COURSEWARE

The POM Design Pattern

Contents

Overview

The **Page Object Model (POM) Design Pattern** enhances the maintainability of a codebase by having one class per page of the web application being tested. The POM class contains all the `WebElement`s which will be interacted with on the Web page. It also contains all the methods which are needed to interact with those `WebElement`s.

If we look at a simple Web page like wikipedia.org:

Each hyperlink, button and search bar which will be interacted with will be a class variable which the methods will interact with.

For instance, a search method would be put in the class which would first send keys to the *input* `WebElement`, before clicking the search button.

By having all the `WebElement`s and the methods which interact with those `WebElement`s all in the same class, it makes tests very maintainable.

Tutorial

For this tutorial we will use SauceDemo.

Setting up the Web page

For each Web page used in testing, a separate class is created to load the elements on that page:

▶ SauceDemoLandingPage

1) For the constructor the driver is passed through to the POM. This is so that if any driver methods need to be called from within the class they should be made accessible. The second reason is so that `PageFactory.initElements()` can be run, which will work with the `@FindBy` annotations to instantiate found elements.

2) The user input field has an `id` of `user-name`.

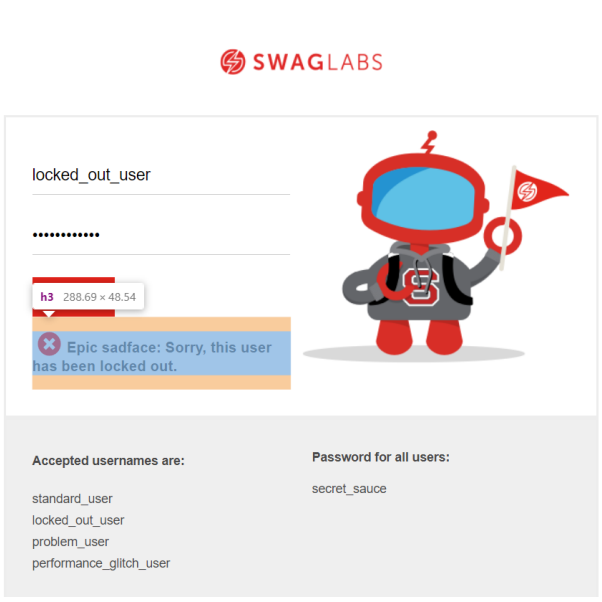
3) The password input field has an `id` of `password`.

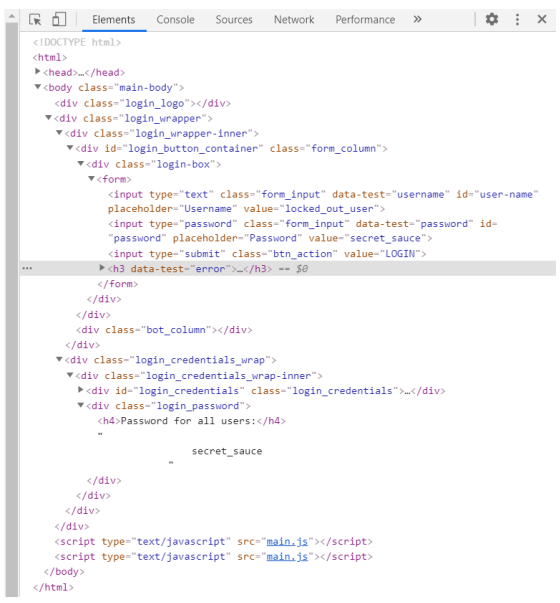
4) The Login button does not have an `id`, which means that another selector must be used. `ClassName` works well here - however, there is a possibility of this test breaking if another action button were to be introduced to the page, as the test would no longer be able to uniquely identify the correct button. If this were to happen, more criteria could be added by using more specific `@FindBy` annotations.

5) Not all login elements in the form have any uniquely identifiable attributes, such as the `.` Therefore multiple criteria can be applied using `@FindBy`s. In this case a class of `"login-box"` is searched for, inside that a `"form"` element is found, and then all elements within that form (*) are returned into a `List`.

6) The login method interacts with the login form elements, takes in the parameters from the test class and then submits the form.

7) Some elements are hard to specify. If we have a look at the element which occurs when we unsuccessfully log in, the element does not have anything specific about it other than the attribute `data-test`:





It would be possible to search for a tag name of `"h3"` or use an `XPath`, but this would make the test more *fragile* - if there is a slight change to the front-end it could cause the test to unexpectedly fail (or worse still, unexpectedly pass!).

Therefore the most optimal solution is to grab all `WebElement`s in the form, and iterate over them until we find the one with the `"data-test"` attribute.

Setting up the Tests

Once the page(s) have been set up the tests can be written:

▶ SauceDemoTest

1) Before the beginning of each test, a new `ChromeDriver` is instantiated. This is done so that each test is independent.

2) The driver goes to the URL held within the `SauceDemoLandingPage` class. This is to prevent magic strings and help maintainability if the Web page is moved. Next, an instance of the landing page is created, and the elements are then found in the current driver instance. Selenium fills in the details and the method submits the form. The successful submission of the form should take us to the inventory page. The inventory page has its own page class (not shown here for readability), which has its own URL value inside it.

3) Here the same steps are taken as 2 (above). However this time the values are expected to give a locked out response. The page is then tested to check that an error appears on the screen.

4) Here the same steps are taken as 2 again, but this time we test for an error by passing incorrect values. In this case, Selenium is used to check the flow of an application is working correctly, rather than the actual response.

```
*(If you wanted to check the actual response (i.e. checking the different responses returned from the different errors), then a unit testing JavaScript library such as [JEST](https://jestjs.io/), or an api testing tool such as [Rest-Assured](http://rest-assured.io/) or [Postman](https://www.postman.com/) would be better suited to this task.)*
```

5) Finally, the driver is closed, so that resources dedicated to opening the driver connection are released.

Exercises

1. Login with the correct credentials
2. Click "Add To Basket" on all items shown
3. Click the basket icon
4. Check that you are taken to the correct URL
5. Make sure all clicked items are on the Web page