Selenium with Java

A Beginner's Guide

Web Browser Automation for Testing using Selenium with Java

PALLAVI SHARMA

bpb

Selenium with Java – A Beginner's Guide

Web Browser Automation for Testing Using Selenium with Java

Pallavi Sharma

www.bpbonline.com

FIRST EDITION 2022

ISBN: 978-93-91392-680

LIMITS OF LIABILITY AND DISCLAIMER OF WARRANTY

The information contained in this book is true to correct and the best of author's and publisher's knowledge. The author has made every effort to ensure the accuracy of these publications, but publisher cannot be held responsible for any loss or damage arising from any information in this book.

All trademarks referred to in the book are acknowledged as properties of their respective owners but BPB Publications cannot guarantee the accuracy of this information.

Distributors:

BPB PUBLICATIONS
20, Ansari Road, Darya Ganj
New Delhi-110002
Ph: 23254990/23254991

DECCAN AGENCIES
4-3-329, Bank Street,
Hyderabad-500195
Ph: 24756967/24756400

MICRO MEDIA
Shop No. 5, Mahendra Chambers,
150 DN Rd. Next to Capital Cinema,
V.T. (C.S.T.) Station, MUMBAI-400 001
Ph: 22078296/22078297

BPB BOOK CENTRE
376 Old Lajpat Rai Market,
Delhi-110006
Ph: 23861747

To View Complete BPB Publications Catalogue Scan the QR Code:

Published by Manish Jain for BPB Publications, 20 Ansari Road, Darya Ganj, New Delhi-110002 and Printed by him at Repro India Ltd, Mumbai

Dedicated to

Neeal *and* ***Maisha***

&

Dr Neelaksh *and* ***Dr Manju***

My kids who continue to make me see the wonders of life.
And my parents because of them I am.

About the Author

Pallavi is a multiskilled professional who dons many hats. She started her career as a Research Scholar under Dr Naren Ramakrishnan at Virginia Tech where she worked on Data mining project for bioinformatics.

At Crestech Software Systems she was part of the Centre of Excellence Team under the CEO, which was responsible for providing cutting edge solutions to the delivery team. She has worked on tools like Selenium, Watir, RFT, JMeter, LoadRunner, Rational Robot and similar tools.

She managed the team of developers, testers, support function, presales, marketing and technical writing for the inhouse product OpKey. She also managed the company patent and trademark division.

In 2014 she founded 5 Elements Learning, where she coach people on Selenium, Appium, Cucumber ecosystem, which includes tech stack for test automation project. She coach using programming languages like Java, Python, CSharp and Ruby. She has taught more than 1000 people from across the globe. She is also an instructor at Udemy, with more than 20K followers, and have many paid and free courses hosted for testers.

She continues to be an active participant for various global conferences like Selenium Summit, APISummit Delhi Software Testing Conference, Selenium India Conference, Global Testing Retreats. She has participated as Jury, Organizing Committee Member, and as Speaker.

She grateful to be mentored and taught by Michael Bolton and Maaret Pyhäjärvi on the concepts of Exploratory Testing.

She is a Steering Committee member with Agile Testing Alliance, since 2021, and a long time advisory board member. Her work involves designing programs on various global certifications on automation, organizing conferences, and handle the social media presence of Agile Testing Alliance, with a team.

She is a published author with BPB publications. Her first book, Selenium with Python A beginner's book is among the top 10 books on amazon for the category. Her second book Selenium 4 – A Quick and Practical Guide is available on LeanPub.

She believes in certifying her learnings, and holds various certifications like ISTQB, CP-SAT, CP-WST, CP-DOF, Cucumber School Completion, SpecFlow among others.

She is an an avid reader, traveller, and a writer at heart. She enjoys fiction, non fiction books, and writing poetry, short stories on her life experiences. She shares her work on Medium, and could be found at - https://medium.com/@pallavimuse.

She is a firm believer in larger good, and likes to live by example. Organization like eVidyaloka, People for Animal, WildLife SOS, and Jabarkhet forest reserve is where she dedicates her resources.

About the Reviewer

Jaya Saini has 6.5 years of experience in Automation testing and API testing using tools like Selenium, TestNg, Robot Framework, Postman, with programming languages like Java, JavaScript Python, and more. Jaya Saini pursued B.Tech in Computer science and engineering from Rajasthan Technical University, Alwar. She has worked with companies like Sopra Banking and Thinksys software. She is currently working as Automation acting Test lead in Thinksys software, Noida.

Acknowledgement

There are a few people I want to thank for the continued and ongoing support they have given me during the writing of this book. First and foremost, I would like to thank the creator of Selenium Jason Huggins, and creator of Selenium Webdriver Simon Stewart for their creation. The people behind the Selenium project and community, who continue to work for the project and make it available for the world.

I would like to thank my husband Ravinder for his unconditional love, and silly jokes. My parents for setting example to live life by giving others. My children Neeal and Maisha who remind me to be a child at heart always. My sister Dr Mahak for her endless patience, and our conversations about life, environments, politics, humour and of course relatives.

I would like to thank Manish Jain, for giving me the opportunity with BPB Publications, and help me achieve my dream of becoming a writer. The editor and technical team at BPB Publications for their continuous support.

Finally my immense thanks to the various form Almighty takes, and allows me to experience its endless wisdom, as I live.

