

Selenium WebDriver: Page Object Pattern & Page Factory

Writing automated tests is more than just a luxury for any agile software development team. It is a need, and is an essential tool to find bugs quickly during early phases of software development cycles. When there is a new feature that is still in development phase, developers can run automated tests and see how other parts of the system are affected by those changes.

Through [test automation](#), it is possible to lower the cost of bug fixing and bring overall improvement to quality assurance (QA) process. With proper tests, developers get a chance at finding and resolving bugs even before it gets to QA. Test automation further helps us to automate test cases and features that are constantly regressing. This way QAs have more time in testing other parts of the application. Moreover, this helps in ensuring quality of the product in production releases. As a result, we get products that are effectively more stable, and a QA process that is more efficient.

Although writing automated tests may seem like an easy task for developers and engineers, there is still the possibility of ending up with poorly implemented tests, and the high cost of code maintenance in any agile process. Trying to constantly deliver changes or features in any agile development project can prove to be costly when tests are involved. Changing one element on a web page that 20 tests rely on will require one to go through these 20 test routines and update each one to adapt to this newly introduced change. Not only can this be really time consuming, but a serious de-motivating factor when it comes to implementing automated tests early on.

But, what if we could make the change in one place only, and have every relevant test routine use it? In this article, we will take a look at automated tests in Selenium, and how we can use Page Object models to write maintainable and reusable test routines.

Page Object Model in Selenium

Page Object model is an object design pattern in Selenium, where web pages are represented as classes, and the various elements on the page are defined as variables on the class. All possible user interactions can then be implemented as methods on the class:

```
clickLoginButton();  
setCredentials(user_name, user_password);
```

Since well-named methods in classes are easy to read, this works as an elegant way to implement test routines that are both readable and easier to maintain or update in the future. For example:

In order to support Page Object model, we use Page Factory. Page Factory is an extension to Page Object and can be used in various ways. In this case we will use Page Factory to initialize web elements that are defined in web page classes or Page Objects.

Web page classes or Page Objects containing web elements need to be initialized using Page Factory before the web element variables can be used. This can be done simply through the use of *initElements* function on PageFactory:

```
LoginPage page = new LoginPage(driver);
PageFactory.initElements(driver, page);
```

Or, even simpler:

```
LoginPage page = PageFactory.intElements(driver,LoginPage.class)
```

Or, inside the web page class constructor:

```
public LoginPage(WebDriver driver) {
    this.driver = driver;
    PageFactory.initElements(driver, this);
}
```

Page Factory will initialize every *WebElement* variable with a reference to a corresponding element on the actual web page based on configured “locators”. This is done through the use of *@FindBy* annotations. With this annotation, we can define a strategy for looking up the element, along with the necessary information for identifying it:

```
@FindBy(how=How.NAME, using="username")
private WebElement user_name;
```

Every time a method is called on this *WebElement* variable, the driver will first find it on the current page and then simulate the interaction. In case we are working with a simple page, we know that we will find the element on the page every time we look for it, and we also know that we will eventually navigate away from this page and not return to it, we can cache the looked up field by using another simple annotation:

```
@FindBy(how=How.NAME, using="username")
@CacheLookup
private WebElement user_name;
```

This entire definition of the *WebElement* variable can be replaced with its much more concise form:

```
@FindBy(name="username")
private WebElement user_name;
```

The *@FindBy* annotation supports a handful of other strategies that make things a bit easier:

```
id, name, className, css, tagName, linkText, partialLinkText, xpath
@FindBy(id="username")
private WebElement user_name;
```

```
@FindBy(name="password")
private WebElement user_password;
```

```
@FindBy(className="h3")
private WebElement label;
```

```
@FindBy(css="#content")
private WebElement text;
```

Once initialized, these *WebElement* variables can then be used to interact with the corresponding elements on the page. The following code will, for example:

```
user_password.sendKeys(password);
```

... send the given sequence of keystrokes to the password field on the page, and it is equivalent to:

```
driver.findElement(By.name("user_password")).sendKeys(password);
```

Moving on, you will often come across situations where you need to find a list of elements on a page, and that is when *@FindBy* comes in handy:

```
@FindBy(@FindBy(css="div[class='yt-lockup-tile yt-lockup-video']"))  
private List<WebElement> videoElements;
```

The above code will find all the *div* elements having two class names “yt-lockup-tile” and “yt-lockup-video”. We can simplify this even more by replacing it with the following:

```
@FindBy(how=How.CSS, using="div[class='yt-lockup-tile yt-lockup-video']")  
private List<WebElement> videoElements;
```

Additionally, you can use *@FindAll* with multiple *@FindBy* annotations to look for elements that match any of the given locators:

```
@FindAll({@FindBy(how=How.ID, using="username"),  
          @FindBy(className="username-field")})  
private WebElement user_name;
```

Now that we can represent web pages as Java classes and use Page Factory to initialize *WebElement* variables easily, it is time we see how we can write simple Selenium tests using PO and PF patterns.

