Page Object Model in Playwright

What is the Page Object Model (POM)?

The **Page Object Model** is a design pattern used in automated testing, particularly for web applications. It helps structure the code by separating the page elements and actions associated with a webpage into distinct files. Each page in a web application has its own file, which contains all the page elements and interactions related to that page. This separation helps in maintaining and reusing code more effectively.

Structure of Page Object Model:

1. **Page Elements File**: Contains the locators (selectors) and methods to interact with the elements on a page.
2. **Test Case File**: Contains the actual test scripts that call the methods defined in the page object files.

In Playwright, this model is used to manage tests efficiently by keeping page-related elements and actions isolated from the test scripts.

Why Do We Need the Page Object Model?

Problem without Page Object Model:

Imagine we are working with a web application that has **10 pages**. Let's consider two test cases:

1. **Test Case 1:**
   - Log in to the application
   - Select a product from the homepage
   - Navigate to the add-to-cart page
   - Verify that the product is added to the cart.
2. **Test Case 2:**
   - Log in to the application
   - Navigate to the add-to-cart page
   - Buy the product.

In both test cases, we are navigating through multiple pages. Without Page Object Model, we would have to repeat the locators and interactions for each page in every test case. This leads to **code duplication** and makes maintenance difficult.

Solution with Page Object Model:

With Page Object Model, we **define the page elements and actions** for each page in a separate file. The test case files will only interact with these pages, which helps to reduce duplication and makes the tests more manageable.

Benefits of Using Page Object Model:

- **Reduced Code Duplication**: By defining page elements in separate files, we avoid repeating the same code in multiple test cases.
- **Easier Maintenance**: If an element's locator changes, we only need to update it in the page object file rather than all test cases.
- **Improved Readability**: The test cases become simpler and more focused on the behavior rather than the details of interacting with the page.
- **Reusability**: Page object methods can be reused across multiple test cases, making the code more modular.

Example of Page Object Model in Playwright

Let’s consider an example of a simple login test using the Page Object Model.

1. LoginPage File (Page Object)

```
class LoginPage {
  constructor(page) {
    this.page = page;
    this.usernameField = page.locator('#username');
    this.passwordField = page.locator('#password');
    this.loginButton = page.locator('#login');
  }

  async login(username, password) {
    await this.usernameField.fill(username);
    await this.passwordField.fill(password);
    await this.loginButton.click();
  }
}

module.exports = LoginPage;
```

2. HomePage File (Page Object)

```
class HomePage {
  constructor(page) {
    this.page = page;
    this.productLink = page.locator('.product-link');
  }

  async selectProduct() {
    await this.productLink.click();
  }
}

module.exports = HomePage;
```

3. Test Case File (Using Page Objects)

```
const { test, expect } = require('@playwright/test');
const LoginPage = require('./LoginPage');
const HomePage = require('./HomePage');

test('Login and add product to cart', async ({ page }) => {
  // Create instances of page objects
  const loginPage = new LoginPage(page);
  const homePage = new HomePage(page);

  // Step 1: Login to the application
  await loginPage.login('testuser', 'password123');

  // Step 2: Select a product from the homepage
  await homePage.selectProduct();

  // Add more steps as needed to verify cart
  // Example: expect(await page.locator('.cart')).toContainText('Product
added');
});
```

Conclusion

The **Page Object Model** improves the **structure** and **maintainability** of automated tests. It helps in keeping the tests DRY (Don't Repeat Yourself) by moving page-specific interactions to separate files. This leads to **cleaner code**, **easy updates**, and **better scalability** as your application grows and evolves.

Understanding Page Object Model (POM) in Automation Testing Step by Step

What is the Page Object Model (POM)?

The **Page Object Model** (POM) is a design pattern in test automation where each web page of the application is represented as a class. The web elements on that page are represented as variables inside the class, and actions or behaviors that can be performed on the page are represented as methods inside that class.

Structure of the Pages Folder

In your project, you'll have a folder called `pages` which contains JavaScript files for each web page of the application you want to automate. The structure will look like this:

```
Pages
    ├── LoginPage.js
    ├── HomePage.js
    ├── CartPage.js
```

Creating the LoginPage Class

Now, let’s go through creating the `LoginPage` object class.

