Sauce Labs API Testing - Integrating Contract Tests in a Continous Integration Pipeline

Workshop for SauceCon 2022

by Julian Vargas, Engineering Manager API Testing

Preparation

In order to properly proceed with the workshop and aim to make the most out of the time, it is recommended to check the following items:

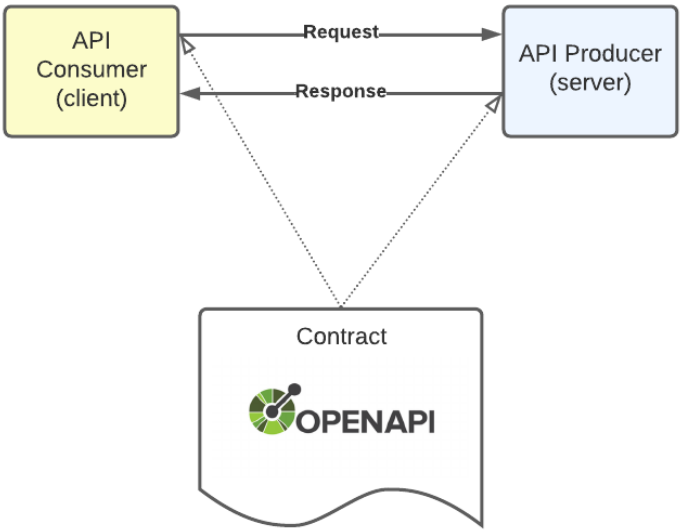
- You have an active Sauce Labs account. This means that you can go to Sauce Labs | Accounts and log in successfully.
- You have set up Sauce Connect Proxy on your computer. This means that you can successfully execute Step 2 in Sauce Labs | Tunnel Proxies as an alternative, you can use **ngrok** instead of SauceConnect
- You have installed Deno on your computer. This means that you can successfully execute

`$ $HOME/.deno/bin/deno run https://deno.land/std/examples/welcome.ts`

- You have Git installed on your computer

Contract testing

Is the process of testing an agreement between a producer and a consumer. The contract is implemented in Open API 3+



The contract is tested from the consumer side (client) using a mocking server that creates valid HTTP responses based on what the contract establishes.

The contract is tested from the producer side (server) by executing the request specified in the contract to the real API server and validating the responses against what the contract expects.

More details can be found in [API Contract Testing | Resources & Community (saucelabs.com)]

Example application

For this workshop we are using an API that helps a user know which is the best sauce to use for a given food so that they can enjoy their food in the best possible manner.

Acceptance criteria

- I will have a command-line application that will receive a word corresponding to any food and will print in the terminal the name of the best sauce for that given food. For example "`$ sauce_for 'fries'`" will print "ketchup"
- This application will be a client terminal application that consumes an HTTP API with one endpoint `` `/sauce` `` with a mandatory query string parameter `` `?_for` `` that will contain the name of the food.
- The API will return a JSON document with only one property `` `sauce_name` `` containing the name of the best sauce for the given food.
- The Server will expose the endpoint `` `http://localhost:8000/sauce/` `` for any GET request received and will use the query string parameter `` `_for=` `` to read the food and respond with the best sauce.

Folder and file structure

```
.
```

```
├── sauce-recipe
    ├── sauce-client
    │   ├── sauce-contract.yml
    │   ├── sauce.test.ts
    │   └── sauce.ts
    ├── sauce-server
    │   ├── sauce-server.ts
    │   ├── sauce.test.ts
    │   ├── sauce.ts
    │   └── tunnel.yml
    └── test-suite.sh
```

Example application

First of all, we are going to set up a continuous integration server, for simplicity we are going to use only our localhost for everything. In order to properly trigger tests after committing code to master, we are going to set up a git hook as follows:

Initialize the root directory as a git repository.

```
$ git init .
Initialized empty Git repository in /home/julian/sauce-recipe/.git/
```

Create a bare repository to push changes to, this will be used as the remote repo and we will use it only to trigger the CI before pushing changes

```
$ git init /tmp/sauce-recipe-origin --bare
Initialized empty Git repository in /tmp/sauce-recipe-origin/
```

Set the remote repository to the one created in the previous step

```
$ git remote add origin /tmp/sauce-recipe-origin
```

This means that whenever we do `$ git push origin master` the changes will be pushed to `/tmp/sauce-recipe-origin`

Rename the default hook `pre-push.sample` to `pre-push`.

```
$ mv .git/hooks/pre-push{.sample,}
```

and include the following content.

```
1 #!/bin/bash --login
2 echo "@@@@ This is better than Jenkins :) @@@@"
3 echo "Running tests..."
4 source test-suite.sh
```

add the following content to `test-suite.sh`

```
echo "tests are passing"
```

add create a commit adding `test-suite.sh` and push to master

```
$ git add test-suite.sh
$ git commit -m "Initial commit setting up CI"
[master (root-commit) 460c8d3] initial commit
 1 file changed, 2 insertions(+)
 create mode 100644 test-suite.sh
```

```
$ git push origin master
@@@@ This is better than Jenkins :) @@@@
running tests
Enumerating objects: 3, done.
Counting objects: 100% (3/3), done.
Writing objects: 100% (3/3), 246 bytes | 246.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To /tmp/sauce-recipe-origin
 * [new branch]      master -> master
```

Notice the first two lines in the output, the first one comes from the hook `.git/hooks/pre-push`, the second one comes from `test-suite.sh`

Just for testing purposes, create an empty commit and push again.

```
$ git commit -m "triggering CI" -allow-empty
[master 923c024] triggering CI

$ git push origin master
@@@@ This is better than Jenkins :) @@@@
running tests
Enumerating objects: 1, done.
Counting objects: 100% (1/1), done.
Writing objects: 100% (1/1), 195 bytes | 195.00 KiB/s, done.
Total 1 (delta 0), reused 0 (delta 0)
To /tmp/sauce-recipe-origin
   460c8d3..923c024  master -> master
```

Starting the implementation from the consumer side

Let us begin by creating a project for the frontend of the application, this will be a Deno app.

```
$ mkdir sauce-client
$ cd sauce-client
```

Add the following content to a new file `sauce.test.ts`

```
  1 import {sauceFor} from './sauce.ts'
  2 import {assertEquals} from
"https://deno.land/x/std@0.65.0/testing/asserts.ts";
  3
  4 Deno.test("it returns the sauce for the given food", async () => {
  5   assertEquals(await sauceFor("fries"), "ketchup");
  6 });
```

Run the tests:

```
$ $HOME/.deno/bin/deno test --allow-net
error: Module not found
"file:///home/julian/sauce-recipe/sauce-client/sauce.ts".
    at file:///home/julian/sauce-recipe/sauce-client/sauce.test.ts:1:24
```

This is good, in a normal Test-Driven Development (TDD) cycle the first step is to create failing tests and let them drive us until the code is green and clean.

