Rex Jones II

(Transcript) Create Page Object Model

Video Playlist https://www.youtube.com/playlist?list=PLfp-cJ6BH8u_CynFLk3yzd8Kl1naC0a2T

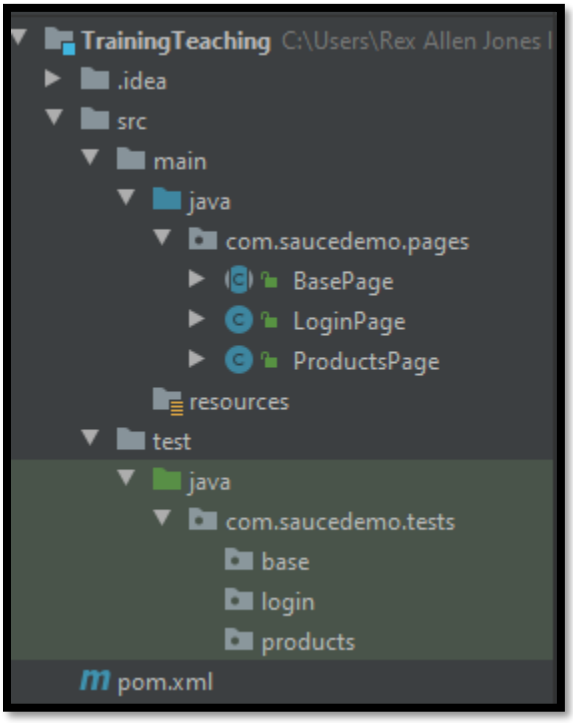
When creating a Page Object Model, I prefer to start with a Base Page. The purpose of a Base Page is to have a class with common actions that can be used across different Page Objects. Recall from the introduction, a Page Object is a class with variables and methods. The variables define elements of the page while methods are implemented as interactions with those elements.

In this video, I will demo step by step, how to create a Base Page and how to create 2 Page Objects. The Page Objects will be LoginPage and ProductsPage. My plan is to explain each concept and not just show you the concepts.

If you are interested in more videos, you can subscribe to my YouTube channel and click the bell icon. You can also follow me on Twitter, connect with me on LinkedIn and Facebook. After this video, I will create a transcript and place the transcript and code on GitHub.

BasePage

Here's the layout of our Page Object Model. We have 2 folders. One for pages and one for tests. The Base Page and Page Objects are placed in the com.saucedemo.pages folder then I placed the Base Test and Test Scripts in the com.saucedemo.tests folder. The url of our application is saucedemo.com.



Let's start the Page Object Model with an abstraction BasePage class: public abstract class BasePage { }. Abstract classes cannot be instantiated. Therefore, I am making the BasePage abstract so an object is not created from this class. With Inheritance, the role of this BasePage is to operate as a parent class and each Page class operates as the child class.

Since the BasePage is a parent class, it should contain class members that's also available on other pages in our application. Some variables that's available on other pages are page url, page title, and an object that waits for an element. When it comes to the driver, protected WebDriver driver; we can take more than 1 approach. At least 3 approaches. One approach; we can take is inheriting WebDriver throughout all of our Page classes: from the beginning of our Page class to the end of our Page class then instantiate the object in our Base Test.

The 2nd approach is to declare WebDriver as public static and make sure the driver always retains the last reference then add an if statement to check if the driver is null. The first 2 approaches is better for the DRY principle. However, in this session, I am going with a different approach and create a constructor then pass in WebDriver as an argument.

```
public BasePage(WebDriver driver) {
  this.driver = driver;

}
```

```
public abstract class BasePage {
  protected WebDriver driver;

  public BasePage (WebDriver driver) {
    this.driver=driver;
  }
```

The driver will be used to access the browser and locate elements. findElement is the Selenium method we use to locate elements so let's write,
protected WebElement find(By locator) {
  return driver.findElement(locator);

}

```
protected WebElement find (By locator) {
  return driver.findElement(locator);
}
```

This will be the only place in my Page Object Model that you will see driver.findElement. I will never write it again. We can reuse this find method in our BasePage and all of our Page classes.

Typing data and clicking an element are more generic methods that is common to all pages in our class. So, let's write

protected void type (String text, By locator) { }
We pass in String for the data our Test Script will type. The By locator is a parameter for locating an element. Look how the previous method reduces code duplication. I will not rewrite driver.findElement but only write find(locator).clear(). If there is data in a field then clear the data and write find(locator).sendKeys(text). That's smooth and that's all we write to type data. I will use this same process for the next method.

```
protected void type (String text, By locator) {
  find(locator).clear();
  find(locator).sendKeys(text);
}
```

The next method is protected void click(By locator) {
  find(locator).click();
}

```
protected void click (By locator) {
  find(locator).click();
}
```

That's all we need to click an element. This same click method will click a button, click a link, and click any other element that can be clicked on a web page. Let's write 1 more method.