Preface

Selenium automates the browser, what we do with that power is up to us. This is the message which we get the first time we visit the official website of Selenium. It is the most popular open source solution to automate the web browser, and is freely available to anyone and everyone to use. People who are into web application testing, have been using Selenium for many years now. It helps automate mundane tasks, achieve efficiency, and accuracy which in the end improves the test process. Selenium power lies in the fact that it can be implemented using any programming language, can automate any web browser which is W3C compliant, and can run on any operating system.

Selenium project and WebDriver merged to make Selenium WebDriver, which is now a W3C standard. This means any web browser which is W3C compliant can be automated using Selenium. Various versions of Selenium are used across projects both small and big. The version of Selenium which we have used in this book is 3.141.59. The implementation of Selenium in this book is done using Java as a programming language.

In this book we have tried to cover the basic classes and interfaces of Selenium which help us automate the browser and browser elements. We have covered the importance of data and object management outside the test scripts, and also covered the implementation of the same. We talk about the popular unit test framework TestNG, and understand its various dimensions which we can use in the our test automation projects. We in the end learn about how we can set our tests for execution in parallel using Selenium Grid for different test environments.

Over the 12 chapters in this book, you will learn the following:

Chapter 1 – Learn about what is Selenium. Get introduced to the three projects under Selenium. The Selenium IDE, Selenium WebDriver and Selenium Grid.

Chapter 2 – Learn about how we set up our system environment to write test scripts, and automate browser using Selenium. Learn about the different applications and scenarios we will be automating to understand Selenium.

Chapter 3 – Learn about the three pillars of Selenium Java client library, WebDriver which automates the web browser. WebElement, which handles automation of

html elements on page. And the By class which help us create locators to locate web element on page.

Chapter 4 – Understand the importance of synchronization, and how to implement in the test scripts.

Chapter 5 – Understand how to handle and automate different html elements like form elements, table, drop down

Chapter 6 - Understand how to handle and automate web elements like frame, alerts and window.

Chapter 7 – Learn about some advance concepts like handling of keyboard and mouse events using action class. Capturing screenshot. And manage browser drivers using webdrivermanager class.

Chapter 8 - Learn about what is TestNG, understand its various features which help in test automation projects.

Chapter 9 – Learn the importance of object management outside the test scripts. And understand how to implement it using the design pattern page object model.

Chapter 10 – Learn about the importance of managing data outside the test scripts. Also learn about implementing reading of data from external data files like csv, and excel.

Chapter 11 – Get introduced to a project life cycle management tool Maven. Understand its importance in managing project which uses open source solutions.

Chapter 12 – Learn about Selenium Grid, the component of Selenium which allows test suite execution to happen in parallel across different environments.

Code Bundle and Coloured Images

Please follow the link to download the
Code Bundle and the ***Coloured Images*** of the book:

https://rebrand.ly/9dd688

The code bundle for the book is also hosted on GitHub at **https://github.com/bpbpublications/Selenium-with-Java-A-Beginner-s-Guide**. In case there's an update to the code, it will be updated on the existing GitHub repository.

We have code bundles from our rich catalogue of books and videos available at **https://github.com/bpbpublications**. Check them out!

Errata

We take immense pride in our work at BPB Publications and follow best practices to ensure the accuracy of our content to provide with an indulging reading experience to our subscribers. Our readers are our mirrors, and we use their inputs to reflect and improve upon human errors, if any, that may have occurred during the publishing processes involved. To let us maintain the quality and help us reach out to any readers who might be having difficulties due to any unforeseen errors, please write to us at :

errata@bpbonline.com

Your support, suggestions and feedbacks are highly appreciated by the BPB Publications' Family.

Piracy

If you come across any illegal copies of our works in any form on the internet, we would be grateful if you would provide us with the location address or website name. Please contact us at **business@bpbonline.com** with a link to the material.

If you are interested in becoming an author

If there is a topic that you have expertise in, and you are interested in either writing or contributing to a book, please visit **www.bpbonline.com**. We have worked with thousands of developers and tech professionals, just like you, to help them share their insights with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

Reviews

Please leave a review. Once you have read and used this book, why not leave a review on the site that you purchased it from? Potential readers can then see and use your unbiased opinion to make purchase decisions. We at BPB can understand what you think about our products, and our authors can see your feedback on their book. Thank you!

For more information about BPB, please visit **www.bpbonline.com**.

Table of Contents

Chapter 1
Introduction to Selenium

Selenium automates the browser; that is it. What you do with that is entirely up to you. One finds this information highlighted as soon as you open the main page of the selenium website, hosted at the URL—**http://www.selenium.dev**. Currently, the Selenium project consists of three main components—Selenium IDE, Selenium WebDriver, and Selenium Grid. In this book, we will be looking at selenium WebDriver and selenium grid usage using Java as a programming language.

Structure

In this chapter, we will discuss the following topics:

- What is Selenium
- Selenium main components
    - Selenium IDE
    - Selenium WebDriver
    - Selenium Grid
    - Why Selenium is popular
    - People behind the Selenium project

Objectives

After studying this unit, you will be able to understand:

- Understand the popularity of Selenium.
- Have a brief idea about Selenium as a tool to automate the Web browser.
- Know the different components of Selenium and where they are used.

What is Selenium?

