Part 6: Automating the Setup

An Introduction to Fixtures

In our journey so far, we've designed our "craftsmen" (`Page Objects`) and we've seen how our tests send them high-level messages. But there's a common problem we still need to solve: **setup**.

Almost every test needs some setup work done before it can begin. For example, a test that checks the "My Account" page must have a user logged in *first*. A test that edits a product must first *create* that product.

Writing this setup code at the beginning of every single test is repetitive and clutters our test files. This is where Playwright has an elegant and powerful solution called **Fixtures**.

What is a Fixture? Your Personal Test Assistant

The best way to think of a **fixture** is as a personal assistant or a stagehand for your test.

Imagine a play. Before the main actor comes on stage for a scene, a stagehand has already made sure all the props are in the right place, the lighting is correct, and the backdrop is set. The actor doesn't have to worry about any of that; they just walk on stage and start their scene.

Fixtures do the exact same thing for our tests. They are pieces of code that run *before* a test to set up everything it needs, providing a perfect, ready-to-go environment.

You have already been using the most important fixture without even realizing it: the `page` object!

When you write `test('my test', async ({ page }) => { ... })` , you are telling Playwright, "For this test, I need the `page` assistant." Before your test code runs, Playwright automatically performs the setup: it launches a browser, creates a clean session, and opens a new tab (`page`) for you to use.

How Fixtures Are Used in a Test

A fixture is "used" by adding it as an argument inside the curly braces `{}` of the `test` function.

```
test('a test description', async ({ fixture1, fixture2 }) => {
  // This is the test body.
  // By the time this code runs, fixture1 and fixture2 are ready to u
});
```

When Playwright prepares to run this test, it reads the list of requested fixtures. It then finds the code for `fixture1` and `fixture2`, executes their setup steps, and passes the resulting objects into your test.

Creating Our Own Custom Fixture

This is where fixtures become incredibly powerful. We are not limited to the built-in ones like `page`. We can create our own custom assistants to handle our project's specific, repetitive setup tasks.

A Common Problem: In many of our tests, the first thing we do is create an instance of our `LoginPage`.

```
// Repetitive setup in every test...
test('a test that needs the login page', async ({ page }) => {
    const loginPage = new LoginPage(page); // <-- This line is repeat
    await loginPage.navigateTo();
    // ...
});

test('another test that also needs the login page', async ({ page })
    const loginPage = new LoginPage(page); // <-- This line is repeat
    await loginPage.performLogin('user', 'pass');
    // ...
});
```

We can eliminate this repetition by creating a `loginPage` fixture. We do this in a special file (e.g., `test-fixtures.ts`) where we extend Playwright's basic `test` object.

The Solution:

```
// In a file like 'test-fixtures.ts'

import base from '@playwright/test';
import { LoginPage } from './page-objects/LoginPage';
```

```
// This is where we teach Playwright about our new assistant
export const test = base.extend<{ loginPage: LoginPage }>({
  // Define the new fixture named 'loginPage'
  loginPage: async ({ page }, use) => {
    // 1. SETUP: This code runs before each test that uses it.
    const loginPage = new LoginPage(page);

    // 2. USE: This hands the created loginPage over to the test.
    // The test itself runs right here.
    await use(loginPage);

    // 3. TEARDOWN: This code runs after the test is finished (optior
    // You could add cleanup logic here if needed.
  },
});
```

The most important part is the `use` function. Think of it as the stagehand's cue:

1. **Setup Phase:** Everything *before* `await use()` is the setup. Here, we create the `LoginPage` instance.
2. **Execution Phase:** The `await use(loginPage)` line effectively says, "Okay, the stage is set. I'm providing the `loginPage` object. Now, run the actual test."
3. **Teardown Phase:** Everything *after* `await use()` is the cleanup phase, which runs after the test completes.

The Payoff: Cleaner and Simpler Tests

Now that we have our custom `loginPage` fixture, our tests become much cleaner. We just have to ask for our new assistant by name.

```
// In our test file 'login.spec.ts'

// Import 'test' from our special fixture file, not from '@playwright
import { test } from './test-fixtures';
import { expect } from '@playwright/test';

test('user can log in successfully', async ({ loginPage, page }) => {
    // No more manual setup! The 'loginPage' fixture is ready to go.
    await loginPage.navigateTo();
    await loginPage.performLogin('testuser', 'password123');
```

```
    await expect(page).toHaveURL('/dashboard');
});
```

Notice the test is now shorter and more focused. It doesn't care *how* the `loginPage` object is created; it just trusts the fixture to provide it. By creating fixtures for our Page Objects, we automate our setup, reduce repetition, and make our tests easier to write and read.