Effective Testing with API Simulation and (Micro)Service Virtualisation

Module Five: Stateful Simulations

Purpose of this Lab

- Understanding and orchestration of Hoverflies state-store
- Using state in a templated response
- Building matchers which depend on state
- Building responses which mutate state

Questions

Feel free to ask questions and ask for help at any point during the workshop. Also, please collaborate with others.

Pre-requisites

Make sure we are in the correct directory!

```
$ cd api-simulation-training/5-stateful
```

Exercise 1: Orchestration and Templating of State

During this exercise, we will learn about Hoverflies state store, and how we can use it to template responses.

Steps

1. Make sure Hoverfly is running:

```
$ hoverctl start
```

```
target Hoverfly is already running
```

2. Make sure the shopping API is running:

> In this exercise, we will be simulating a shopping API instead of the flights service

```
$ ./run-shopping-service.sh
service started
```

3. Make sure all the existing data is deleted from Hoverfly:

```
$ hoverctl delete
Are you sure you want to delete the current simulation? [y/n]: y
Simulation data has been deleted from Hoverfly
```

4. Now, using the API, put some bacon into the shopping basket. We want the basket to be in a state where if we retrieve it, it contains bacon. Here are some example requests to get you started:

```
1. Retrieving from the basket

Example Request:

$ curl localhost:8081/api/v1/shopping-basket

Example Response:

HTTP/1.1 200 OK
Content-Type: application/json;charset=UTF-8

[{"item":"some-item"}]

2. Adding to the Basket:

Example Request:
```

```
$ curl -H 'Content-Type: application/json' -X PUT -d '{"item":"name of
item"}' localhost:8081/api/v1/shopping-basket

Example Response:

HTTP/1.1 200 OK
Content-Type: application/json;charset=UTF-8
```

5. Once are basket returns bacon, create a simulation of the request.

We are assuming you know how to make a simulation based on your work in the previous modules. If you are unsure then consult the previous exercises, or ask for help.

6. The problem we have now is that if we wanted our simulation to also simulate an empty basket, then the matcher would be identical to the one for our full basket. This is because the HTTP request for both scenarios is the same:

```
Example Request:

$ curl localhost:8081/api/v1/shopping-basket
```

This creates a requirement for an internal representation of state. Hoverfly can use that state information to determine whether it should return an empty or a full basket.

Now, let's look at Hoverflies state-store. First, let's run help and then look at all the available commands:

```
$ hoverctl state-store --help
```

And then, let's look at what is currently stored in the state store (It should be empty):

```
$ hoverctl state-store get-all
The state for Hoverfly is empty
```

7. Let's see how we can modify the state store. First we can set a state key and value:

```
$ hoverctl state-store set basket bacon
Successfully set state key and value:
```

```
"basket"="bacon"
$ hoverctl state-store get-all
State of Hoverfly:
"basket"="bacon"
```

And second of all we can delete the contents of the state store:

```
$ hoverctl state-store delete-all
State has been deleted
$ hoverctl state-store get-all
The state for Hoverfly is empty
```

8. Now, by using templating and the state store, modify your single matchers response so it either returns a different item based on Hoverfly's state.

State can be accessed as follows:

Example	State Store	Result
{{ State.name }}	{"name": "value"}	value

9. Verify your simulation behaves as expected by using the CLI to see if changing state modifies the response for your shopping basket.

Exercise 2: Matching on State

In this exercise we will perform matching on state. This means we will achieve a similar outcome to the previous exercise, only this time it will be through multiple matchers.

Steps

1. Make sure Hoverfly is running:

```
$ hoverctl start
target Hoverfly is already running
```

2. Make sure the shopping API is running:

```
$ ./run-shopping-service.sh
service started
```

3. Make sure all the existing data is deleted from Hoverfly:

```
$ hoverctl delete
Are you sure you want to delete the current simulation? [y/n]: y
Simulation data has been deleted from Hoverfly
```

4. Just like in the previous example, create a simulation of retrieving items from the shopping basket.

If you are not sure how to do this, consult the previous exercise or ask.

5. Hoverfly can also match on state. Here is an example of how:

```
Example Matcher:

{
    "request": {
        "requiresState": {
            "some-state-key" : "some-state-value",
            "other-state-key" : "other-state-value"
        }
        // Rest of matcher here
    },
    "response": {
        // Response fields here
    }
}
```

Using stateful matching, modify your simulation so it returns either an empty basket or a basket returning bacon based on state.