Selenium took birth in the year 2004, the idea of it. It was at that time known as JavaScriptTestRunner and created by Jason Huggins while working in the organization ThoughtWorks. The tool was created to overcome the challenges faced by the existing set of solutions available than in the market, and since then, a significant number of people, organizations have contributed to this open-source project to make it what it is today. Simon Stewart showcased at the GTAC conference the WebDriver version of talking to the browsers, considered more refined than the JavaScript proxy server version adopted by Selenium, eventually leading these two concepts to get married. Selenium married WebDriver and became the Selenium WebDriver, which we all use as of now. The current stable release available at the time of the book being written is 3.141.59.

Why Selenium is popular

Selenium popularity is primarily due to the fact the three important features it has:

- It supports multiple programming languages
- It supports test execution in multiple operating systems
- It supports test execution on multiple browsers

This kind of rich feature set was rare and is still rare in many of the available commercial, free and open-source test automation solutions for Web applications. Another reason was the flexibility which the open-source tool provides. They inherently come up with the power of adjusting as per the requirements of your project instead of making you adjust within the tight boundaries of the tool. Also, the cost associated with procuring Selenium was nil. The tool was free to avail. One most important factor for Selenium to become widely popular was the strong user community behind it, which is still very active and can be found at—**https://groups.google.com/forum/#!forum/selenium-users**

Components of Selenium

Selenium currently has three major components which are meant to solve three important criteria as follows:

- Selenium IDE
- Selenium WebDriver
- Selenium Grid

The following table explains the use of every component:

Component	Use
Selenium IDE	It is basically used for recording and playback. The current version of selenium ide can be used with both Firefox and Chrome browsers. The current version of Selenium IDE is backed by the organization Applitools, and the most important contributor for its development and well keep is Dave Haeffner. More about him and Applitools here—**https://applitools.com/blog/team-dave-haeffner/**
Selenium WebDriver	This component talks to the browser as a real-time user/person will do whether the browser is in the local system or the remote system. To use this component, we will have to use the client libraries, available in different programming languages such as Java, Ruby, Perl, Python, C#, and so on. Our program script is written using these languages, which have commands we would want the browser to do. The associated WebDriver for the browser automates the browser, and we get our results. The current stable version is 3.141.59 as of writing this book.
Selenium Grid	This component allows the tests to be executed in parallel. It uses the selenium remote control component of Selenium, and we then set it up in two different modes, the hub mode and the node mode. This allows us to run the different tests in parallel in different nodes at the same time.

Table 1.1: *Use of Selenium components*

People behind Selenium

There are a lot of people behind Selenium; rather, it is a fine example of a tool that has been driven and maintained by a community rather than made popular by them. What started as a work project in 2004 in the organization ThoughtWorks by Jason Huggins was made open source to the world. A good history about its usage,

adaptation, and different avatars it took is available here **https://www.selenium.dev/history/.**

Selenium IDE was first created by Shinya Kasatani in Japan, this was later picked up by Mike Williams from ThoughtWorks, and now this project is supported by the organization Applitools, Dave Haeffner, and the team takes care of it.

Google started using Selenium grid internally and revealed this at the GTAC conference; this was done by Jennifer Bevan, who eventually became one of the contributors to the Selenium project. Haw-Bin Chai is the person behind the concept of locators UI-Elements used in Selenium. And we all now know about Simon Stewart, the one who came up with WebDriver, and is now the current founding father of Selenium as we know it.

Conclusion

In this chapter, we have been introduced to Selenium as a tool to automate the Web browser. We saw the different components associated with it and the different people associated with its development. We also understood the reason behind its immense popularity. In the upcoming chapter, we will be discussing the Selenium API in Java.

Questions

1. Is Selenium one tool or multiple tools combined under one name?
2. What is the current stable version of Selenium available?
3. What was Selenium known as when Jason Huggins created it while working in ThoughtWorks?
4. Selenium IDE works both on Chrome and Firefox. True or False?
5. Why is Selenium popular?

Chapter 2

Preparing System and Application Under Test

In this chapter, we will set up the IDE eclipse on our systems. We will be taking only the Windows system. After setting the eclipse as the script integrated development environment, we will set the Selenium jar files in the project so that when we write programs using Java as language, our environment is able to decipher them. Before we run the tests in the browsers, we also need to set the drivers for the respective browser, so in this chapter, we will learn that. In the upcoming chapters, we will also see how we can create a Maven project and use it to have the dependencies managed at run time. But for the time being, we will be taking the preceding mentioned approach.

In this chapter, we will also see the application we would be using to explore Selenium. We will look at some of the scenarios provided. Besides this, we will also explore a few other Web applications, which scenarios we will be using to understand the concepts of Selenium.

Structure

In this chapter, we will discuss the following topics:

- Setting right eclipse version on system
- Creating Java project

- Setting Selenium jars
- Setting browser drivers
- Walkthrough of the BPB application
- Knowing other applications for practice

Objectives

After studying this unit, you should be able to understand:

- Set up eclipse
- Create Java project
- How to add Selenium jars
- How to set the browsers drivers
- Walkthrough of the BPB application
- Understanding other applications

Setting eclipse

Before we download and set up eclipse on the system, we need to ensure we have Java installed. The version of Java JDK we require is Java JDK 8 version. To install it, we will need to visit the link—**https://www.oracle.com/java/technologies/javase-jdk8-downloads.html.** According to the OS we have, we will download the respective file available—**https://www.oracle.com/java/technologies/javase-jdk8-downloads.html**

Once downloaded, we will install and set up the Java JDK on our system. Please note it is important that we need to have Java available on our system before we proceed toward installing eclipse.