Add the following code to a new file called `sauce.ts`

```
  1 export async function sauceFor(food: string): Promise<string> {
  2   const jsonResponse = await
fetch(`http://localhost:5000/sauce?_for=${food}`);
  3   const jsonData = await jsonResponse.json();
  4   return jsonData.sauce_name;
  5 }
```

And run the tests again.

```
$ $HOME/.deno/bin/deno test --allow-net
Check file:///home/julian/sauce-recipe/sauce-client/sauce.test.ts
running 1 test from
file:///home/julian/sauce-recipe/sauce-client/sauce.test.ts
test it returns the sauce for the given food ... FAILED (9ms)
failures:

it returns the sauce for the given food
TypeError: error sending request for url
(http://localhost:5000/sauce?_for=fries): error trying to connect: tcp
connect error: Connection refused (os error 111)
    at async Object.runTests (deno:runtime/js/40_testing.js:986:22)

failures:

        it returns the sauce for the given food

test result: FAILED. 0 passed; 1 failed; 0 ignored; 0 measured; 0
```

```
filtered out (23ms)

error: Test failed
```

It is still failing because there is no API server to produce the json content we are expecting.

Add the following content to the file that will work as our contract `sauce-contract.yml`

```yaml
openapi: "3.0.0"
info:
  version: 1.0.0
servers:
  - url: http://localhost:8000
paths:
  /sauce:
    get:
      parameters:
        - in: query
          name: _for
          schema:
            type: string
          required: true
          example: fries
      responses:
        '200':
          description: returns the best sauce for the given food.
          content:
            application/json:
              schema:
                type: object
                properties:
                  sauce_name:
                    required: true
                    type: string
              example:
                sauce_name: ketchup
```

Run `piestry` to mock the endpoints described in the contract.

```
$ docker run -v "$(pwd)/:/specs" -p 5000:5000 quay.io/saucelabs/piestry
-u /specs/sauce-contract.yml --validate-request
2022-04-25T09:57:28.212Z info: Piestry booting on port: 5000
2022-04-25T09:57:28.231Z info: Registering GET /sauce
```

Test the mocking server by performing a request

```
$ curl "http://localhost:5000/sauce?_for=fries"
```

```
{"sauce_name":"ketchup"}
```

now run the tests again.

```
$ $HOME/.deno/bin/deno test --allow-net
Check file:///home/julian/sauce-recipe/sauce-client/sauce.test.ts
running 1 test from
file:///home/julian/sauce-recipe/sauce-client/sauce.test.ts
test it returns the sauce for the given food ... ok (22ms)

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered
out (44ms)
```

Now that tests are passing in our development environment, it is time to include this test in our CI pipeline.

Go to the root folder, open `test-suite.sh` and add the following content.

```
  1 # Testing the client
  2 cd sauce-client
  3
  4 docker run -d -v "$(pwd)/:/specs" -p 5000:5000
quay.io/saucelabs/piestry -u /specs/sauce-contract.yml
--validate-request
  5 sleep 5
  6
  7 /home/julian/.deno/bin/deno test --allow-net
  8
  9 #cleanup
 10 sudo docker stop $(sudo docker ps -q)
 11 cd ..
```

This will execute piestry to mock the server just as we did manually in our previous steps, run the tests and once they are done, stop and delete all containers.

Commit all changes.

```
$ git add test-suite.sh sauce-client/
$ git commit -m "Adds client implementation"
[master 434e634] Adds client implementation
 4 files changed, 48 insertions(+), 2 deletions(-)
 create mode 100644 sauce-client/sauce-contract.yml
 create mode 100644 sauce-client/sauce.test.ts
 create mode 100644 sauce-client/sauce.ts
```

```
$ git push origin master
@@@@ This is better than Jenkins :) @@@@
running 1 test from
file:///home/julian/sauce-recipe/sauce-client/sauce.test.ts
test it returns the sauce for the given food ... ok (32ms)

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered
out (51ms)

Enumerating objects: 9, done.
Counting objects: 100% (9/9), done.
Delta compression using up to 4 threads
Compressing objects: 100% (7/7), done.
Writing objects: 100% (7/7), 1.18 KiB | 1.18 MiB/s, done.
Total 7 (delta 0), reused 0 (delta 0)
To /tmp/sauce-recipe-origin
   923c024..9b6d064  master -> master
```

What have we achieved so far?

1. We have implemented a client application emulating the server side using Piestry
2. We have implemented a contract from the consumer side point of view.
3. We have integrated contract testing from the consumer side in a Continuous Integration pipeline.

