": "",
               "delay": 1000
            },
            {
               "urlPattern": "date\\.jsontest\\.com",
               "httpMethod": "",
               "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",
               "httpMethod": "",
               "delay": 2000
            },
            {
               "urlPattern": "echo\\.jsontest\\.com\\/b\\/c",
               "httpMethod": "",
               "delay": 2000
            },
            {
               "urlPattern": "echo\\.jsontest\\.com\\/c\\/d",
               "httpMethod": "",
               "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",
               "httpMethod": "GET",
               "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://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.

Loose request matching using a Request Matcher

See also:

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

In some cases you may want Hoverfly to return the same stored response for more than one incoming request. This can be done using Request Matchers.

Let's begin by capturing some traffic and exporting a simulation. This step saves us having to manually create a simulation ourselves and gives us a request to work with.

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

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

```
"path": [
        {
           "matcher": "exact",
           "value": "/foo/baz/bar/spam"
        }
 ]
```

Modify them to:

```
"path": [
        {
           "matcher": "glob",
           "value": "/foo/*/bar/spam"
        }
 ]
```

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.

See also:

In this example we used the `globMatch` Request Matcher type. For a list of other Request Matcher types and examples of how to use them, please see the *Request matchers* section.

Note: Key points:

- To change how incoming requests are matched to stored responses, capture a simulation, export it, edit it
- While editing, choose a request field to match on, select a Request Matcher type and a matcher value
- Re-import the simulation
- Requests can be manually added without capturing the request

---

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.

Capturing a sequence of responses

You may want capture the same request multiple times. You may to do this because the response served changes. An example of this could include a response that returns the time.

To record a sequence of duplicate requests, we need to enable stateful recording in capture mode.

```
hoverctl mode capture --stateful
```

Now that we have enabled stateful recording, we can make several capture several duplicate requests.

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

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

Once we have finished capturing requests, we can switch Hoverfly back to simulate mode.

```
hoverctl mode simulate
```

Now we are in simulate, we can make the same requests again, and we will see the time update each request we make until we reach the end of our recorded sequence.

```
{
    "time": "01:59:21 PM",
    "milliseconds_since_epoch": 1528120761743,
    "date": "06-04-2018"
}
{
    "time": "01:59:23 PM",
    "milliseconds_since_epoch": 1528120763647,
    "date": "06-04-2018"
}
```

If we look at the simulation captured, can see that the requests have the `requiresState` fields set. a sequence counter.

```
"requiresState": {
    "sequence:1": "1"
}

"requiresState": {
    "sequence:1": "2"
}
```

We can also see that the first respone has *transitionsState*' field set.

```
"transitionsState": {
    "sequence:1": "2"
}
```

See also:

For a more detailed explaination of how sequences work in hoverfly: see *Sequences* in the *Key Concepts* section.

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. With hoverctl, you can do this using the **targets** feature.

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**, you can create a **target** named `remote` using hoverctl. This target will be configured to communicate with Hoverfly.

```
hoverctl targets create remote \
    --host hoverfly.example.com \
    --admin-port 8880 \
    --proxy-port 8555
```

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 this instance:

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

You will now need to specify the `remote` target every time you want to interact with this Hoverfly instance. If you are only working with this remote instance, you can set it to be the default target instance for hoverctl.

```
hoverctl targets default remote
```

Note: The `--host` value of the hoverctl target 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

Sometimes, you may need authentication on Hoverfly. An example use case would be when running Hoverfly on a remote host.

Hoverfly can provide authentication for both the admin API (using a JWT token) and the proxy (using HTTP basic auth).

Setting Hoverfly authentication credentials

To start a Hoverfly instance with authentication enabled, you need to run the `hoverctl start` command with the authentication (`--auth`) flag.

```
hoverctl start --auth
```

Running this command will prompt you to set a username and password for the Hoverfly instance.

This can be bypassed by providing the `--username` and `--password` flags, although **this will leave credentials in your terminal history**.

> **Warning:** By default, hoverctl will start Hoverfly with authentication disabled. If you require authentication you must make sure the `--auth` flag are supplied every time Hoverfly is started.