Simple Test Automation Project in Java

For our simple project let’s automate developer sign up for Toptal. To do that, we need to automate the following steps:

- Visit www.toptal.com
- Click on the “Apply As A Developer” button
- On Portal Page first check if it’s opened
- Click on the “Join Toptal” button
- Fill out the form
- Submit the form by clicking on “Join Toptal” button

Setting Up a Project

- Download and install [Java JDK](#)
- Download and install Eclipse
- Create a new Maven project
- Add required dependencies like Selenium-Java, Testng, WebDriverManager etc Maven in your project POM file

How does it work?

Page Object Pattern is a pattern that displays user interface as a class. In addition to user interface, functionality of the page is also described in this class. This provides a bridge between page and test.



Why should it be used?

Here are the main advantages of Page Object Pattern using:

- Simple and clear tests.
- Good support of tests, because everything is stored in one place.
- Easy creation of new tests. In fact, tests can be created by a person not knowing the features of automation tools.
- Simple example of using Page Object Pattern and PageFactory in Selenium WebDriver.

Why use PageFactory?

For example, in the above classes, we used the following description of the elements on the page:

```
@FindBy(how = How.NAME, using = "text")
private WebElement helloText;
```

But as we use PageFactory, code can be written as follows:

```
private WebElement text;
```

where text will refer to an element with the attribute name = text

What's going on?

When we initialize the page using PageFactory, then, if annotation FindBy is not specified, PageFactory searches for elements on the page which name or id attributes match the name of WebElement. As a result, we actually do not have to worry about searching for elements on the page. If the element will appear on the page, so when we turn to it, it will be initialized.

Also, if we know that element is always present on the page, it is best to use the following declaration:

```
@FindBy(how = How.NAME, using = "text")
@CacheLookup private WebElement helloText;
```

If we don't do it, then every time when we turn to our element, WebDriver will check if the element is present on the page.

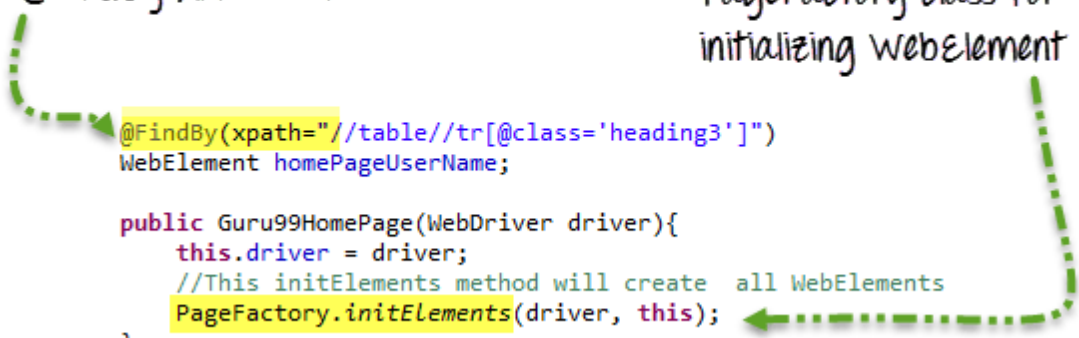
What is Page Factory?

Page Factory is an inbuilt page object model concept for Selenium WebDriver but it is very optimized.

Here as well we follow the concept of separation of Page Object repository and Test methods. Additionally with the help of PageFactory class we use annotations **@FindBy** to find WebElement. We use `initElements` method to initialize web elements

WebElements are identified by
@FindBy Annotation

Static `initElements` method of
PageFactory class for
initializing WebElement



```
@FindBy(xpath="//table//tr[@class='heading3']")
WebElement homePageUserName;

public Guru99HomePage(WebDriver driver){
    this.driver = driver;
    //This initElements method will create all WebElements
    PageFactory.initElements(driver, this);
}
```

@FindBy can accept **tagName**, **partialLinkText**, **name**, **linkText**, **id**, **css**, **className**, **xpath** as attributes.