Rex Jones II

(Transcript) Create BaseTest & Execute 1st Test Script

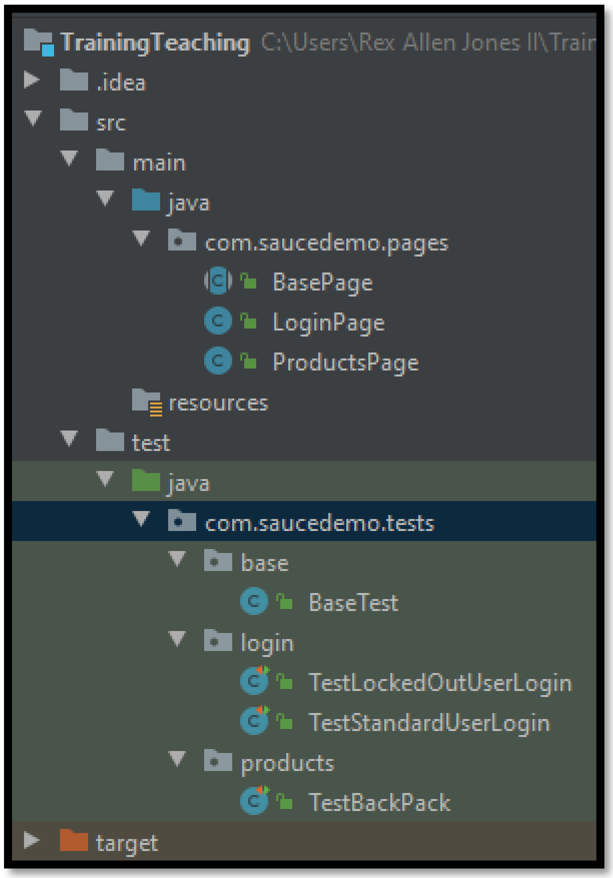
Video Playlist https://www.youtube.com/playlist?list=PLfp-cJ6BH8u_CynFLk3yzd8Kl1naC0a2T

Create Test Using Page Object Model

BaseTest

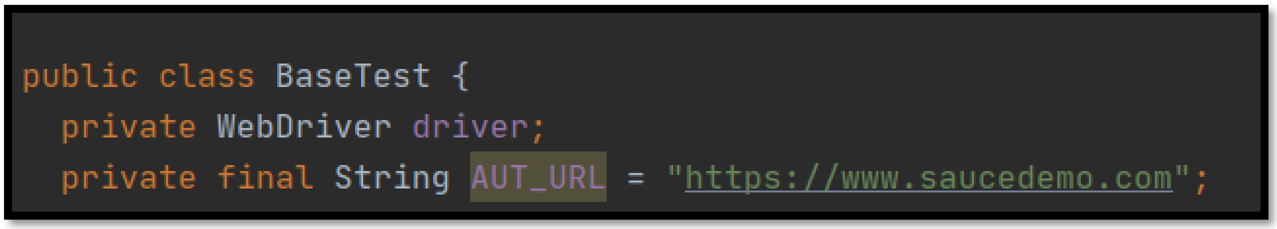
We create a test using the Page Object Model by starting with a BaseTest. One of the reasons for creating a BaseTest is to set up WebDriver and to tear down WebDriver.

Here's the BaseTest and 3 Test Scripts. 2 of the Test Scripts is for the Login functionality and 1 Test Script is for the Products functionality.



```
TrainingTeaching  C:\Users\Rex Allen Jones II\Trair
▶ .idea
▼ src
    ▼ main
        ▼ java
            ▼ com.saucedemo.pages
                BasePage
                LoginPage
                ProductsPage
        resources
    ▼ test
        ▼ java
            ▼ com.saucedemo.tests
                ▼ base
                    BaseTest
                ▼ login
                    TestLockedOutUserLogin
                    TestStandardUserLogin
                ▼ products
                    TestBackPack
▶ target
```