Starting the implementation from the producer side

In the root folder, create a directory for the server side implementation.

```
$ mkdir sauce-server
```

Use the cd command to get into the sauce-server directory and add the following content to a new file called sauce.test.ts

```
  1 import {sauceFor} from './sauce.ts';
  2 import {assertEquals} from
"https://deno.land/x/std@0.65.0/testing/asserts.ts";
  3
  4 Deno.test("it returns ketchup for fries", () => {
  5   assertEquals(sauceFor("fries"), "ketchup");
  6 });
```

Run the tests with:

```
$ $HOME/.deno/bin/deno test
error: Module not found
"file:///home/julian/sauce-recipe/sauce-server/sauce.ts".
    at file:///home/julian/sauce-recipe/sauce-server/sauce.test.ts:1:24
```

The test is failing because there is no file sauce.ts, create this file and add the following content.

```
export function sauceFor(food: string): string {
  const toppingsSaucesMap = new Map<string, string>();

  toppingsSaucesMap.set("fries","ketchup")
  toppingsSaucesMap.set("rice","honey-mustard")
  toppingsSaucesMap.set("nuggets","Szechuan")

  return toppingsSaucesMap.get(food) || "ketchup";
}
```

Run the tests again.

```
$HOME/.deno/bin/deno test
Check file:///home/julian/sauce-recipe/sauce-server/sauce.test.ts
running 1 test from
file:///home/julian/sauce-recipe/sauce-server/sauce.test.ts
test it returns ketchup for fries ... ok (5ms)

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered
out (31ms)
```

Now that tests are passing in the backend, it is time to include them in the CI pipeline. Open `test-suite.sh` in the root folder and add the following content to the end of the file.

```
#testing the server
cd sauce-server
/home/julian/.deno/bin/deno test
```

The file should look like this:

```
# Testing the client
cd sauce-client

docker run -d -v "$(pwd)/:/specs" -p 5000:5000
quay.io/saucelabs/piestry -u /specs/sauce-contract.yml
--validate-request
sleep 5

/home/julian/.deno/bin/deno test --allow-net

#cleanup
sudo docker stop $(sudo docker ps -q)
cd ..
```

```
12
13 #testing the server
14 cd sauce-server
15 /home/julian/.deno/bin/deno test
```

Create a commit with new changes.

```
$ git add test-suite.sh sauce-server/
$ git commit -m "Adds server implementation"
[master 784da89] Adds server implementation
 3 files changed, 19 insertions(+)
 create mode 100644 sauce-server/sauce.test.ts
 create mode 100644 sauce-server/sauce.ts
```

Push the changes to trigger the test suite.

```
$ git push origin master
@@@@ This is better than Jenkins :) @@@@
83c4bf83729f066233a9157bb5f59f599deee1a28287e57120d8da920433c21f
running 1 test from
file:///home/julian/sauce-recipe/sauce-client/sauce.test.ts
test it returns the sauce for the given food ... ok (40ms)

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered
out (55ms)

running 1 test from
file:///home/julian/sauce-recipe/sauce-server/sauce.test.ts
test it returns ketchup for fries ... ok (5ms)

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered
out (18ms)

Enumerating objects: 8, done.
```

At the moment we have the business logic fully implemented but we have not exposed that logic through an API yet. To do that, inside `sauce-server` create a new file called `sauce-server.ts` and add the following content:

```
1 import {serve} from "https://deno.land/std@0.119.0/http/server.ts";
2 import {sauceFor} from "./sauce.ts";
3
4 function handler(_req: Request): Response {
5   const u = new URL(_req.url);
6   const food: string = u.searchParams.get("_for") || "fries";
7   const deliciousSauce: string = sauceFor(food);
```

```
  8   return new Response(JSON.stringify({deliciousSauce:
deliciousSauce}));
  9 }
 10 console.log("Listening on http://localhost:8000");
 11 serve(handler);
```

Note. This implementation includes an intentional error that will be fixed in later steps.

Run the server with:

```
$ $HOME/.deno/bin/deno run --allow-net sauce-server.ts --allow-net
Listening on http://localhost:8000
```

Perform a test request:

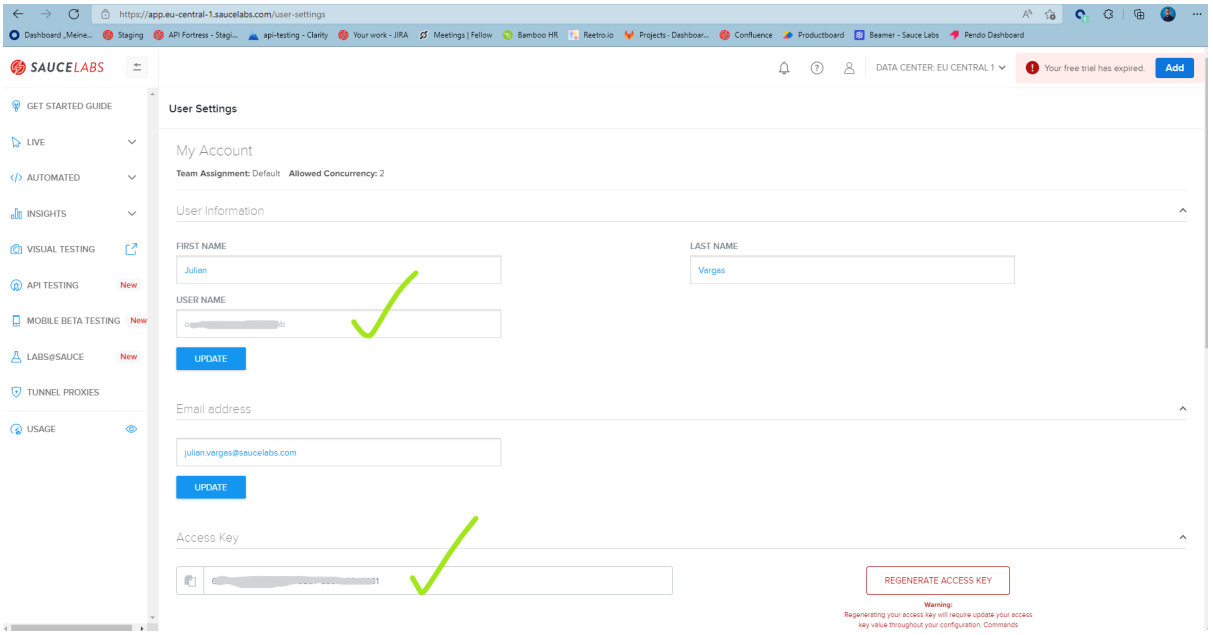
```
$ curl "http://localhost:8000/sauce?_for=rice"

{"deliciousSauce":"honey-mustard"}
```

The server is indeed running and executing our business logic. At this point there is only one step left to fully test the contract.