Eclipse is a famous IDE (integrated development environment), which is used to create projects using various programming languages. It supports Java, Python, Ruby, Ada, C++, Natural, and many more. It is an open-source IDE for professional developers, backed by a strong community. More details about it can be found here—**https://www.eclipse.org/eclipseide/**

We will be downloading the Eclipse Photon version to set up Selenium 3.141.59 jars. For this, please visit this URL—**https://www.eclipse.org/downloads/packages/release/photon/r**

Download the Eclipse IDE for Java Developers, depending upon the OS version you are using.

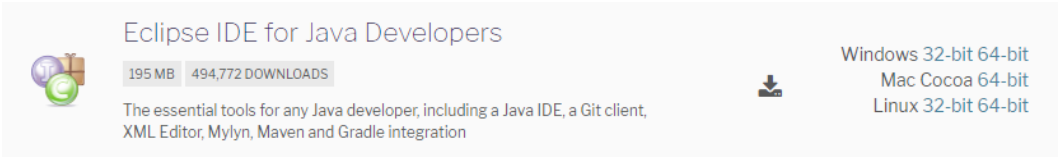


***Figure 2.1**: Eclipse version*

For example, if you are using Windows 64 bit, download that option. Once the **`.exe`** file is available, install eclipse on your system.

After eclipse is installed on your system, you should be able to see an icon of it on the desktop, double click on it to launch eclipse. It opens up a pop-up to ask you a place for **`Workspace`**—a place where you will be storing the eclipse projects.

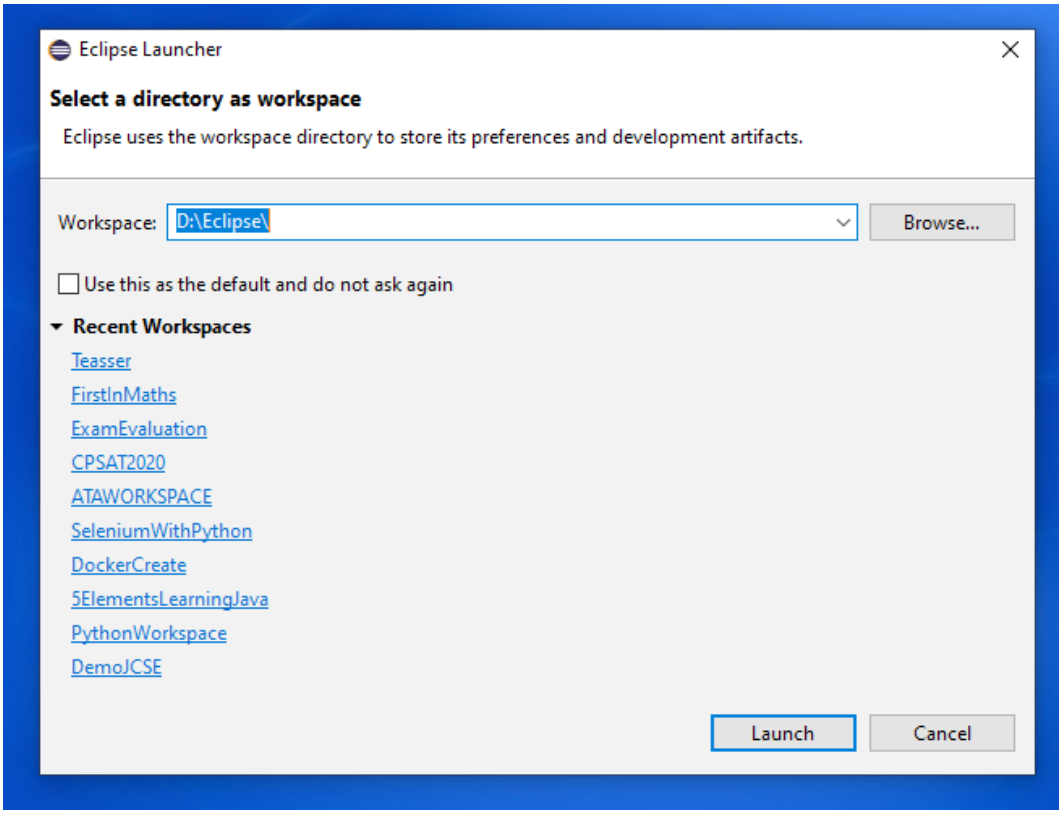


***Figure 2.2**: Creating workspace*

Once you set up the path to your workspace and click on the **`Launch`** button, it will take you to the welcome screen, which we will be closing to move forward to create our new Java project.