Step 1: Defining the LoginPage Class

To start, we will define a class called `LoginPage` in `LoginPage.js`. This class will represent the Login page of the application.

```
class LoginPage {
}
```

Step 2: Adding the Constructor

Inside the class, we define a **constructor**. The constructor is a special method in JavaScript that gets executed when an object of the class is created. It is used to initialize the class and pass in any required data or objects.

Why Use a Constructor?

- **Initialization:** When you create an object of the `LoginPage` class, the constructor gets invoked automatically, and it’s where you can initialize variables or set up necessary data.
- **Test Fixture:** In this case, we need a reference to the **page fixture** passed from the test case. This is necessary because it allows us to interact with the web page elements on the test page.

Example of Constructor in `LoginPage`

```
class LoginPage {
  constructor(page) {
    this.page = page;
  }
}
```

Explanation:

- `page` is passed to the constructor when we create an object of `LoginPage`.
- `this.page` is used to store that **page fixture** for later use. This `page` will allow us to interact with the actual web page in our test.

Adding Locators to the Page Object

Once we have the `page` fixture, we can use it to interact with the elements on the page. Let's add locators for the **username field**, **password field** and **login button**.

Step 3: Define Web Element Locators

You need to store each web element's locator in a variable. To do this, you can use the `this` keyword to define these locators as properties of the `LoginPage` class.

Example:
```
class LoginPage {
  constructor(page) {
    this.page = page;  // Page fixture passed from the test case
    this.userNameField = "#usernameId";  // Locator for the username field
    this.passwordField = "#passwordId";  // Locator for the password field
    this.loginButton = "//button[@loginButton]";  // Locator for the login
button
  }
}
```

- **Explanation of Locators:**
  - `this.userNameField`: Locator for the username input field.
  - `this.passwordField`: Locator for the password input field.
  - `this.loginButton`: Locator for the login button.

We are using **`this`** for the following reasons:

Why Use `this` for Each Element?

The `this` keyword is essential in JavaScript, particularly inside a class. Here's why we use it:

1. **Referring to Instance Variables:**
   - In a class, **`this`** is used to refer to instance variables (properties) of the class.
   - When we create an instance of the `LoginPage` class, the **`this`** keyword helps us distinguish between instance properties (like `this.userNameField`) and local variables (if any) within methods.
2. **Accessing Page Elements from Test Case:**
   - When you create an object of `LoginPage` in your test file, you are able to access the web elements (username, password, etc.) through the instance object. For example:

     ```
     const loginPage = new LoginPage(page);
     ```

3. **Setting the Test Context:**
   - The **`this.page`** is the connection to the actual page in the test case, allowing you to interact with the elements like clicking buttons, filling out fields, etc.

Test Case Example: How `LoginPage` is Used

In the **test case file**, you will import the `LoginPage` class and create an object of that class by passing the page fixture.

Example Test Case:

```
const { test, expect } = require('@playwright/test');
const LoginPage = require('./pages/LoginPage');

test('Login test', async ({ page }) => {
  // Creating an object of LoginPage class and passing the page fixture
  const loginPage = new LoginPage(page);

  // Using the locators from the LoginPage class
  await page.fill(loginPage.userNameField, 'testuser');
  await page.fill(loginPage.passwordField, 'password123');
  await page.click(loginPage.loginButton);

  // Assert successful login
  await expect(page).toHaveURL('https://example.com/home');
});
```

Explanation of Test Case:

- **`page` Fixture:** The `page` fixture is automatically passed by the test framework (e.g., Playwright, Cypress) to interact with the browser.
- **`new LoginPage(page)`:** Creates an object of `LoginPage`, and the `page` fixture is passed to the constructor.
- **Accessing Locators**: The `loginPage.userNameField`, `loginPage.passwordField`, and `loginPage.loginButton` are used to locate the elements and perform actions on the page.