For the following steps, we need to use our Sauce Labs account and since the server app is running in our local environment, we need a tunnel to allow SauceLab APIT to perform requests. Create a file `tunnel.yml` with the following content

```
  1 rest-url: "https://api.eu-west-3-lnbf.saucelabs.com/rest/v1"
  2 user: "..."
  3 api-key: "..."
  4 no-remove-colliding-tunnels: true
  5 vm-version: "v2alpha"
  6 tunnel-cert: private
  7 tunnel-identifier: contract-testing-prod
```

To retrieve the user and api-key, open your Sauce Labs account and go to `/user-settings`, copy these values.



Then, in your terminal execute:

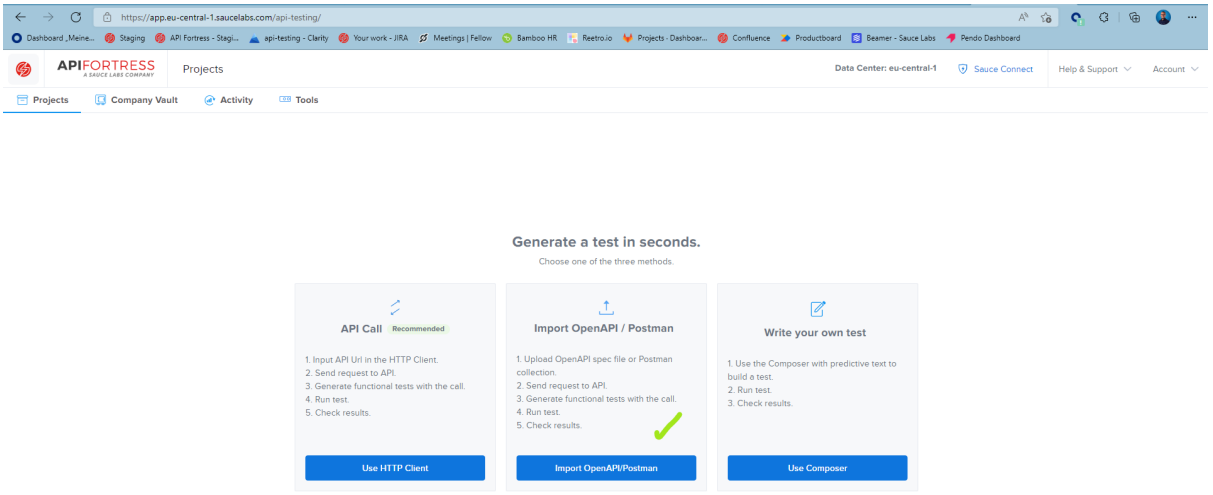
```
$ sc -c tunnel.yml
Sauce Connect ProxyTM opens a secure connection between Sauce Labs and a
locally hosted application.

  Find more information at:
https://docs.saucelabs.com/dev/cli/sauce-connect-proxy

Sauce Connect 4.7.1, build 5439 fb74241

INFO: Adding tunnel to pool 'julian-contract-testing-prod-eu', now
running 1 instance.

Sauce Connect runtime information:
- Name: julian-contract-testing-prod-eu
- PID: 2054
- PID file: /tmp/sc_client-julian-contract-testing-prod-eu1.pid
- Log file: /tmp/sc-julian-contract-testing-prod-eu.log
- SCProxy Port: 35603
- Metric Port: None
- Selenium listener: None
- External proxy for REST API: None
Please wait for 'you may start your tests' to start your tests: \
Secure remote tunnel provisioned. Tunnel ID:
e185688d5fdb418bb070eb8b46f235ec

Sauce Connect is up, you may start your tests.
```

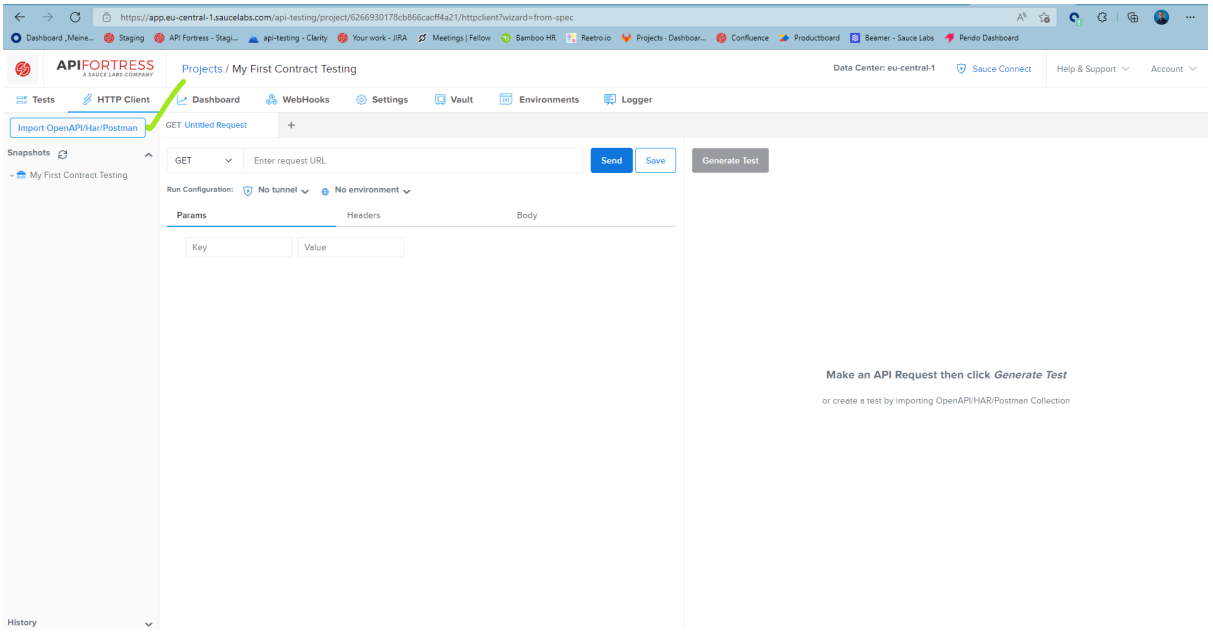
Make note of the tunnel ID, we will need it later.