9. Verify your simulation behaves as expected by using the CLI to see if changing state modifies the response for your shopping basket.

Once this exercise is finished, **export your simulation for use in the next exercise.**

Exercise 3: Modelling a State Machine

Currently we have only modified Hoverfly's state by using the CLI. With a real API, state transitions would be triggered by API requests. In this exercise we will learn how to simulate those types of requests, building a state machine.

Steps

1. Make sure Hoverfly is running:

```
$ hoverctl start
target Hoverfly is already running
```

2. Make sure the shopping API is running:

```
$ ./run-shopping-service.sh
service started
```

3. Make sure all the existing data is deleted from Hoverfly:

```
$ hoverctl delete
Are you sure you want to delete the current simulation? [y/n]: y
Simulation data has been deleted from Hoverfly
```

4. Import the simulation from the previous exercise. If you didn't complete it, load one from the answers directory.

```
$ hoverctl import answers/stateful-exercise-two-simulation.json
Successfully imported simulation from
answers/stateful-exercise-two-simulation.json
```

5. Now, let's add two extra requests to the simulation by putting Hoverfly into capture mode and intercepting them:

```
1. Deleting from the basket
```

```
Example Request:

$ curl -X DELETE localhost:8081/api/v1/shopping-basket

Example Response:

HTTP/1.1 200 OK

2. Adding to the Basket:

Example Request:

$ curl -H 'Content-Type: application/json' -X PUT -d '{"item":"name of
item"}' localhost:8081/api/v1/shopping-basket

Example Response:

HTTP/1.1 200 OK
Content-Type: application/json;charset=UTF-8
```

6. Now our simulation should contain:

1. A request to get an empty basket which matches on state
2. A request to get a full basket which matches on state
3. A request to delete a basket
4. A request to add bacon to the basket

As we can see, adding and deleting should change the state of our simulation. Deleting will make make our basket empty, and adding would populate it.

Hoverfly allows you to transition and delete state on a match. Here is an example:

```
Example Matcher:

{
    "request": {
        // field matchers
```

```
    },
    "response": {
        "removesState" : ["some-state-key"],
        "transitionsState" : {
            "payment-flow" : "complete",
        },
        // rest of response fields
    }
}
```

Now, using the above example for reference build a state machine. What we want is:

1. Deleting from the basket should delete the basket state
2. Adding to the basket should set the state

6. Verify your state machine works by following this flow in the terminal:

1. Delete all state using the CLI
2. Retrieve the basket and assert that it is empty
3. Add bacon to the basket
4. Retrieve from the basket and assert that it contains bacon
5. Delete from the basket
6. Retrieve the basket and assert that it is empty

Advanced Exercise

In this exercise, we are going to model the workflow for publishing a blog post. There is no API so capture, but we have documentation. This means we will create our simulation manually.

The basic workflow will be as follows:

1. User creates a blog post and it enters “draft” state
2. Once the draft is complete, the user can change its state to “review”
3. Once in “review”, the post can be rejected and moved back into “draft” or accepted and moved into “published”
4. At any point the blog post can be completely deleted

Here are the API endpoints:

1. Creating a blog post

Example Request:

```
$ curl -H 'Content-Type: application/json' -X POST -d '{"content":"the blog content"}' http://localhost:8081/api/v1/blogs
```

Example Response:

```
HTTP/1.1 201 CREATED
Location: http://localhost:8081/api/v1/blogs/1
```

2. Retrieving a blog post

Example Request:

```
$ curl http://localhost:8081/api/v1/blogs/1
```

Example Response:

```
HTTP/1.1 200 OK
Content-Type: application/json;charset=UTF-8
{
    "content":"the blog content",
    "Status":"draft"
}
```

3. Deleting a blog post

Example Request:

```
$ curl -X DELETE http://localhost:8081/api/v1/blogs/1
```

Example Response:

```
HTTP/1.1 200 OK
```

```
3. Changing the status of a blog post

Example Request:

$ curl -H 'Content-Type: application/json' -X PUT -d '{"status":"review"}'
'http://localhost:8081/api/v1/blogs/1/status'

Example Response:

HTTP/1.1 200 OK
```

What we would like to end up with:

1. If there is no blog then retrieving it and updating it will result in a 404
2. Creating a blog will create a corresponding entry in the state store
3. When changing the status of the blog, moving from draft directly to published should result in a 400