Full Example with DOM Structure

Let's assume this is the structure of the HTML for the login page:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Login</title>
</head>
<body>
  <form id="loginForm">
    <input type="text" id="usernameId" name="username"
placeholder="Username">
    <input type="password" id="passwordId" name="password"
placeholder="Password">
    <button type="submit" id="loginButton">Login</button>
  </form>
</body>
</html>
```

- The locators (`#usernameId`, `#passwordId`, and `#loginButton`) refer to the elements in the above HTML.

Conclusion

- **Why `this` Keyword**: We use **`this`** to refer to the instance variables within the class. It helps us assign values to the page elements and makes them accessible throughout the class.
- **Why Constructor**: The constructor is used to initialize the page object with necessary fixtures (like the `page` object) that allow us to locate and interact with web elements.
- **Test Case Interaction**: In the test case, when we create an object of `LoginPage`, it gives us access to the web elements defined in the class and allows us to interact with the application.

This approach makes the tests more maintainable and readable by separating the test logic from the page element interactions.

Understanding the LoginPage Class in Page Object Model

In Page Object Model (POM), the **LoginPage class** represents the login page of your application. It contains:

1. **Locators** for elements on the page.
2. **Functions (methods)** to perform actions on the page (e.g., login, navigation).

LoginPage Class Structure

```
class LoginPage {
  constructor(page) {
    // Constructor receives the page fixture from the test
    this.page = page; // 'this.page' will refer to the current page in the
test
    // Locators for the elements on the LoginPage
    this.userNameField = "#usernameId";  // Locator for the username field
    this.passwordField = "#passwordId";  // Locator for the password field
    this.loginButton = "//button[@id='loginButton']"; // Locator for the
login button
  }

  // Method to navigate to the LoginPage URL
  async gotoLoginPage() {
    await this.page.goto('https://yourwebsite.com/login'); // Navigates to
the login page
  }

  // Method to perform login action
  async login(username, password) {
    // Filling the username and password fields and clicking the login
button
```

```
    await this.page.locator(this.userNameField).fill(username);
    await this.page.locator(this.passwordField).fill(password);
    await this.page.locator(this.loginButton).click();
  }
}

// Export the LoginPage class to use it in test files
module.exports.LoginPage = LoginPage;
```

Key Points to Understand

1. **Constructor**: The constructor receives the `page` fixture from the test. This `page` object represents the browser page in which all interactions will occur.
   - **`this.page`**: Refers to the browser page instance that you can use to interact with the page elements.
   - **Element Locators**: In the constructor, locators are defined for username, password, and login button. These are represented by the `this.userNameField`, `this.passwordField`, and `this.loginButton` variables.
2. **`async` and `await`**:
   - **`async`**: This keyword is used in the method definition (`async gotoLoginPage()`, `async login()`) to indicate that these methods will involve asynchronous operations (e.g., waiting for elements to load or actions to complete).
   - **`await`**: This is used inside async functions to pause the execution until the promise (e.g., page load or clicking a button) is resolved. Without `await`, the code would not wait for the page actions to finish before moving on to the next operation, which could lead to errors.

   Example:

   ```
   async gotoLoginPage() {
     await this.page.goto('https://yourwebsite.com/login');  // Wait for
   the page to load
   }
   ```

3. **Exporting the Class**:
   - By using `module.exports.LoginPage = LoginPage;`, we are making the `LoginPage` class available for import into other test files.

Using the `LoginPage` in the Test File

Now, we'll use the `LoginPage` class in our test file to perform actions like navigating to the login page and logging in with test credentials.

Test File Structure (e.g., `login.spec.js`)

```
// Import required modules
```

```
const { test, expect } = require('@playwright/test');  // Playwright
testing functions
const { LoginPage } = require('../pages/LoginPage');   // Import the
LoginPage class

// Define the test
test('Login Test', async ({ page }) => {
  // Create an instance of the LoginPage class and pass the 'page' fixture
from Playwright
  const loginPage = new LoginPage(page);

  // Perform actions using the methods defined in the LoginPage class
  await loginPage.gotoLoginPage();  // Go to the login page
  await loginPage.login('Yogesh', 'testLife@123');  // Perform login action
with credentials