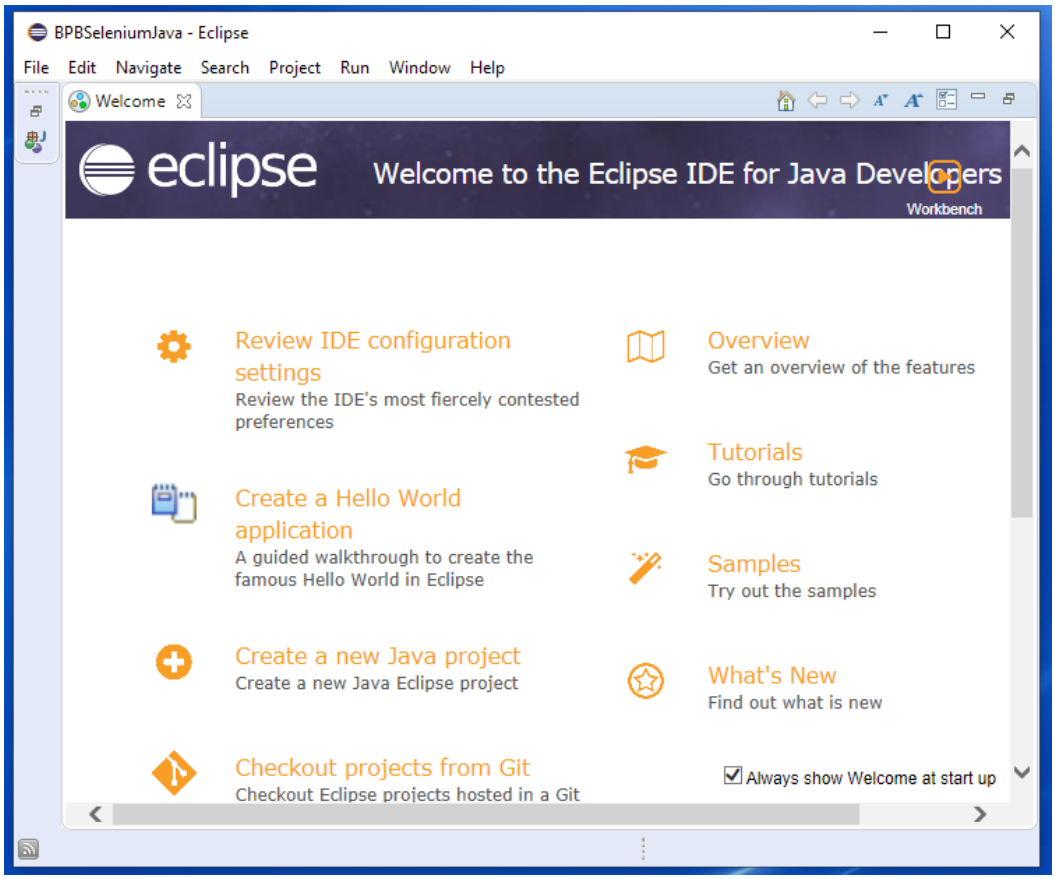


***Figure 2.3**: Eclipse welcome screen*

Create new Java project

To create a new Java project in eclipse, we will click on the **`File`** menu, select **`New`**, and then **`Java Project`**:

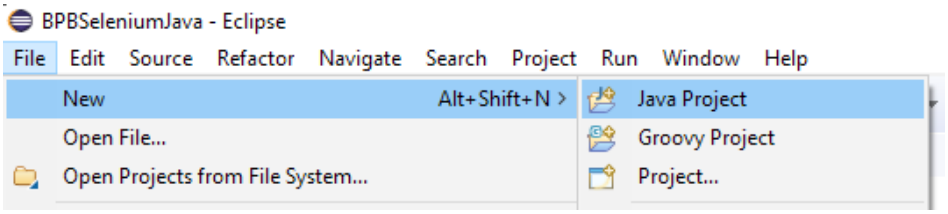


***Figure 2.4**: Create new Java project*

We provide the name of the project and select the JRE version. Please note it is possible to have more than one Java version available on the system, as well as have more than one eclipse version available on the system.