Logging in to a Hoverfly instance with hoverctl

Now that a Hoverfly instance has started with authentication enabled, you will need to **login** to the instance using hoverctl.

```
hoverctl login
```

Running this command will prompt you to enter the username and password you set for the Hoverfly instance. Again, this can be bypassed by providing the `--username` and `--password` flags.

There may be situations in which you need to log into to a Hoverfly instance that is already running. In this case, it is best practice to create a new **target** for the instance (please see *Controlling a remote Hoverfly instance with hoverctl* for more information on **targets**). You can do this using the `--new-target` flag.

In this example, a remote Hoverfly instance is already running on the host `hoverfly.example.com`, with the ports set to 8880 and 8555 and authentication enabled (the example from *Controlling a remote Hoverfly instance with hoverctl*). You will need to create a new target (named `remote`) for the instance and log in with it.

```
hoverctl login --new-target remote \
    --host hoverfly.example.io \
    --admin-port 8880 \
    --proxy-port 8550
```

You will be prompted to enter the username and password for the instance.

Now run the following commands to capture and simulate a URL using the remote Hoverfly:

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

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.

Troubleshooting

Why isn't Hoverfly matching my request?

When Hoverfly cannot match a response to an incoming request, it will return information on the closest match:

```
Hoverfly Error!

There was an error when matching

Got error: Could not find a match for request, create or record a valid matcher first!
```

```
The following request was made, but was not matched by Hoverfly:

{
    "Path": "/closest-miss",
    "Method": "GET",
    "Destination": "destination.com",
    "Scheme": "http",
    "Query": "",
    "Body": "",
    "Headers": {
        "Accept-Encoding": [
            "gzip"
        ],
        "User-Agent": [
            "Go-http-client/1.1"
        ]
    }
}

The matcher which came closest was:

{
    "path": {
        "exactMatch": "/closest-miss"
    },
    "destination": {
        "exactMatch": "destination.com"
    },
    "body": {
        "exactMatch": "body"
    }
}

But it did not match on the following fields:

[body]

Which if hit would have given the following response:

{
    "status": 200,
    "body": "",
    "encodedBody": false,
    "headers": null
}`
```

Here, you can see which fields did not match. In this case, it was the `body`. You can also view this information by running `hoverctl logs`.

Why isn't Hoverfly returning the closest match when it cannot match a request?

Hoverfly will only provide this information when the matching strategy is set to **strongest match** (the default). If you are using the **first match** matching strategy, the closet match information will not be returned.

How can I view the Hoverfly logs?

```
hoverctl logs
```

Why does my simulation have a `deprecatedQuery` field?

Older simulations that have been upgraded through newer versions of Hoverfly may now contain a field on requests called `deprecatedQuery`. With the v5 simulation schema, the request query field was updated to more fully represent request query paramters. This involves storing queries based on query keys, similarly to how headers are stored in a simulation.

Currently the `deprecatedQuery` field will work and works alongside the `query` field and support for this field will eventually be dropped.

If you have `deprecatedQuery` field, you should remove it by splitting it by query keys.

```
"deprecatedQuery": "page=20&pageSize=15"
```

```
"query": {
    "page": [
        {
            "matcher": "exact",
            "value": "20"
        }
    ],
    "page": [
        {
            "matcher": "exact",
            "value": "15"
        }
    ],
}
```

If you cannot update your `deprecatedQuery` from your simulation for a technical reason, feel free to raise an issue on Hoverfly.

Why am I not able to access my Hoverfly remotely?

That's because Hoverfly is bind to loopback interface by default, meaning that you can only access to it on localhost. To access it remotely, you can specify the IP address it listens on. For example, setting `0.0.0.0` to listen on all network interfaces.

```
hoverfly -listen-on-host 0.0.0.0
```

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:
  completion  Create Bash completion file for hoverctl
  config      Show hoverctl configuration information
  delete      Delete Hoverfly simulation
  destination Get and set Hoverfly destination
  diff        Manage the diffs for Hoverfly
  export      Export a simulation from Hoverfly
  flush       Flush the internal cache in Hoverfly
  import      Import a simulation into Hoverfly
  login       Login to Hoverfly
  logs        Get the logs from Hoverfly
  middleware  Get and set Hoverfly middleware
  mode        Get and set the Hoverfly mode
  start       Start Hoverfly
  state       Manage the state for Hoverfly
  status      Get the current status of Hoverfly
  stop        Stop Hoverfly
  targets     Get the current targets registered with hoverctl
  version     Get the version of hoverctl