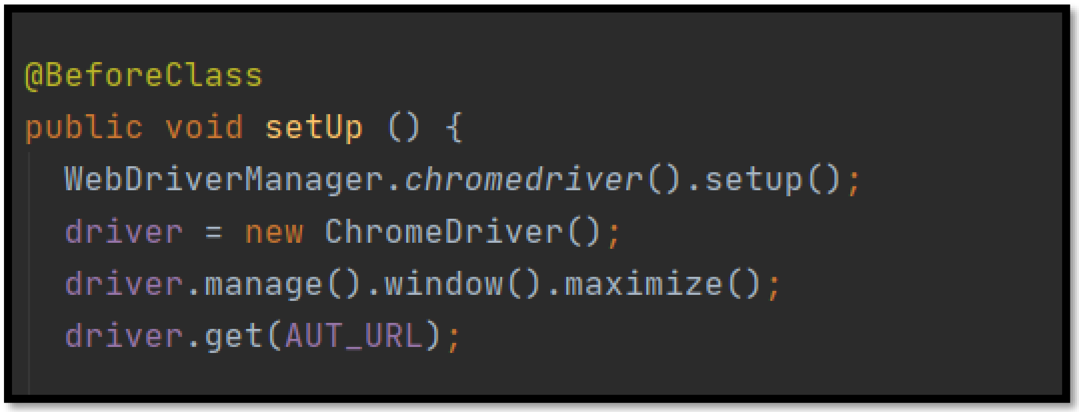
For the BaseTest, the variables are private WebDriver driver; and the url is private final String Application Under Test AUT_URL = "https://www.saucedemo.com";



```java
public class BaseTest {
  private WebDriver driver;
  private final String AUT_URL = "https://www.saucedemo.com";
```

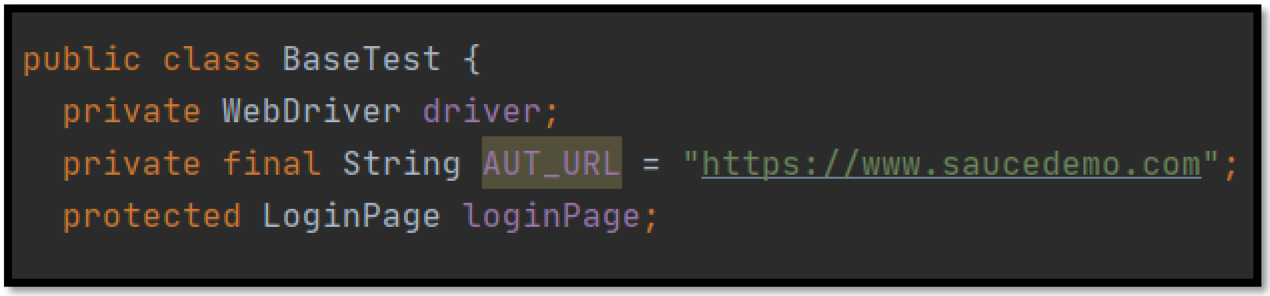
Some fields in our Page Object Model may include final and may not include final. It depends on how you want to create your design pattern. Writing final is a way of making sure the value does not change. In this case, the value for AUT_URL cannot be changed.

Let's start with the annotations for TestNG. @BeforeClass / public void setup () { }
We can use System.setProperty to set the arguments for webdriver and a path to the executable file. However, I'm going to write WebDriverManager.chromedriver().setup(). You can watch video 142 if you want more info about WebDriverManager. It's a library for automating the management of our drivers. Initialize the driver by writing driver = new ChromeDriver (); / Maximize the window driver.manage().window().maximize() / load our application by writing driver.get(AUT_URL);



```java
@BeforeClass
public void setUp () {
  WebDriverManager.chromedriver().setup();
  driver = new ChromeDriver();
  driver.manage().window().maximize();
  driver.get(AUT_URL);
```

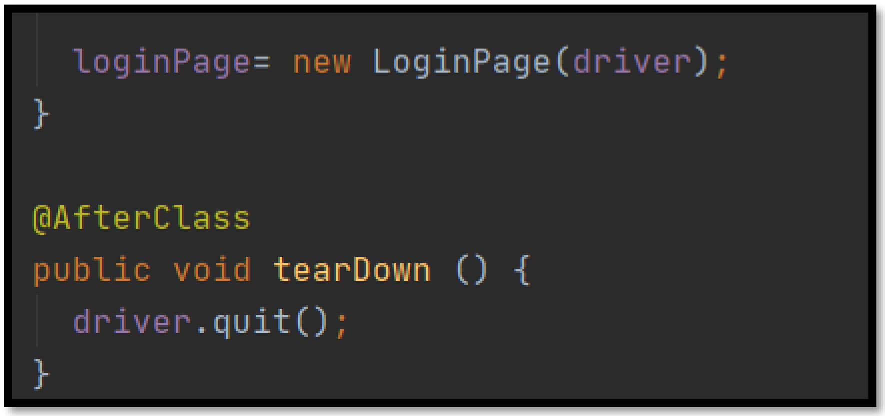
After loading the application, let's implement a way for our 3 Test Scripts to access the LoginPage. protected LoginPage loginPage; I marked this field protected so each Test Script has access to the LoginPage after inheriting the BaseTest.



```java
public class BaseTest {
  private WebDriver driver;
  private final String AUT_URL = "https://www.saucedemo.com";
  protected LoginPage loginPage;
```

Create an instance by writing loginPage = new LoginPage(driver);

Next is the annotation for @AfterClass. This configuration will tear down our test automatically after executing the Test Script. public void tearDown () { }
  driver.quit();