Now we have to add our local endpoint to the contract, open `sauce-contract.yml` located in `sauce-client` directory and add the server section as shown below:

```
openapi: "3.0.0"
info:
  version: 1.0.0
servers:
  - url: http://localhost:8000
paths:
...
```

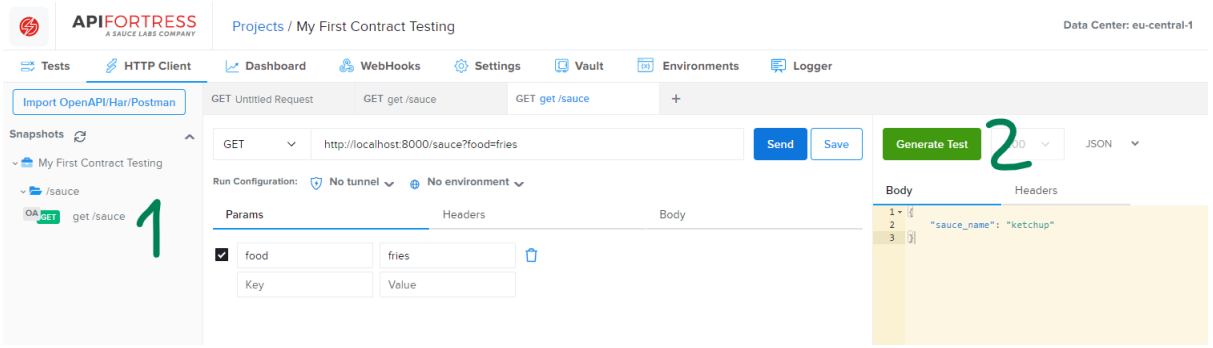
Back into the browser, go to “API Testing” and select the option shown in the picture to import the contract.



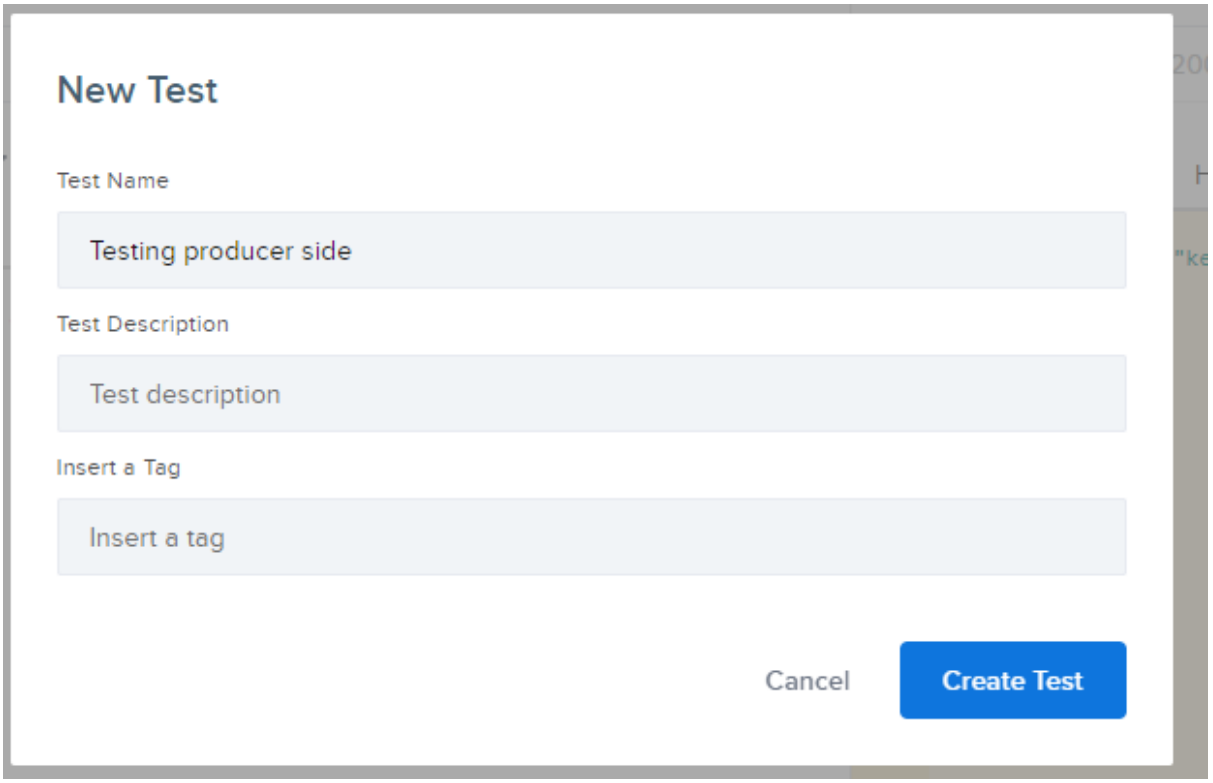


Browse to the folder where the project is located, open the `sauce-client` folder, and import `sauce-contract.yml`

