Mock APIs

Introduction

Web APIs are usually implemented as HTTP endpoints. Playwright provides APIs to **mock** and **modify** network traffic, both HTTP and HTTPS. Any requests that a page does, including XHRs and fetch requests, can be tracked, modified and mocked. With Playwright you can also mock using HAR files that contain multiple network requests made by the page.

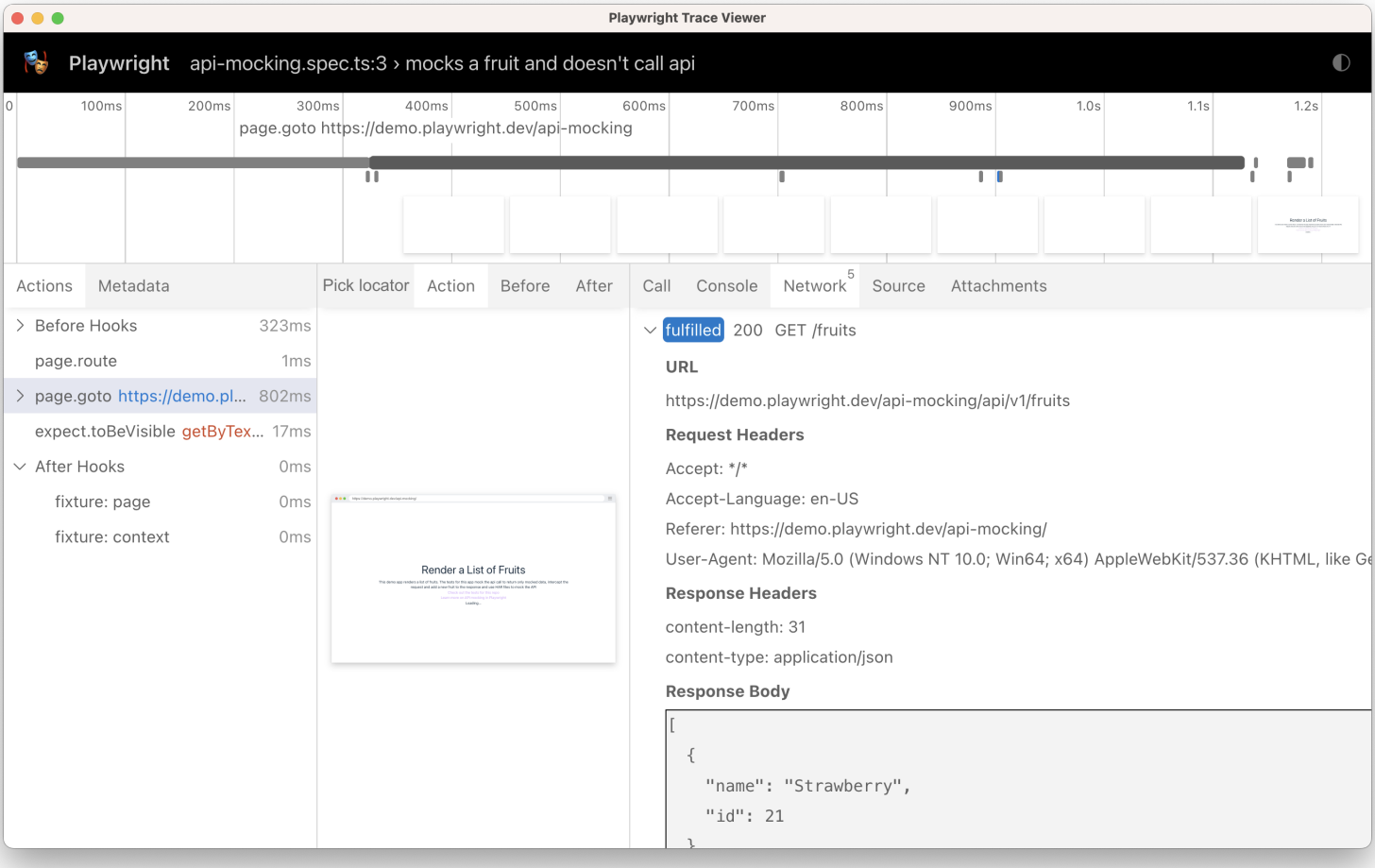
Mock API requests

The following code will intercept all the calls to `*/**/api/v1/fruits` and will return a custom response instead. No requests to the API will be made. The test goes to the URL that uses the mocked route and asserts that mock data is present on the page.