  // Verify that the user has successfully logged in (you can adjust this
based on your actual app)
  await expect(page).toHaveURL('https://yourwebsite.com/home'); // Check
that the user is redirected to the home page
});
```

Explanation of the Test File

1. **Importing Dependencies**:
   - We import `test` and `expect` from Playwright to define and assert the test.
   - We import the `LoginPage` class so we can use the functions like `gotoLoginPage()` and `login()` defined in it.
2. **Test Case**:
   - **Creating an instance of LoginPage**: `const loginPage = new LoginPage(page);`
     - The `page` fixture (provided by Playwright) is passed to the constructor of `LoginPage`. This allows the `LoginPage` class to interact with the page on which the test is running.
   - **Calling the functions from LoginPage**:
     - `await loginPage.gotoLoginPage();` will navigate to the login page.
     - `await loginPage.login('Yogesh', 'testLife@123');` will fill the username and password fields, then click the login button.
   - **Assertion**:
     - `await expect(page).toHaveURL('https://yourwebsite.com/home');` asserts that the URL has changed to the home page after successful login.

Why Export the Class?

In JavaScript, **exports** allow us to share the functionality of a class, function, or object across different files. By exporting the `LoginPage` class in the `LoginPage.js` file, we make it accessible in our test file.

```
// In LoginPage.js
module.exports.LoginPage = LoginPage; // Exporting the LoginPage class

// In test file (login.spec.js)
const { LoginPage } = require('../pages/LoginPage'); // Importing the
LoginPage class
```

Summary

1. **LoginPage Class**:
   - Contains **locators** for the page elements.
   - Contains **methods** (functions) to interact with those elements, such as navigating to the login page and performing login actions.
2. **Test File**:
   - You **import the `LoginPage` class** into the test file.
   - **Create an instance** of the `LoginPage` class, passing the `page` fixture to the constructor.
   - **Call the methods** (e.g., `gotoLoginPage()`, `login()`) to perform actions on the login page.
3. **Why Use `async` and `await`?**
   - **`async`**: Declares that the method is asynchronous and will involve operations that return promises (e.g., loading a page or clicking an element).
   - **`await`**: Pauses the execution until the asynchronous operation (like filling a field or navigating to a page) is complete.

By using Page Object Model (POM), you make your test code cleaner, more maintainable, and easier to scale. You can change locators in the `LoginPage` class without affecting the tests themselves.

Now once we login we will land into homepage and we need to select the product

The products are in homepage so we need to create one page object class for that homepage

Our test case 1 is

1. **Test Case 1**:
   - Log in to the application - completed
   - Select a product from the homepage from list of products
   - Navigated to the add-to-cart page
   - Click on add to cart button
   - In dialog box click of if the box has text added
   - Now go to cart page and verify the product is added

For each main step, we'll follow this structure:

1. **DOM structure** (example)
2. **Page Object Class** (with explanation)
3. **Update to `testcase1.spec.js`**

Step 1: Login Step

1. Example DOM for Login Page

```
<form id="loginForm">
  <input type="text" id="username" placeholder="Enter Username" />
  <input type="password" id="password" placeholder="Enter Password" />
  <button id="loginButton">Login</button>
</form>
```

2. Page Object Class – `LoginPage.js`

```
exports.LoginPage = class LoginPage {
  constructor(page) {
    this.page = page;
    this.usernameField = '#username';
    this.passwordField = '#password';
    this.loginButton = '#loginButton';
  }

  async gotoLoginPage() {
    await this.page.goto('https://yourwebsite.com/login');
  }

  async login(username, password) {
    await this.page.fill(this.usernameField, username);
    await this.page.fill(this.passwordField, password);
    await this.page.click(this.loginButton);
  }
};
```

3. Update `testcase1.spec.js` – Login Part Only

```
const { test, expect } = require('@playwright/test');
const { LoginPage } = require('../pages/LoginPage');

test('test case one - step 1: login', async ({ page }) => {
  const loginPage = new LoginPage(page);

  // Step 1: Login
  await loginPage.gotoLoginPage();  // Navigate to login page
  await loginPage.login('Yogesh', 'testLife@123');  // Login with
credentials
  await expect(page).toHaveURL('https://yourwebsite.com/home');  // Verify
redirection
});
```

Step 1 completed. Now moving on to Step 2.

Created By: Yogesh Pandian

Step 2: Select Product from Homepage

1. Example DOM for Homepage Product Selection

```
<div class="product-list">
  <div class="product-item">
    <a>iPhone 20 Pro Max</a>
  </div>
  <button id="cartButton">Add to Cart</button>
</div>
```

Note: After clicking the product, a dialog might pop up like:

```
// Dialog message: "Product successfully added to cart"
```

2. Page Object Class – `HomePage.js`