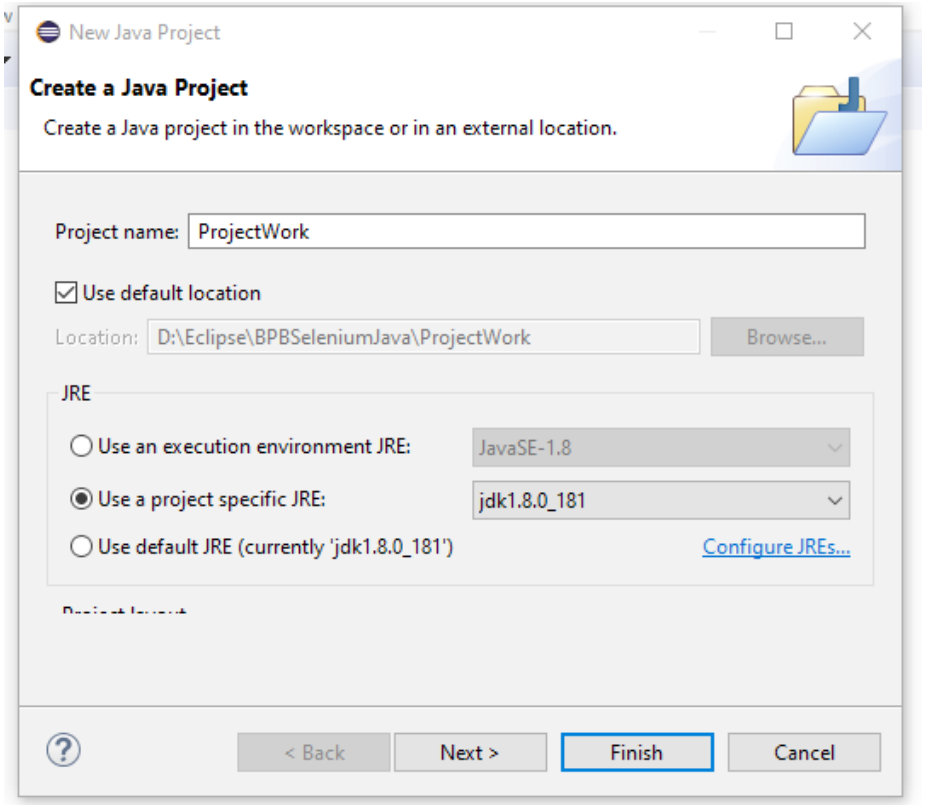


***Figure 2.5**: Project details*

Click on Next, and then click on the Finish button. This will take you to a screen where you will see **`ProjectWork`** in your Package Explorer window on the left side, which one can expand.

The next part is to create a new package and create a **`helloWorld.java`** program to check all settings. For this, right-click on the **`src`** folder, and create a new package.

Name it **packageHello**. Then right-click on the package and select new class, fill in the details. Provide a name as **helloWorld**, and select the main method.

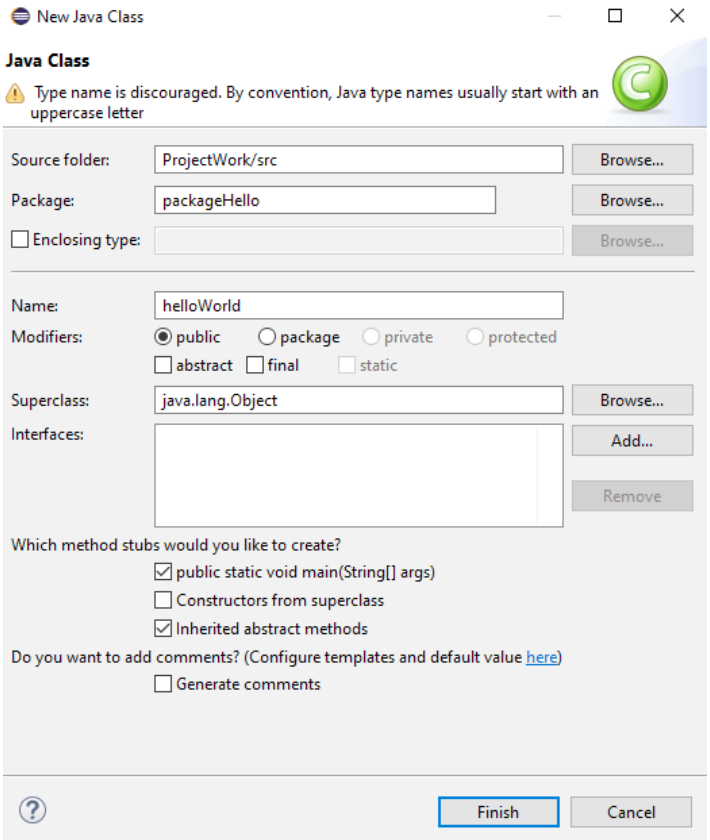


***Figure 2.6**: Hello World java file creation*

Click on **Finish**.

On the right side of the package explorer window, where the **helloWorld.java** file is opened. A code snippet will be automatically generated. There we will write a code line inside the main method to print hello world on the console window. The final code would look as follows:

```
package packageHello;

public class helloWorld {

	public static void main(String[] args) {
		// TODO Auto-generated method stub
		System.out.println("Hello World");
	}
}
```

The output of this program is—we see **`Hello World`** printed on the console window.

Adding Selenium jars

The next step in our project is now to add the Selenium jar files. We will go to the URL—**https://www.selenium.dev/downloads/**, scroll down to the Selenium client and WebDriver language bindings. The current beta release available is version 3.141.59, which we will download.

Java	3.141.59	November 14, 2018	4.0.0-alpha-5	March 18, 2020	Download Alpha Download Changelog API Docs

***Figure 2.7**: Selenium Java client libraries*

Once we download the ZIP file in our system, we will extract the jars available in them. These jar files now have to be added to our Java project in eclipse. For this, we will perform the following steps:

1. Right-click the Java project and select **`Properties`**.

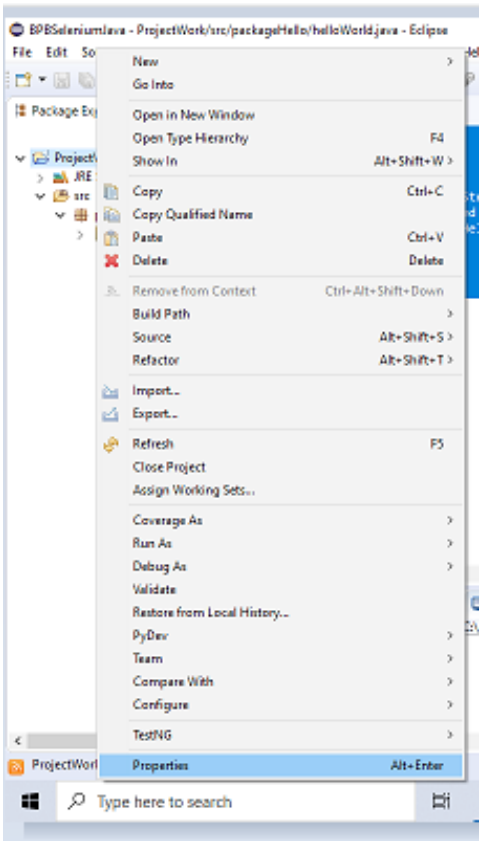


***Figure 2.8**: Project properties*

2. Next is select **`Java Build Path | Libraries | Add External Jars`**, then traverse to the path in the system folder, where you have extracted the Selenium jars.