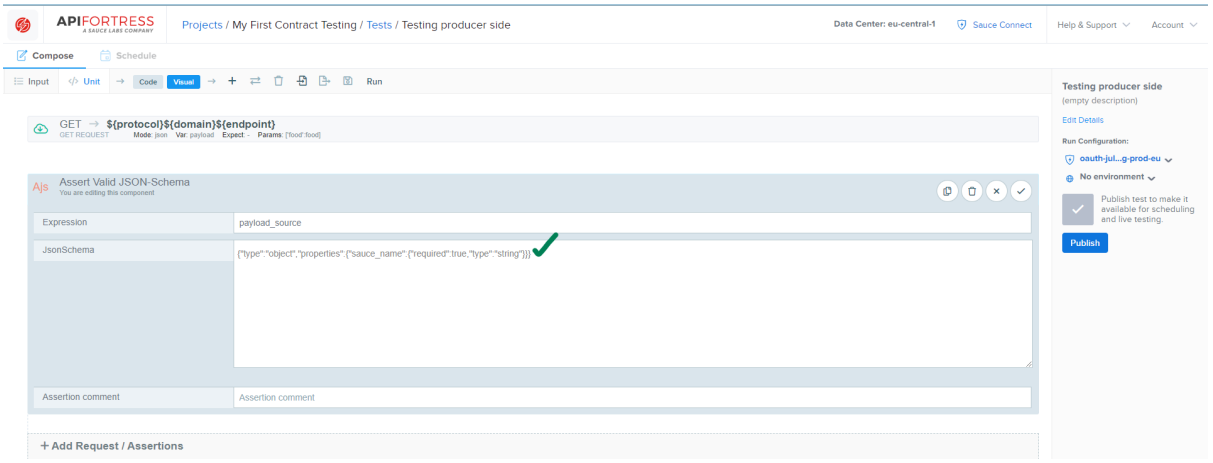
From the available endpoint, select `get/sauce` and then click on `Generate test` as shown in the screenshot below.



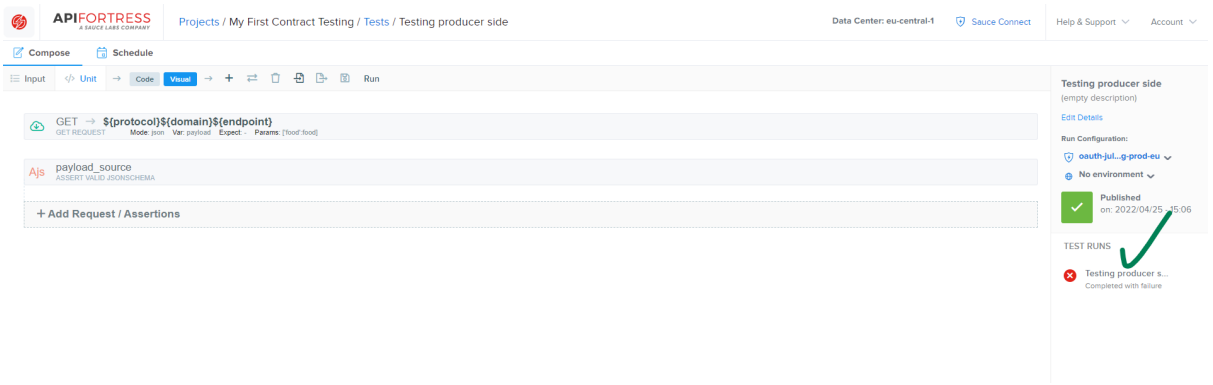
Enter the test name and click “Create Test.”



The test generated will contain one step for validating the response produced by the server against what the contract requires to.

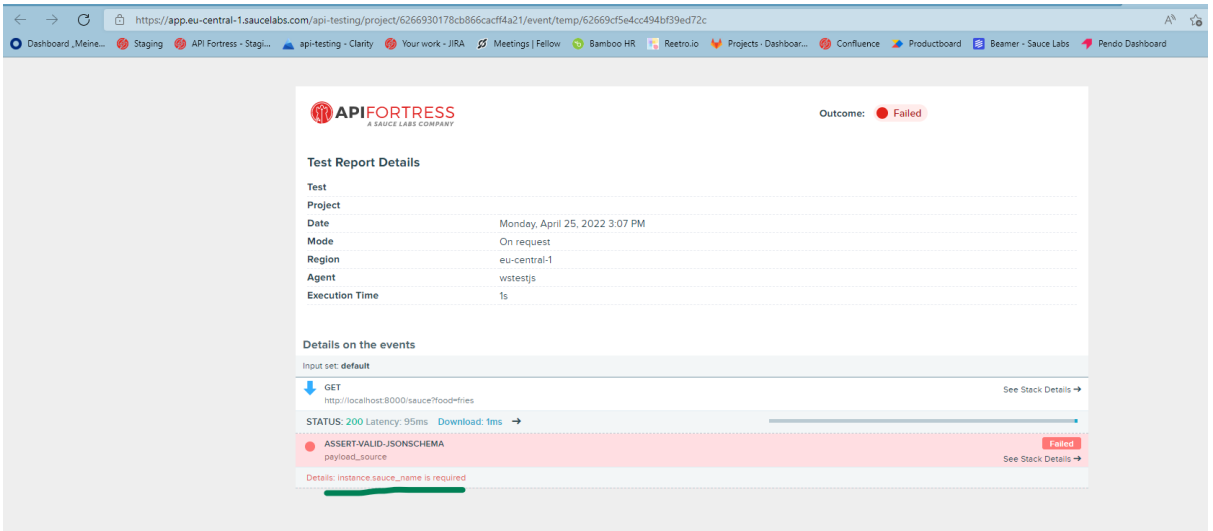


Click on Publish and then on Run. Since we are connected to a tunnel, Sauce Labs will perform a live request to our localhost, the result will be shown in the bottom right corner.



In this case, the test failed, meaning that the response is not exercising the contract.

Click on the failed result.