Flags:
  -f, --force           Bypass any confirmation when using hoverctl
  -h, --help            help for hoverctl
      --set-default     Sets the current target as the default target for hoverctl
  -t, --target string   A name for an instance of Hoverfly you are trying to␣
↪communicate with. Overrides the default target (default)
  -v, --verbose         Verbose logging from hoverctl

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

hoverctl auto completion

hoverctl supplies auto completion for Bash. Run the following command to install the completions.

```
hoverctl completion
```

This will create the completion file in your hoverfly directory and create a symbolic link in your bash_completion.d folder.

Optionally you can supply a location for the symbolic link as an argument to the completion command.

```
hoverctl completion /usr/local/etc/bash_completion.d/hoverctl
```

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 "memory")
    -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
        Enable CORS headers to allow frontend development
    -disable-cache
        Disable the cache that sits infront of matching
    -generate-ca-cert
        generate CA certificate and private key for MITM
    -https-only
        allow only secure secure requests to be proxied by hoverfly
    -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'
    -journal-size int
        Set the size of request/response journal (default "1000") (default 1000)
    -key string
        private key of the CA used to sign MITM certificates
    -logs string
        Specify format for logs, options are "plaintext" and "json" (default
↪"plaintext") (default "plaintext")
```

```
    -logs-size int
        Set the amount of logs to be stored in memory (default "1000") (default 1000)
    -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
    -password-hash string
        password hash for new user instead of password
    -plain-http-tunneling
        use plain http tunneling to host with non-443 port - defaults to false
    -pp string
        proxy port - run proxy on another port (i.e. '-pp 9999' to run proxy on port␣
↪9999)
    -proxy-auth Proxy-Authorization
        Switch the Proxy-Authorization header from proxy-auth Proxy-Authorization to␣
↪header-auth `X-HOVERFLY-AUTHORIZATION`. Switching to header-auth will auto enable -
↪https-only (default "proxy-auth")
    -synthesize
        start Hoverfly in synthesize mode (middleware is required)
    -tls-verification
        turn on/off tls verification for outgoing requests (will not try to verify␣
↪certificates) - defaults to true (default true)
    -upstream-proxy string
        specify an upstream proxy for hoverfly to route traffic through
    -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)
```

Request matchers

A Request Matcher is used to define the desired value for a specific request field when matching against incoming requests. Given a **matcher value** and **string to match**, each matcher will transform and compare the values in a different way.

Exact matcher

Evaluates the equality of the matcher value and the string to match. There are no transformations. This is the default Request Matcher type which is set by Hoverfly when requests and responses are captured.

Example

```
"matcher": "exact"
"value": "?"
```

Glob matcher

Allows wildcard matching (similar to BASH) using the * character.

Example

```
"matcher": "glob"
"value": "?"
```

Regex matcher

Parses the matcher value as a regular expression which is then executed against the string to match. This will pass only if the regular expression successfully returns a result.

Example

```
"matcher": "regex"
"value": "?"
```

XML matcher

Transforms both the matcher value and string to match into XML objects and then evaluates their equality.

Example

```
"matcher": "xml"
"value": "?"
```

XPath matcher

Parses the matcher value as an XPath expression, transforms the string to match into an XML object and then executes the expression against it. This will pass only if the expression successfully returns a result.

Example

```
"matcher": "xpath"
"value": "?"
```

JSON matcher

Transforms both the matcher value and string to match into JSON objects and then evaluates their equality.

Example

```
"matcher": "json"
"value": "?"
```

JSONPath matcher

Parses the matcher value as a JSONPath expression, transforms the string to match into a JSON object and then executes the expression against it. This will pass only if the expression successfully returns a result.

Example