How about we check if an element is displayed?
protected Boolean isDisplayed (By locator) {
  return find(locator).isDisplayed()

} Know what, let's wrap this return statement in a try/catch block.
try {
  return find(locator).isDisplayed();
}
catch (NoSuchElementException exc) {
  return false;
}

```
protected Boolean isDisplayed (By locator) {
  try {
    return find(locator).isDisplayed();
  }
  catch (NoSuchElementException exc) {
    return false;
  }
}
```

If an element is not on a page then our code will catch the NoSuchElementException and return false. I added Boolean as the return type because isDisplayed returns true or false. These other 2 methods have void as the return type because they do not return a value. This find method has WebElement as the return type because findElement returns a WebElement.

The objective of this BasePage is to allow all of the Page classes to inherit the base variables and methods. Next is the Login Page.

LoginPage

For convention, it is best to have a class name that represents a page in our application. That's why I have LoginPage as the class name. In the application, we have some elements on the Login page: a username field, password field, login button, and error message. Our page class will have properties. We can also say fields to represent these 4 elements. The LoginPage extends our BasePage and thanks to inheritance we see the methods created in the BasePage: isDisplayed, click, type, and find.

Those 4 methods will perform actions on the private By usernameField;, private By passwordField;, private By loginButton;, and private By errorMessage;

```
public class LoginPage extends BasePage {
  private By usernameField;
  private By passwordField;
  private By loginButton;
  private By errorMessage;
}
```

With Encapsulation, the fields are private so only methods in this LoginPage class have access to the fields. Remember, By is a class for locating elements and the values are going to be locators for each element.

Let's go back to the application and get their values. Right click, Inspect username. We see the form tag has username, password, and Login button. All of the elements have an id attribute. Username has a value of user-name. Okay – cool. Password has a value of password. The Login button has a value of login-button. Inspect the error message and it does not contain an id attribute but the closest id is located in a parent tag. Let's use CSS Selector for the error message. CTRL + F and write #login_button_container h3. Bingo 1 of 1, Copy the value. You can watch my Playlist Customize XPath & CSS Values if you would like to know how I created this value.

The value for usernameField = By. and we see the Selenium locators: className, cssSelector, id, linkText, name, partialLinkText, tagName, and xpath. We are going to use id("user-name"); passwordField = By.id("password"); loginButton = By.className("login-button"); and errorMessage = By.cssSelector paste ("login_button_container h3")

```
public class LoginPage extends BasePage {
  private By usernameField = By.id("user-name");
  private By passwordField = By.id("password");
  private By loginButton = By.id("login-button");
  private By errorMessage = By.cssSelector("#login_button_container h3");
```

The constructor is public LoginPage(WebDriver driver) {
super(driver); We are calling super then passing in the driver to load the constructor from our BasePage. The constructor is created to bring in the WebDriver so the driver can access the browser.

```
public LoginPage (WebDriver driver) {
  super(driver);
}
```

We have our constructor and fields. Now, let's work on the methods which will perform actions on these elements. There are 4 types of methods we can use in our Page Object Model. We have getter and setter methods that comes with encapsulation. Also, we have transition methods and convenience methods. The convenience methods can also be called helper methods. The getter method gets the state of our application and setter methods helps us to create a test that interacts with our application. A transition method is important when our application changes to a different page. It may change when

we click a button or click a link. A convenience method is created when combining more than 1 method into a single method. I'm going to demo all 4.

First, we are going to use the setter method by setting the username. public void setUsername (String username) {
type the (username, in the usernameField) type came from the BasePage, username is the parameter and usernameField is the property variable. That's all we write to type for the username. Finish – Done Deal

```
public void setUsername (String username) {
  type(username, usernameField);
}
```

} Same with Password. We write public void setPassword(String password) {
type(password, in the passwordField)

```
public void setPassword (String password) {
  type(password, passwordField);
}
```

} Next is the transition method public ProductsPage clickLoginButton { }. Do you see why it's called a transition method? We are going to set the username, set the password, and click the Login button then the LoginPage transitions to the ProductsPage. We click(loginButton) / return new ProductsPage pass in the (driver);.

```
public ProductsPage clickLoginButton () {
  click(loginButton);
  return new ProductsPage(driver);
}
```

In this scenario, we return the page we transition to and not use the keyword void. An error shows up because we have a constructor that's needed in the ProductsPage.