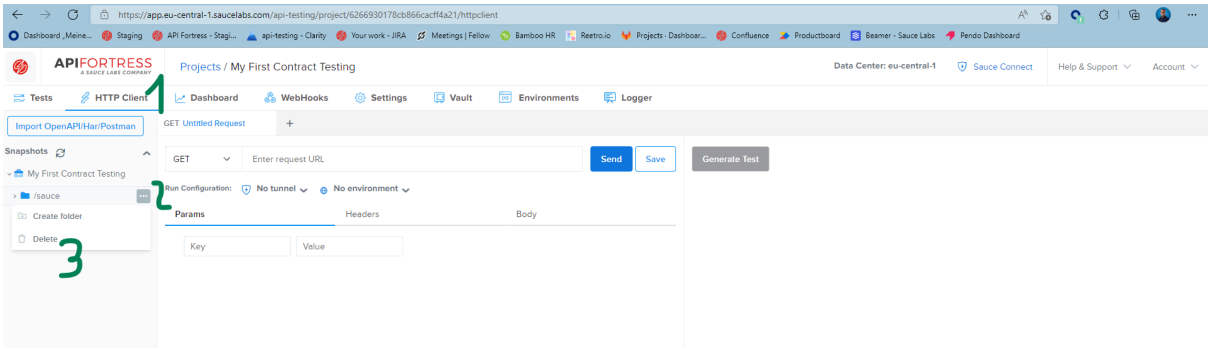


The error says that the contract requires a property `sauce_name` as part of the response but this property was not present. The reason is that in our implementation of `sauce-server.ts` the JSON has a different name.

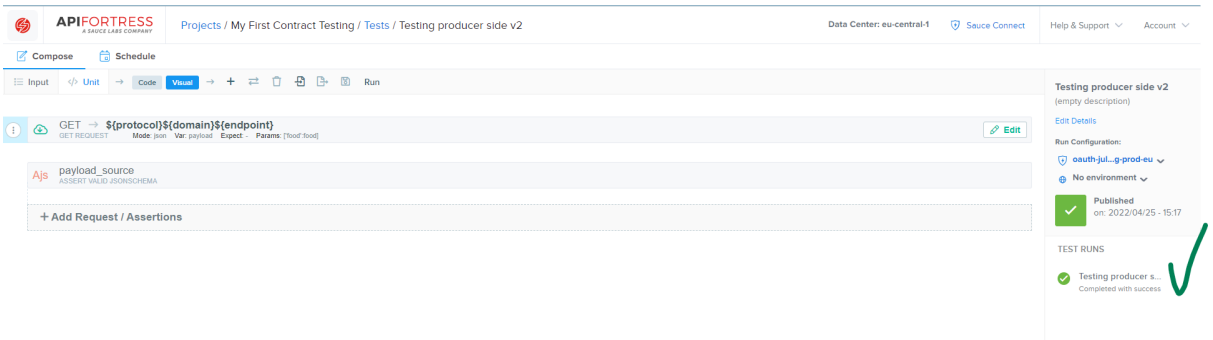
Let's do something very naive and change the contract to use the property name from our implementation in `sauce-server.ts`. Open the contract and in line `23` and `27` change `sauce_name` to `deliciousSauce`.

```
21                type: object
22                properties:
23                  deliciousSauce:
24                    required: true
25                    type: string
26              example:
27                deliciousSauce: ketchup
```

In the browser, go to the HTTP Client and delete the contract.



Import the new contract and repeat the steps to create a test for this new version. Publish and Run the test.



The test now is passing because the response received from our API matches the contract.

Congratulations now let us commit all our changes.

```
$ git add sauce-client/sauce-contract.yml sauce-server/sauce-server.ts
$ git commit -m "Adds contract testing on the producer side"
[master d45fe23] Adds contract testing on the producer side
 2 files changed, 13 insertions(+), 1 deletion(-)
```

```
 create mode 100644 sauce-server/sauce-server.ts

$ git push origin master
@@@@ This is better than Jenkins :) @@@@
running 1 test from
file:///home/julian/sauce-recipe/sauce-client/sauce.test.ts
test it returns the sauce for the given food ... FAILED (28ms)

failures:

it returns the sauce for the given food
AssertionError: Values are not equal:
    [Diff] Actual / Expected
-   undefined
+   "\"ketchup\""
    at assertEquals
failures:
        it returns the sauce for the given food
test result: FAILED. 0 passed; 1 failed; 0 ignored; 0 measured; 0
filtered out (41ms)

error: Test failed
running 1 test from
file:///home/julian/sauce-recipe/sauce-server/sauce.test.ts
test it returns ketchup for fries ... ok (5ms)

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered
out (17ms)
```

And we have broken the front-end tests.

In order to fix the error the proper way. Do the following

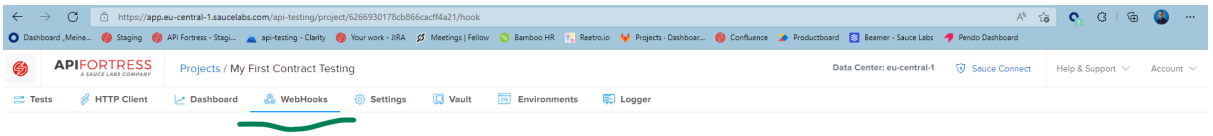
1. Restore the contract to use `sauce_name` instead of `deliciousSauce`
2. Change the implementation in `sauce-server.ts` to use `sauce_name` instead of `deliciousSauce`
3. Run the application again in localhost:8000.
4. Remove the contract in Sauce Labs APIT and upload again the new version, create a test, publish and execute. The result should be green.
5. Commit all changes and push to master.

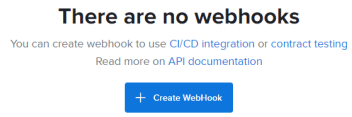
```
$ git push origin master
@@@@ This is better than Jenkins :) @@@@
running 1 test from
file:///home/julian/sauce-recipe/sauce-client/sauce.test.ts
test it returns the sauce for the given food ... ok (35ms)
```