```
"matcher": "jsonpath"
"value": "?"
```

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": {
          "path": {
            "exactMatch": "/"
          },
          "method": {
            "exactMatch": "GET"
          },
          "destination": {
            "exactMatch": "myhost.io"
          },
          "scheme": {
            "exactMatch": "https"
          },
          "query": {
            "exactMatch": ""
          },
          "body": {
            "exactMatch": ""
          },
          "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"
            ]
          },
```

```
          "templated": false
        },
        "request": {
          "path": {
            "exactMatch": "/template"
          }
        }
      }
    ],
    "globalActions": {
      "delays": []
    }
  },
  "meta": {
    "schemaVersion": "v3",
    "hoverflyVersion": "v0.11.0",
    "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": {
          "path": {
            "exactMatch": "/"
          },
          "method": {
            "exactMatch": "GET"
          },
          "destination": {
            "exactMatch": "myhost.io"
          },
          "scheme": {
            "exactMatch": "https"
          },
          "query": {
            "exactMatch": ""
          },
          "body": {
```

```
            "exactMatch": ""
          },
          "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": {
          "path": {
            "exactMatch": "/template"
          }
        }
      }
    ],
    "globalActions": {
      "delays": []
    }
  },
  "meta": {
    "schemaVersion": "v3",
    "hoverflyVersion": "v0.11.0",
    "timeExported": "2016-11-11T11:53:52Z"
  }
}
```


GET /api/v2/simulation/schema

Gets the JSON Schema used to validate the simulation JSON.

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

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

GET /api/v2/cache

Gets the requests and responses stored in the cache.

Example response body

```
{
    "cache": [
        {
            "key": "2fc8afceec1b6bcf99ff1f547c1f5b11",
            "matchingPair": {
                "request": {
                    "path": {
                        "exactMatch": "hoverfly.io"
                    }
                },
                "response": {
                    "status": 200,
                    "body": "response body",
                    "encodedBody": false,
                    "headers": {
                        "Hoverfly": [
                            "Was-Here"
                        ]
                    }
                }
            },
            "headerMatch": false
        }
    ]
}
```

DELETE /api/v2/cache

Delete all requests and responses stored in the cache.

GET /api/v2/logs

Gets the logs from Hoverfly.

Example response body

```
{
    "logs": [
        {
            "level": "info",
            "msg": "serving proxy",
            "time": "2017-03-13T12:22:39Z"
        },
        {
            "destination": ".",
            "level": "info",
            "mode": "simulate",
            "msg": "current proxy configuration",
            "port": "8500",
            "time": "2017-03-13T12:22:39Z"
        },
        {
            "destination": ".",
            "Mode": "simulate",
            "ProxyPort": "8500",
            "level": "info",
            "msg": "Proxy prepared...",
            "time": "2017-03-13T12:22:39Z"
        },
    ]
}
```

GET /api/v2/journal

Gets the journal from Hoverfly. Each journal entry contains both the request Hoverfly recieved and the response it served along with the mode Hoverfly was in, the time the request was recieved and the time taken for Hoverfly to process the request. Latency is in milliseconds.

Example response body

```
{
  "journal": [
    {
      "request": {
        "path": "/",
        "method": "GET",
        "destination": "hoverfly.io",
        "scheme": "http",
        "query": "",
```

```
        "body": "",
        "headers": {
          "Accept": [
            "*/*"
          ],
          "Proxy-Connection": [
            "Keep-Alive"
          ],
          "User-Agent": [
            "curl/7.50.2"
          ]
        }
      },
      "response": {
        "status": 502,
        "body": "Hoverfly Error!\n\nThere was an error when matching\n\nGot error:␣
↪Could not find a match for request, create or record a valid matcher first!",
        "encodedBody": false,
        "headers": {
          "Content-Type": [
            "text/plain"
          ]
        }
      },
      "mode": "simulate",
      "timeStarted": "2017-07-17T10:41:59.168+01:00",
      "latency": 0.61334
    }
  ]
}
```

DELETE /api/v2/journal

Delete all entries stored in the journal.

POST /api/v2/journal

Filter and search entries stored in the journal.

Example request body

```
{
    "request": {
        "destination": {
          "exactMatch": "hoverfly.io"
        }
    }
}
```

GET /api/v2/state

Gets the state from Hoverfly. State is represented as a set of key value pairs.

Example response body

```
{
  "state": {
    "page_state": "CHECKOUT"
  }
}
```

DELETE /api/v2/state

Deletes all state from Hoverfly.

PUT /api/v2/state

Deletes all state from Hoverfly and then sets the state to match the state in the request body.

Example request body

```
{
  "state": {
    "page_state": "CHECKOUT"
  }
}
```

PATCH /api/v2/state

Updates state in Hoverfly. Will update each state key referenced in the request body.