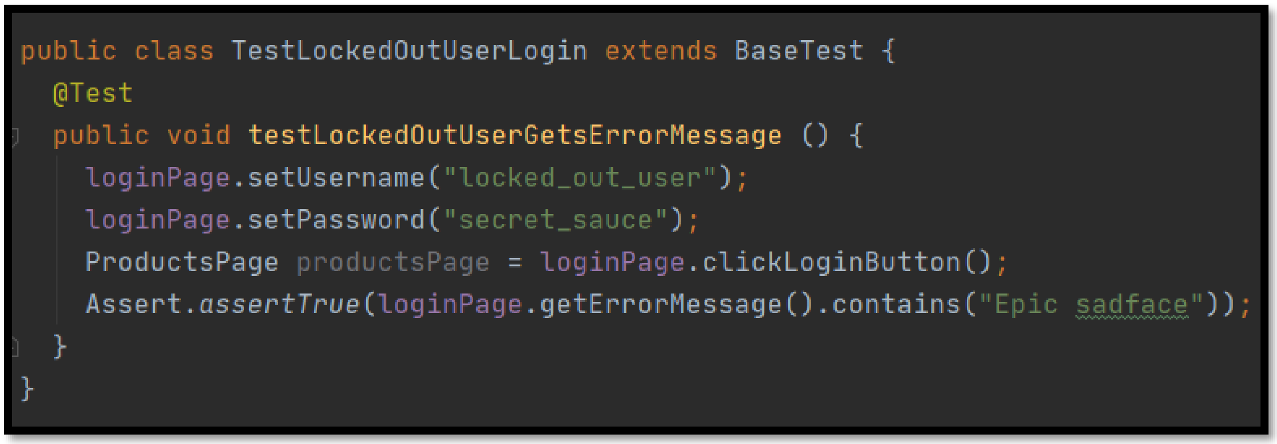
```
    loginPage= new LoginPage(driver);
}

@AfterClass
public void tearDown () {
    driver.quit();
}
```

We can write more in this BaseTest but that's all I'm going to write for now. The benefit of having a BaseTest is to avoid setting up and tearing down every Test. Now, let's move on to creating our Login test.

Login Test

For the Login LockedOutUserTest, we inherit the BaseTest by writing extends BaseTest { }
The annotation is @Test / public void testLockedOutUserGetsErrorMessage () {. Now, we write the logic for testing the locked out user.
From the loginPage., we the setUsername to ("locked_out_user"); / Also, the loginPage. will setPassword to ("secret_sauce") / then click the button on the loginPage.cl. Do you see how the intellisense shows the method clickLoginButton() and ProductsPage? This lets us know that calling clickLoginButton has a transition which returns the ProductsPage. The application makes a transition by changing from the LoginPage to the Products page. Therefore, we assign clickLoginButton to ProductsPage productsPage =

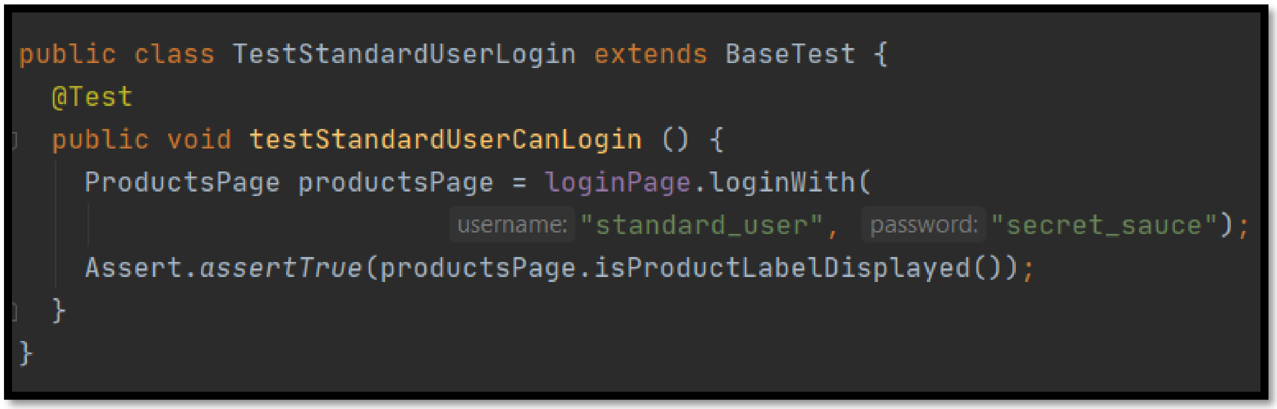
Thankfully, we did not add a failed assertion to our Page Object class. In this Test Script, we expect an error because the user is locked out and we want the test to pass. Go back to the application. The username is locked_out_user, password is secret_sauce and we see the message contains Epic sadface. Go back to our Test Script and write Assert.assertTrue(loginPage.getErrorMessage(). Make sure the message contains("Epic sadface"));



```
public class TestLockedOutUserLogin extends BaseTest {
  @Test
  public void testLockedOutUserGetsErrorMessage () {
    loginPage.setUsername("locked_out_user");
    loginPage.setPassword("secret_sauce");
    ProductsPage productsPage = loginPage.clickLoginButton();
    Assert.assertTrue(loginPage.getErrorMessage().contains("Epic sadface"));
  }
}
```