Sync **Async**

```python
def test_mock_the_fruit_api(page: Page):
    def handle(route: Route):
        json = [{"name": "Strawberry", "id": 21}]
        # fulfill the route with the mock data
        route.fulfill(json=json)

    # Intercept the route to the fruit API
    page.route("*/**/api/v1/fruits", handle)

    # Go to the page
    page.goto("https://demo.playwright.dev/api-mocking")

    # Assert that the Strawberry fruit is visible
    page.get_by_text("Strawberry").to_be_visible()
```

You can see from the trace of the example test that the API was never called, it was however fulfilled with the mock data.



Read more about advanced networking.

Modify API responses

Sometimes, it is essential to make an API request, but the response needs to be patched to allow for reproducible testing. In that case, instead of mocking the request, one can perform the request and fulfill it with the modified response.

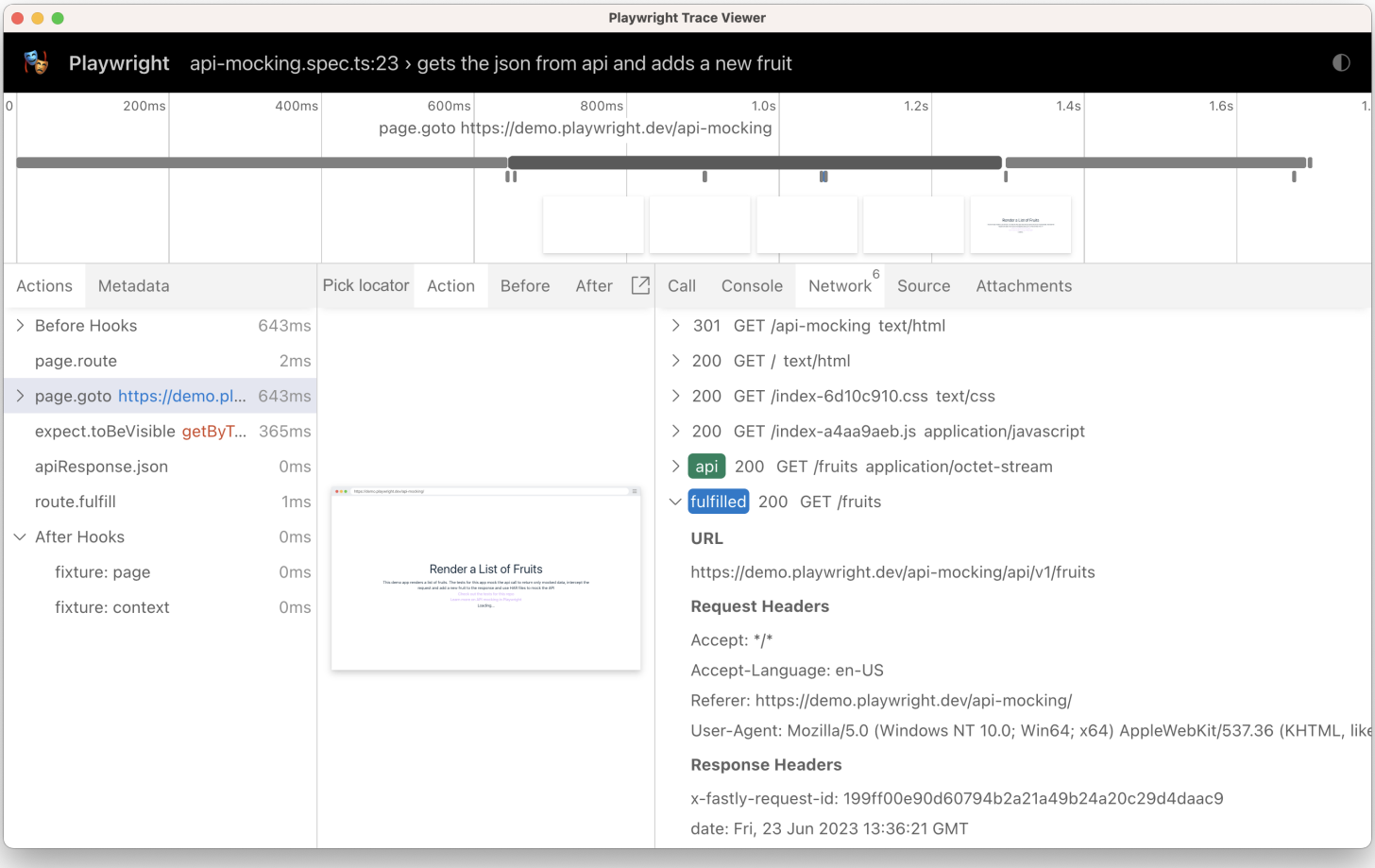
In the example below we intercept the call to the fruit API and add a new fruit called 'playwright', to the data. We then go to the url and assert that this data is there:

Sync **Async**

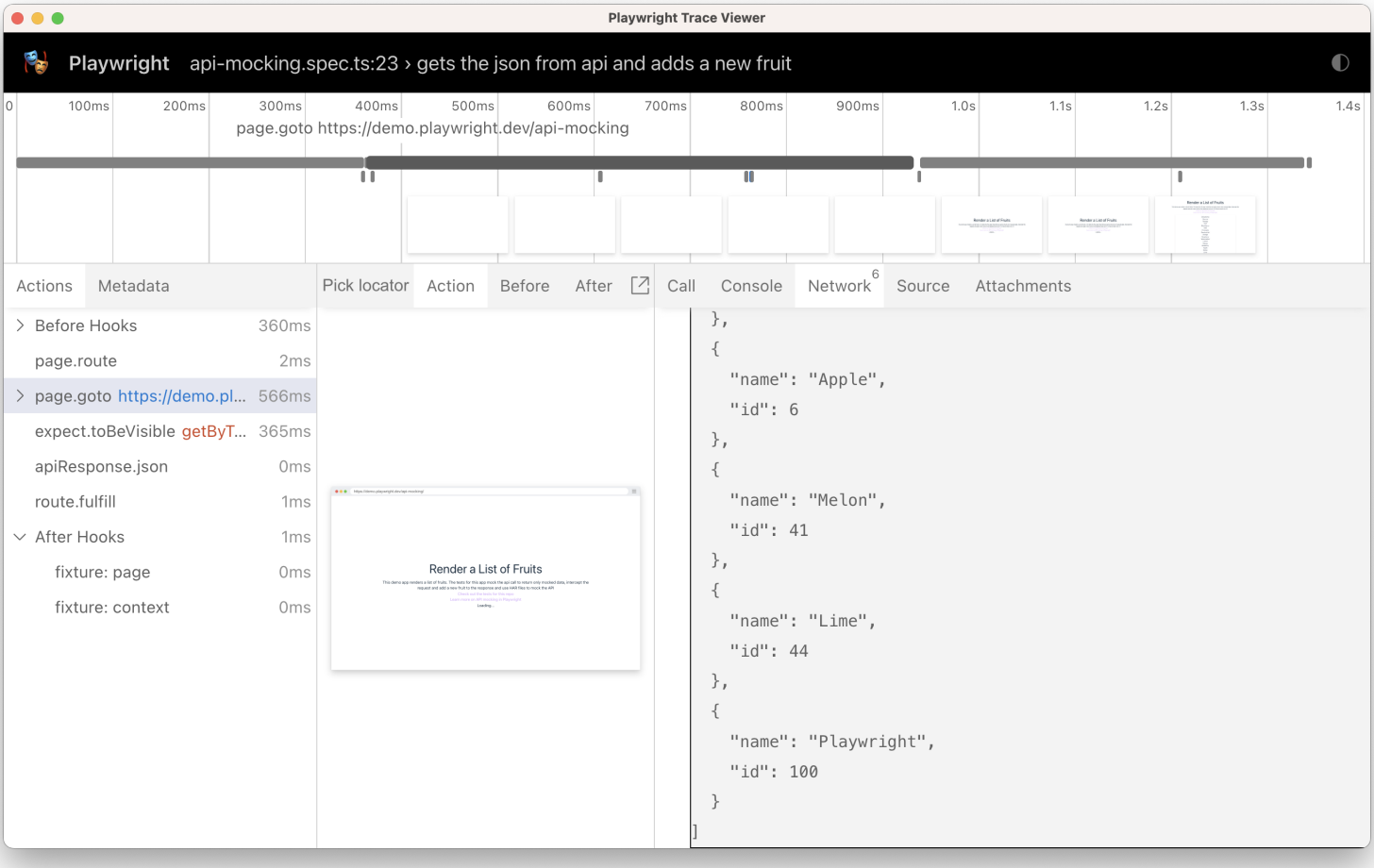
```python
def test_gets_the_json_from_api_and_adds_a_new_fruit(page: Page):
    def handle(route: Route):
        response = route.fulfill()
        json = response.json()
        json.append({ "name": "Playwright", "id": 100})
```

```
        # Fulfill using the original response, while patching the response body
        # with the given JSON object.
        route.fulfill(response=response, json=json)

    page.route("https://dog.ceo/api/breeds/list/all", handle)

    # Go to the page
    page.goto("https://demo.playwright.dev/api-mocking")

    # Assert that the new fruit is visible
    page.get_by_text("Playwright", exact=True).to_be_visible()
```