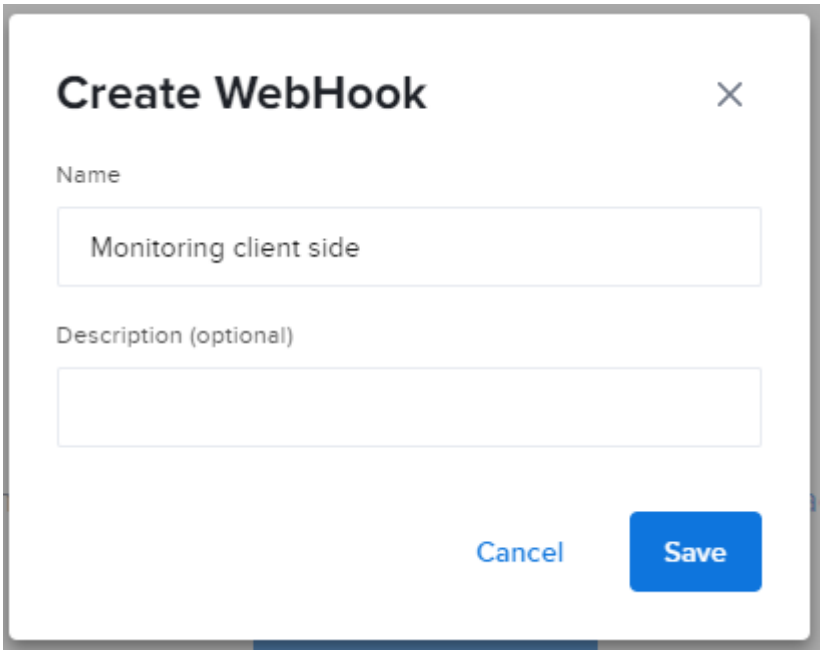
```
test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered
out (47ms)

running 1 test from
file:///home/julian/sauce-recipe/sauce-server/sauce.test.ts
test it returns ketchup for fries ... ok (5ms)

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered
out (18ms)

Enumerating objects: 16, done.
```

Wonderful! At this point we have the fries and ketchup, now let's put a cherry on top.

Create a WebHook by clicking on the on "Create a WebHook."







Copy and paste the webhook into `test-suite.sh` like this:

```
# Testing the client
cd sauce-client
WEBHOOK=<url-of-the-webhook>
docker run -d -v "$(pwd)/:/specs" -p 5000:5000
quay.io/saucelabs/piestry -u /specs/sauce-contract.yml \
  --logger $WEBHOOK/insights/events/_contract \
  --validate-request
sleep 5

/home/julian/.deno/bin/deno test --allow-net

#cleanup
sudo docker stop $(sudo docker ps -q)
cd ..

#testing the server
cd sauce-server
/home/julian/.deno/bin/deno test

docker run --rm quay.io/saucelabs/apifctl run -i <test-id> -T
<tunnel-id> -H $WEBHOOK
```

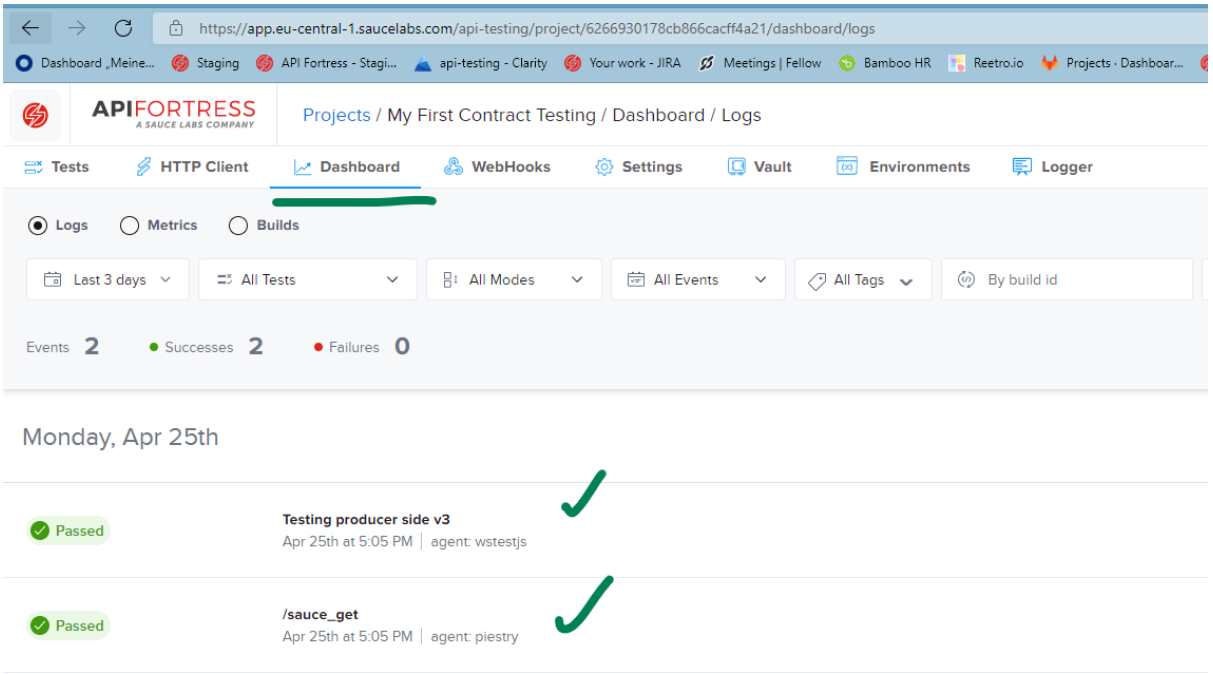
Notice that the url copied from the browser has a `{access_key}` that you have to manually change putting the access key from your user settings.

Save the changes, commit and push to master.

```
$ git push origin master
@@@@ This is better than Jenkins :) @@@@
running 1 test from
file:///home/julian/sauce-recipe/sauce-client/sauce.test.ts
test it returns the sauce for the given food ... ok (31ms)

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered
out (45ms)

running 1 test from
file:///home/julian/sauce-recipe/sauce-server/sauce.test.ts
test it returns ketchup for fries ... ok (5ms)

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered
out (18ms)
```

```
{
    "contextIds": [],
    "eventIds": [],
    "taskId": "0c1cfcd1-9c2a-4796-b6ee-8f290f29f9f4",
    "testIds": []
}
Enumerating objects: 5, done.
```

In our local the tests are still passing, which is good. Now go to the project dashboard and and check the results.



In the dashboard we can see that both consumer and producer are exercising the contract as expected.
Should Add a Final Section
Summarizing what we learned, and others things users can try.