ProductsPage extends BasePage { }
public ProductsPage (WebDriver driver) {super(driver);} Save. Go back to LoginPage and now the error goes away.

```
public class ProductsPage extends BasePage {

  public ProductsPage (WebDriver driver) {super(driver);}
}
```

Let's create a convenience method so our test also has an opportunity to call one method for logging into the application: setUsername, setPassword, and clickLoginButton. This method will be
public ProductsPage loginWith (String username, String password) { }
  setUsername(username);
  setPassword(password); then
  return clickLoginButton();

```
public ProductsPage loginWith (String username, String password)
  setUsername(username);
  setPassword(password);
  return clickLoginButton();
}
```

Someone may wonder, why do we have an individual method for username, password and clicking the LoginButton plus a convenience method? That convenience method and 3 individual methods defeat the purpose of code duplication. You are right but this is an exception. It's good to have an individual method because your test may include a negative test. For example, you might want to enter a username without a password, click the Login button then confirm the application returns an error message. Let's add a getter method for the error message.
public String getErrorMessage() { }
  return find(errorMessage).getText();

```
public String getErrorMessage () {
  return find(errorMessage).getText();
}
```

We have String return type because the method getText returns a String. We are finished with the Page Object for LoginPage. Now, let's create a Page Object for the ProductsPage.

ProductsPage

Go back to Swag Labs and inspect the Products label. Class is product_label. Also inspect the backpack title and we are going to use id item_4_title_link. Click the backpack then inspect the Add To Cart button. Let's test this class value using CSS Selector and write .btn_inventory. Bingo. Okay, we got our values. Let's go to the ProductsPage class.

We already have our constructor. So, let's add those 3 properties and their values.
private By productLabel = By.className("product_label"); / private By backpack = I think it's By.id("item_4_title_link") / private By addToCartButton = By.className("btn_inventory");

```
public class ProductsPage extends BasePage {

    private By productLabel = By.className("product_label");
    private By backpack = By.id("item_4_title_link");
    private By addToCartButton = By.className("btn_inventory");
```

The first method is to check whether the Products label is displayed.
public Boolean isProductLabelDisplayed() { }
return isDisplayed(productLabel);

```
public Boolean isProductLabelDisplayed () {
  return isDisplayed(productLabel);
}
```

Next is add backpack. public void addBackPack () {
First, we want to find the (backpack) then click(); the backpack. / After clicking the backpack, we click the (addToCartButton);

}

```
public void addBackPack () {
  find(backpack).click();
  click(addToCartButton);
}
```

We are going to also going to get the button name after adding backpack to the cart.
public String getButtonName () { }
  return find(addToCartButton).getText();

```
public String getButtonName () {
  return find(addToCartButton).getText();
}
```

I have a quote from Simon Stewart, the creator of Selenium WebDriver, "If you have WebDriver APIs in your test methods, You're Doing It Wrong".

> *If you have WebDriver APIs in your test methods, You're Doing It Wrong.*
>
> *Simon Stewart.*

The WebDriver API commands are methods like click, clear, getText, and isDisplayed. These commands belong in our Page Object class and not in our Test Scripts. However, when it comes to assertions, I prefer to include them in my Test Scripts. There is not a rule that say assertions belong in our Page Object class or Test Scripts but I think it's best to keep them evenhanded neutral. Here's the difference, in our Page Object, the assertion will have 1 outcome: either pass or fail. But, in our Test Script, the assertion has balance so we can use it for a negative test and a positive test.

I have another quote. You probably noticed I did not speak about PageFactory. That's because Simon Stewart who is also the creator of PageFactory said PageFactory is not his best work.

> The PageFactory design ain't my best work. I'd recommend using it for inspiration for something better. Or just dealing with boilerplate.

In a different message, he said the community took the wrong impression with PageFactory.

> A couple years ago I asked Simon Stewart about page factory. I'll list a few points from that conversation:
>
> Simon wrote the initial page factory code in a couple hours.
>
> Page factory is an **example** of what is possible with WebDriver as library code.
>
> The community took the wrong impression with Page Factory and **extended** page factory, rather than **replacing** it with something better(though some newer stuff has come out recently). This is disappointing to Simon.

If you want to watch the Selenium Conference from 2017, he also said some more things about PageFactory.

Next, we are going to create a BaseTest then execute our first test using the Page Object Model and I'll see you in the next session.

Social Media Contact

- ✔ YouTube https://www.youtube.com/c/RexJonesII/videos
- ✔ Facebook http://facebook.com/JonesRexII
- ✔ Twitter https://twitter.com/RexJonesII
- ✔ GitHub https://github.com/RexJonesII/Free-Videos
- ✔ LinkedIn https://www.linkedin.com/in/rexjones34/