Example request body

```
{
  "state": {
    "page_state": "CHECKOUT"
  }
}
```

GET /api/v2/diff

Gets all reports containing response differences from Hoverfly. The diffs are represented as lists of strings grouped by the same requests.

Example response body

```
{
  "diff": [{
    "request": {
      "method": "GET",
      "host": "time.jsontest.com",
      "path": "/",
      "query": ""
    },
    "diffReports": [{
      "timestamp": "2018-03-16T17:45:40Z",
      "diffEntries": [{
        "field": "header/X-Cloud-Trace-Context",
        "expected": "[ec6c455330b682c3038ba365ade6652a]",
        "actual": "[043c9bb2eafa1974bc09af654ef15dc3]"
      }, {
        "field": "header/Date",
        "expected": "[Fri, 16 Mar 2018 17:45:34 GMT]",
        "actual": "[Fri, 16 Mar 2018 17:45:41 GMT]"
      }, {
        "field": "body/time",
        "expected": "05:45:34 PM",
        "actual": "05:45:41 PM"
      }, {
        "field": "body/milliseconds_since_epoch",
        "expected": "1.521222334104e+12",
        "actual": "1.521222341017e+12"
      }]
    }]
  }]
}
```


DELETE /api/v2/diff

Deletes all reports containing differences from Hoverfly.

Simulation schema

This is the JSON schema for v2 Hoverfly simulations.


```
{
  "additionalProperties": false,
  "definitions": {
    "delay": {
      "properties": {
        "delay": {
          "type": "integer"
        },
        "httpMethod": {
          "type": "string"
        },
        "urlPattern": {
          "type": "string"
        }
```

```
      },
      "type": "object"
    },
    "field-matchers": {
      "properties": {
        "exactMatch": {
          "type": "string"
        },
        "globMatch": {
          "type": "string"
        },
        "jsonMatch": {
          "type": "string"
        },
        "regexMatch": {
          "type": "string"
        },
        "xpathMatch": {
          "type": "string"
        }
      },
      "type": "object"
    },
    "headers": {
      "additionalProperties": {
        "items": {
          "type": "string"
        },
        "type": "array"
      },
      "type": "object"
    },
    "meta": {
      "properties": {
        "hoverflyVersion": {
          "type": "string"
        },
        "schemaVersion": {
          "type": "string"
        },
        "timeExported": {
          "type": "string"
        }
      },
      "required": [
        "schemaVersion"
      ],
      "type": "object"
    },
    "request": {
      "properties": {
        "body": {
          "$ref": "#/definitions/field-matchers"
        },
        "destination": {
          "$ref": "#/definitions/field-matchers"
        },
        "headers": {
```

```
          "$ref": "#/definitions/headers"
        },
        "path": {
          "$ref": "#/definitions/field-matchers"
        },
        "query": {
          "$ref": "#/definitions/field-matchers"
        },
        "scheme": {
          "$ref": "#/definitions/field-matchers"
        }
      },
      "type": "object"
    },
    "request-response-pair": {
      "properties": {
        "request": {
          "$ref": "#/definitions/request"
        },
        "response": {
          "$ref": "#/definitions/response"
        }
      },
      "required": [
        "request",
        "response"
      ],
      "type": "object"
    },
    "response": {
      "properties": {
        "body": {
          "type": "string"
        },
        "encodedBody": {
          "type": "boolean"
        },
        "headers": {
          "$ref": "#/definitions/headers"
        },
        "status": {
          "type": "integer"
        },
        "templated": {
          "type": "boolean"
        }
      },
      "type": "object"
    }
  },
  "description": "Hoverfly simulation schema",
  "properties": {
    "data": {
      "properties": {
        "globalActions": {
          "properties": {
            "delays": {
              "items": {
```

```
                "$ref": "#/definitions/delay"
              },
              "type": "array"
            }
          },
          "type": "object"
        },
        "pairs": {
          "items": {
            "$ref": "#/definitions/request-response-pair"
          },
          "type": "array"
        }
      },
      "type": "object"
    },
    "meta": {
      "$ref": "#/definitions/meta"
    }
  },
  "required": [
    "data",
    "meta"
  ],
  "type": "object"
}
```

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