Let’s Run. We see the test Passed. Next is to test a user can log into the application then verify the Products label is displayed. You see, I have each test in separate classes but they can easily be in the same class. Either way is okay. On a project, I would group all test related to the Locked Out user in one class and all test related to a Standard User in a different class because we may have more than 1 Test Script for each user type.

First, we extend BaseTest. @Test / public void testStandardUserCanLogin () { } This time, let’s use the convenience method: On the loginPage. we will loginWith(“standard_user”, “secret_sauce”); then transition to the ProductsPage productsPage = . Now, verify the Products label is displayed. Assert.assertTrue(productsPage.isProductsLabelDisplayed());



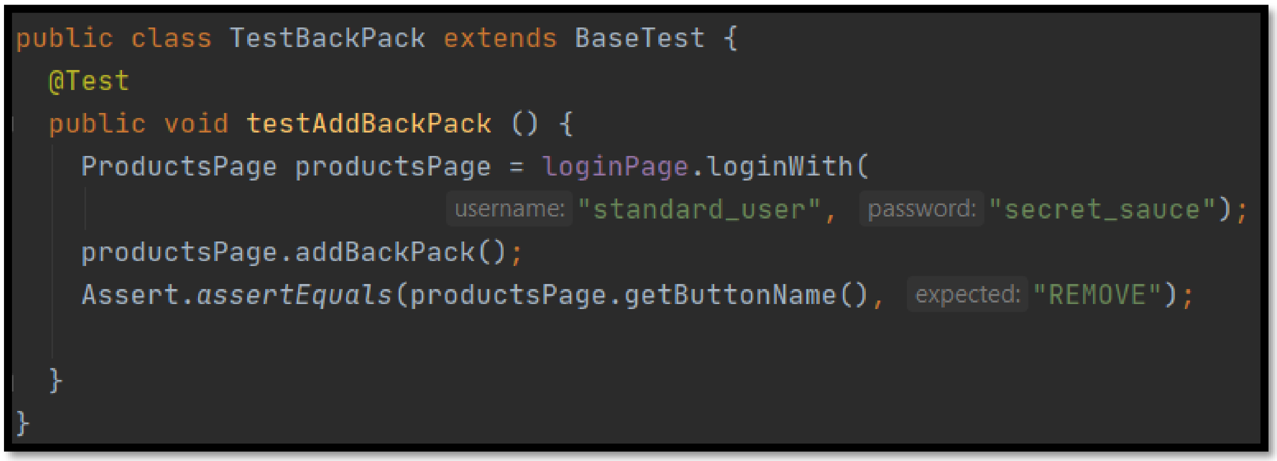
```
public class TestStandardUserLogin extends BaseTest {
  @Test
  public void testStandardUserCanLogin () {
    ProductsPage productsPage = loginPage.loginWith(
                         username: "standard_user",  password: "secret_sauce");
    Assert.assertTrue(productsPage.isProductLabelDisplayed());
  }
}
```

Let’s Run. The Test Passed. Let’s create one more test really quick for the Products page.

Products Test

Go to the TestBackPack class and it also extends BaseTest @Test / public void testAddBackPack () { }

On the loginPage.loginWith(“standard_user”, “secret_sauce”); assign to the ProductsPage productsPage =. At this point our test is on the productsPage. and the test will addBackPack(). After adding a backpack, we are going to verify the button name for Add to Cart changes to REMOVE. Assert.assertEquals(productsPage.getButtonName(), we expect the name to be “REMOVE”); in all caps.



```
public class TestBackPack extends BaseTest {
  @Test
  public void testAddBackPack () {
    ProductsPage productsPage = loginPage.loginWith(
                        username: "standard_user",  password: "secret_sauce");
    productsPage.addBackPack();
    Assert.assertEquals(productsPage.getButtonName(),  expected: "REMOVE");

  }
}
```

This time, let’s run all 3 tests at the same time. We see all 3 Test were a success. That’s it for creating a Base Test and creating 3 Test Scripts using the Page Object Model.

If you like this video, consider following me on Twitter and connecting with me on LinkedIn and Facebook. You can also subscribe to my channel and click the bell icon. I have more videos coming. The documentation and code be uploaded to GitHub.

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/