```
exports.HomePage = class HomePage {
  constructor(page) {
    this.page = page;
    this.productList = '//div[@class="product-item"]//a';
    this.addToCartButton = '//button[@id="cartButton"]';
  }

  async selectProduct(productName) {
    const products = await this.page.$$(this.productList);
    for (const product of products) {
      if ((await product.textContent()).trim() === productName) {
        await product.click();
        break;
      }
    }

    // Handle popup dialog after product is clicked
    this.page.on('dialog', async (dialog) => {
      if (dialog.message().includes('added')) {
        await dialog.accept();
      }
    });

    await this.page.locator(this.addToCartButton).click();
  }
};
```

3. Update `testcase1.spec.js` – Add Step 2

```
const { test, expect } = require('@playwright/test');
const { LoginPage } = require('../pages/LoginPage');
const { HomePage } = require('../pages/HomePage');

test('test case one - step 2: login and select product', async ({ page })
=> {
  const loginPage = new LoginPage(page);
  await loginPage.gotoLoginPage();
  await loginPage.login('Yogesh', 'testLife@123');
  await expect(page).toHaveURL('https://yourwebsite.com/home');

  const homePage = new HomePage(page);
  await homePage.selectProduct('iPhone 20 Pro Max');
});
```

Step 2 completed. Now moving on to Step 3.

Step 3: Go to Cart Page

---

1. Example DOM for Cart Link

```
<a id="cartLink">Cart</a>
```

---

2. Update Page Object Class – Add `goToCartPage()` to `HomePage.js`

Already included earlier, but here it is again for clarity:

```
async goToCartPage() {
  await this.page.locator(this.cartLink).click();
}
```

Add this to `HomePage.js`:

```
this.cartLink = '#cartLink';
```

---

3. Update `testcase1.spec.js` – Add Step 3

```
const { test, expect } = require('@playwright/test');
const { LoginPage } = require('../pages/LoginPage');
const { HomePage } = require('../pages/HomePage');

test('test case one - step 3: login, select product, go to cart', async ({
page }) => {
  const loginPage = new LoginPage(page);
  await loginPage.gotoLoginPage();
  await loginPage.login('Yogesh', 'testLife@123');
  await expect(page).toHaveURL('https://yourwebsite.com/home');

  const homePage = new HomePage(page);
  await homePage.selectProduct('iPhone 20 Pro Max');
  await homePage.goToCartPage(); // Navigate to cart page
});
```

Step 3 completed. Now moving on to Step 4.

Step 4: Verify Product in Cart

---

1. Example DOM for Cart Page

```
<table>
  <tr class="cart-item">
    <td>1</td>
    <td>iPhone 20 Pro Max</td>
  </tr>
</table>
```

---

2. Page Object Class – `CartPage.js`

```
exports.CartPage = class CartPage {
  constructor(page) {
    this.page = page;
    this.numOfProducts = '//tr[@class="cart-item"]/td[2]';
  }

  async verifyProductInCart(addedProductName) {
    const productsInCart = await this.page.$$(this.numOfProducts);
    for (const product of productsInCart) {
      if ((await product.textContent()).trim() === addedProductName) {
        return true;
      }
    }
    return false;
  }
};
```

---

3. Final Update to `testcase1.spec.js` – Add Step 4

```
const { test, expect } = require('@playwright/test');
const { LoginPage } = require('../pages/LoginPage');
const { HomePage } = require('../pages/HomePage');
const { CartPage } = require('../pages/CartPage');

test('test case one - complete add to cart flow', async ({ page }) => {
  const loginPage = new LoginPage(page);
  await loginPage.gotoLoginPage();
  await loginPage.login('Yogesh', 'testLife@123');
  await expect(page).toHaveURL('https://yourwebsite.com/home');
  const homePage = new HomePage(page);
  await homePage.selectProduct('iPhone 20 Pro Max');
  await homePage.goToCartPage();

  const cartPage = new CartPage(page);
  const result = await cartPage.verifyProductInCart('iPhone 20 Pro Max');
  expect(result).toBe(true);  //  Verify the product exists in the cart
});
```

Final Outcome

You now have:

- Modular **Page Object Classes**
- Clean and clear **`testcase1.spec.js`**
- **Step-by-step flow** with dialog handling, product selection, and cart verification