In the trace of our test we can see that the API was called and the response was modified.



By inspecting the response we can see that our new fruit was added to the list.



Read more about advanced networking.

Mocking with HAR files

A HAR file is an HTTP Archive file that contains a record of all the network requests that are made when a page is loaded. It contains information about the request and response headers, cookies, content, timings, and more. You can use HAR files to mock network requests in your tests. You'll need to:

1. Record a HAR file.
2. Commit the HAR file alongside the tests.
3. Route requests using the saved HAR files in the tests.

Recording a HAR file

To record a HAR file we use page.route_from_har() or browser_context.route_from_har() method. This method takes in the path to the HAR file and an optional object of options. The options

object can contain the URL so that only requests with the URL matching the specified glob pattern will be served from the HAR File. If not specified, all requests will be served from the HAR file.

Setting `update` option to true will create or update the HAR file with the actual network information instead of serving the requests from the HAR file. Use it when creating a test to populate the HAR with real data.

Sync **Async**

```
def records_or_updates_the_har_file(page: Page):
    # Get the response from the HAR file
    page.route_from_har("./hars/fruit.har", url="*/**/api/v1/fruits", update=True)

    # Go to the page
    page.goto("https://demo.playwright.dev/api-mocking")

    # Assert that the fruit is visible
    expect(page.get_by_text("Strawberry")).to_be_visible()
```

Modifying a HAR file

Once you have recorded a HAR file you can modify it by opening the hashed .txt file inside your 'hars' folder and editing the JSON. This file should be committed to your source control. Anytime you run this test with `update: true` it will update your HAR file with the request from the API.

```
[
  {
    "name": "Playwright",
    "id": 100
  },
  // ... other fruits
]
```

Replaying from HAR

Now that you have the HAR file recorded and modified the mock data, it can be used to serve matching responses in the test. For this, just turn off or simply remove the `update` option. This will run the test against the HAR file instead of hitting the API.

Sync Async

```python
def test_gets_the_json_from_har_and_checks_the_new_fruit_has_been_added(page:
Page):
    # Replay API requests from HAR.
    # Either use a matching response from the HAR,
    # or abort the request if nothing matches.
    page.route_from_har("./hars/fruit.har", url="*/**/api/v1/fruits",
update=False)

    # Go to the page
    page.goto("https://demo.playwright.dev/api-mocking")

    # Assert that the Playwright fruit is visible
    page.get_by_text("Playwright", exact=True).to_be_visible()
```

In the trace of our test we can see that the route was fulfilled from the HAR file and the API was not called.

If we inspect the response we can see our new fruit was added to the JSON, which was done by manually updating the hashed `.txt` file inside the `hars` folder.

HAR replay matches URL and HTTP method strictly. For POST requests, it also matches POST payloads strictly. If multiple recordings match a request, the one with the most matching headers is picked. An entry resulting in a redirect will be followed automatically.

Similar to when recording, if given HAR file name ends with `.zip`, it is considered an archive containing the HAR file along with network payloads stored as separate entries. You can also extract this archive, edit payloads or HAR log manually and point to the extracted har file. All the payloads will be resolved relative to the extracted har file on the file system.

Recording HAR with CLI

We recommend the `update` option to record HAR file for your test. However, you can also record the HAR with Playwright CLI.

Open the browser with Playwright CLI and pass `--save-har` option to produce a HAR file. Optionally, use `--save-har-glob` to only save requests you are interested in, for example API endpoints. If the har file name ends with `.zip`, artifacts are written as separate files and are all compressed into a single `zip`.

```
# Save API requests from example.com as "example.har" archive.
playwright open --save-har=example.har --save-har-glob="**/api/**"
https://example.coms
```

Read more about advanced networking.