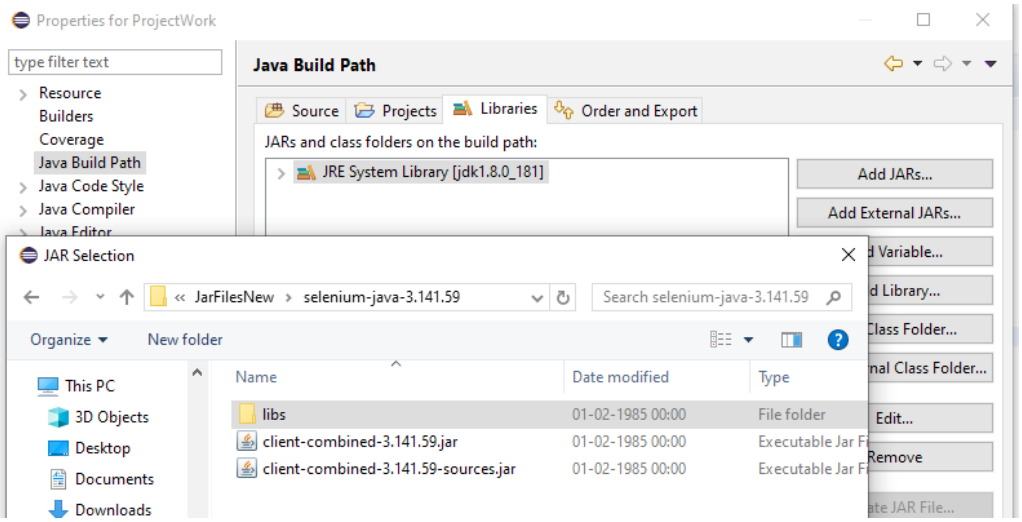


***Figure 2.9**: Adding Selenium jars*

3. Select the two jars available outside, then go inside the **`lib`** folder and add all the jars available in that folder to the project.

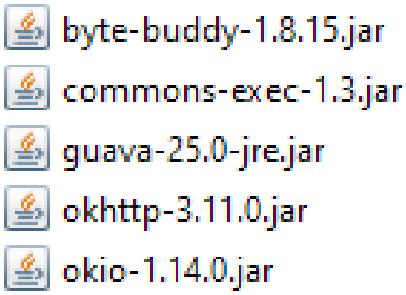


***Figure 2.10**: Selenium jars*

4. Once all jars are added, our project is now ready to accept the Java programs written with Selenium commands.

5. But before we write any Selenium commands to drive the browsers, we should be ready with the respective drivers for our three browsers: Internet Explorer, Firefox, and Chrome, to drive them with commands of our choice.

Set browser drivers

Selenium 3.0 onwards as W3C added it into the regulations to be adopted by the browsers to be compatible with Selenium, the onus of providing the drivers so that Selenium can talk to you fell on the browsers corporations. So now, chrome has to provide a Chrome driver, Internet Explorer has to provide an Internet Explorer driver, and Firefox provides geckdriver, which in the Selenium 2.0 series were maintained and provided by the good people at Selenium.

In an ideal situation, one should have the latest browser version available on the system and then download the compatible driver version available with it. Or with respect to the browser version available on your system, you can check which driver version will be suitable and download that accordingly.

So, first of all, create a folder in your system on which you have reading and writing rights. In that folder, we will be downloading the three different driver executables from the respective websites hosting them.

Browser	Version	Driver—URL—Download
Chrome	Chrome version 83	**https://chromedriver.chromium.org/downloads**
Firefox	Geckdriver v0.26.0	**https://github.com/mozilla/geckodriver/releases**
Internet Explorer	IEDriverServer 3.150.1	**https://www.selenium.dev/downloads/**

Table 2.1: *Browser drivers*

Once these executables are downloaded, extract them from their files, and you will find the respective **`.exe`** files, add them to some folder location like—**`drivers`**/**`resources`**, and so on.

Add this folder to the project workspace:

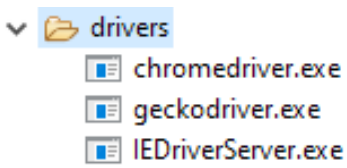


Figure 2.11*: Browser drivers*

We are now ready to run our simple tests on the system.

