UnCodemy

Playwright Network Mocking PDF

Playwright can intercept network requests made by the page with page.route. You can fulfill a request with fake data, abort it, change a real response, or replay recorded traffic from a HAR file. Mocking makes tests faster and more predictable and lets you test errors and slow responses that are hard to create in a real backend.

Key takeaways

- page.route matches URLs and lets you fulfill, abort, continue or modify requests.
- Mock errors, empty lists and slow responses that are hard to produce for real.
- Keep a few end-to-end tests with the real backend so integration stays covered.
- HAR files can record and replay traffic for stable tests.
- Start the route before the action that triggers the request.

What is Playwright network interception?

Network mocking in Playwright is the practice of intercepting browser requests and returning controlled responses, so that a test can check how the interface behaves without depending on a live backend.

Why mock the network?

Real back ends can be slow, unavailable, shared with other teams or hard to put into a given state. If you want to see how the page behaves when the server returns an empty list, a validation error or a failure, mocking is the straightforward way. It also removes noise from third-party services so that the test checks your front end only.

How do I fulfill a request with fake data?

```
import { test, expect } from '@playwright/test';

test('shows an empty state', async ({ page }) => {
  await page.route('**/api/orders', route =>
    route.fulfill({ status: 200, contentType: 'application/json', body: JSON.stringify([]) })
  );
  await page.goto('/orders');
  await expect(page.getByText('No orders yet')).toBeVisible();
});
```

Register the route before navigating or before the action that sends the call. The glob pattern matches the URL; you can also use a regular expression or a function.

What other route actions exist?

Action	Use
`route.fulfill`	Return a custom response
`route.abort`	Fail the request, for example to simulate offline
`route.continue`	Let the request go through, optionally with changed headers
`route.fetch`	Get the real response, then change it before fulfilling
`route.fallback`	Pass to the next matching handler

How do I test error handling?

Return status 500 with a message and assert that the page shows a helpful error, a retry button and no broken layout. Use `route.abort('failed')` to simulate a network failure. Delay a response with a short wait in the handler to test loading indicators and timeouts. These cases are rarely tested manually, yet users meet them often.

How do I modify a real response?

Call `const response = await route.fetch();`, read `await response.json()`, change a field and call `route.fulfill({ response, json: changed })`. This approach keeps most data real but forces one edge case, such as a very long product name or a zero price.

What is HAR replay?

A HAR file records requests and responses. Playwright can record one with `page.routeFromHAR('app.har', { update: true })` and replay it later without `update`. This gives deterministic tests for pages with many calls. Refresh the file when the API changes, and remove secrets from recordings before committing them.

How do I wait for or check a request?

Use `page.waitForRequest` or `page.waitForResponse` to confirm the page called the right endpoint. Start the wait before the click: `const respPromise = page.waitForResponse('**/api/cart'); await button.click(); const resp = await respPromise;`. You can also read `request.postDataJSON()` to check the payload the interface sent.

How much mocking is healthy?

Mock to test front-end logic and rare states, but keep some tests through the real stack, because mocked tests cannot find contract problems. A balanced suite has many fast mocked tests, some API tests and a small set of true end-to-end flows. Keep mock data in shared files and update it when the contract changes.

What mistakes should I avoid?

Registering routes too late, using patterns that are too broad, forgetting to unroute when needed, committing real tokens in HAR files and letting mock data drift away from the real API. Treat mock files as code and review them.

How do I organise mocks in a growing suite?

Put reusable handlers in a helper file with clear names such as mockEmptyOrders or mockServerError, and import them where needed. Keep response bodies in small JSON files near the tests so that data is easy to review and update. Add a short comment on which real endpoint each mock represents and when it was last checked against the API. This habit stops mocks from drifting away from reality and makes failures easier to explain.

How do I test slow responses and loading states?

Inside a route handler, wait for a short time before fulfilling the request, then assert that the loading indicator appears and that the final content replaces it. Also test a very long delay to verify timeout messages. Keep delays small so that the suite stays fast, and avoid fixed waits in the test itself: let the assertion retry until the indicator disappears.

What is a quick checklist before merging a mocked test?

Confirm the route is registered before the action, the URL pattern is narrow, the mock matches the real API shape, the test asserts what the user sees, and at least one unmocked flow covers the same feature.

Frequently asked questions

What does page.route do?

It intercepts requests that match a URL pattern so you can fulfill them with fake data, abort them, change them or let them continue.

Why mock network calls in UI tests?

Mocking makes tests faster and more stable and lets you create states, such as errors or empty lists, that are hard to get from a real server.

Can I change a real response instead of replacing it?

Yes. Use route.fetch to get the real response, modify the data and then call route.fulfill with the changed content.

What is a HAR file used for?

A HAR file records network traffic, and Playwright can replay it so that pages with many calls behave the same way on every run.

Should every test use mocks?

No. Keep some end-to-end tests with the real backend so integration problems are still caught alongside fast mocked tests.

Sources and further reading

- Playwright documentation
- Playwright network guide
- Playwright mock APIs

About this PDF

These PDFs are study material for learning and revision. They are not a certificate, a degree or a promise of a job, and technology changes, so check official documentation for the latest details. Spot an error? Write to info@uncodemy.com and the trainer team will review it.

More free resources and courses: uncodemy.com | info@uncodemy.com | +91 98183 66550