Walkthrough of BPB application

Throughout the course, we will be using an application to practice our concepts. Let us first begin with understanding the workflow of the application. The URL of the application is **http://practice.bpbonline.com/**

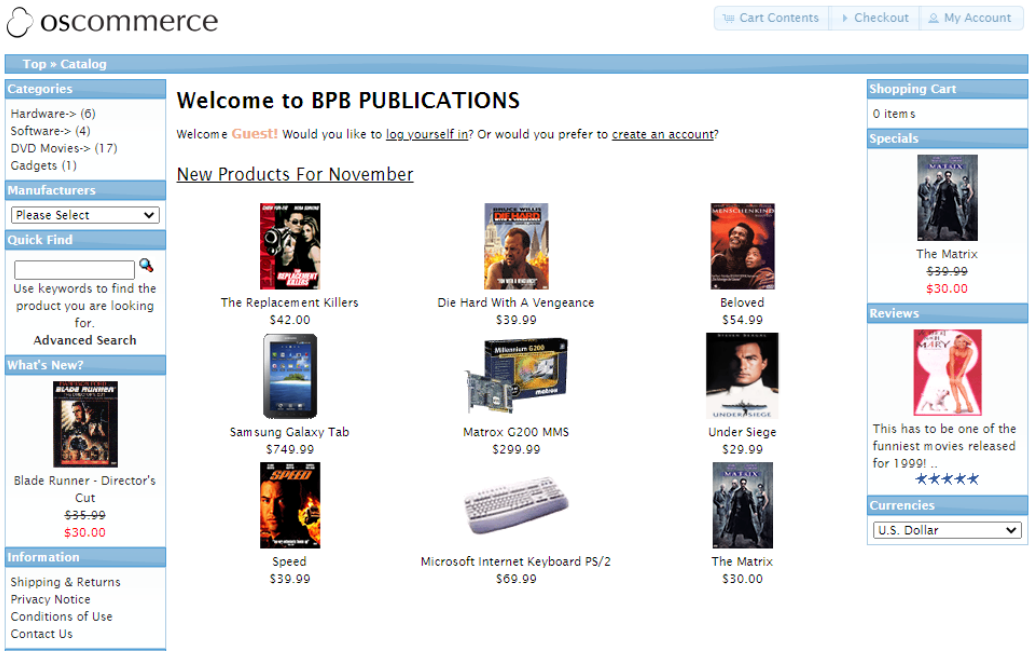


***Figure 2.12**: BPB practice application*

It is a sample eCommerce application with basic workflows, which are as follows:

- Register user
- Login logout
- Search product
- Buy product
- Change profile
- View orders

Our first step as we explore the application is to create a new user. To create a new user, we will have to use the register user scenario. Let us see the steps using which we can do it.

Register user

To register the user, we will have to perform the following steps:

1. Launch the application using the URL—**http://practice.bpbonline.com/**
2. Click on the **`My Account`** link, as highlighted in the following image:

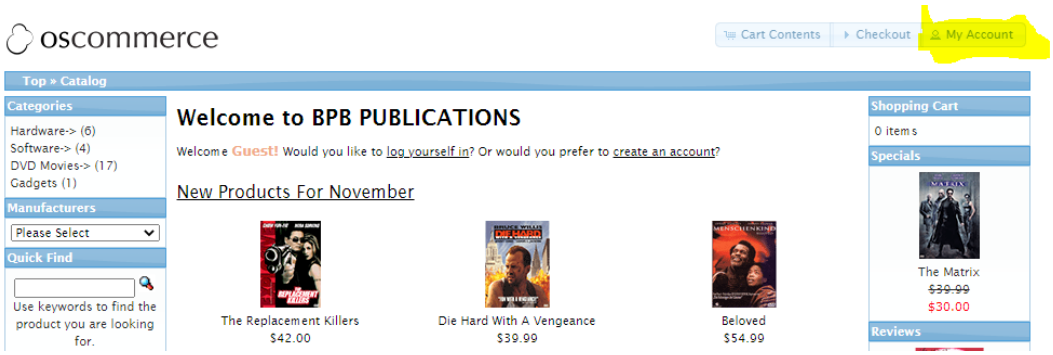


***Figure 2.13**: My Account link*

3. The page which will open is the sign-in page. On this, the next step is to click on **`Continue`** in the **`New User`** section, to go to the register user page. The following image shows it.

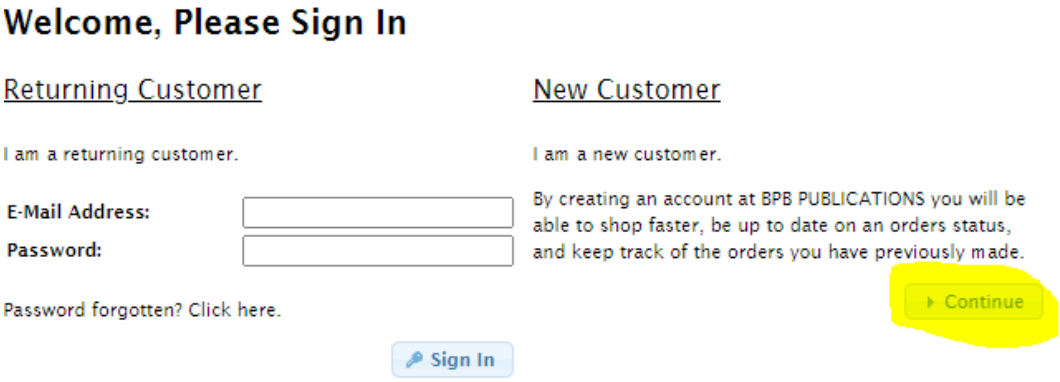


***Figure 2.14**: Click on continue button*

4. Once we are on the register user page, we need to fill in all the mandatory details. Please note that the e-mail address and the password we provide here are the same we will use to login as user credentials once we are successfully

registered. We also need to note here that it is not necessary that we provide an e-mail address that is working. A dummy e-mail address is preferred.

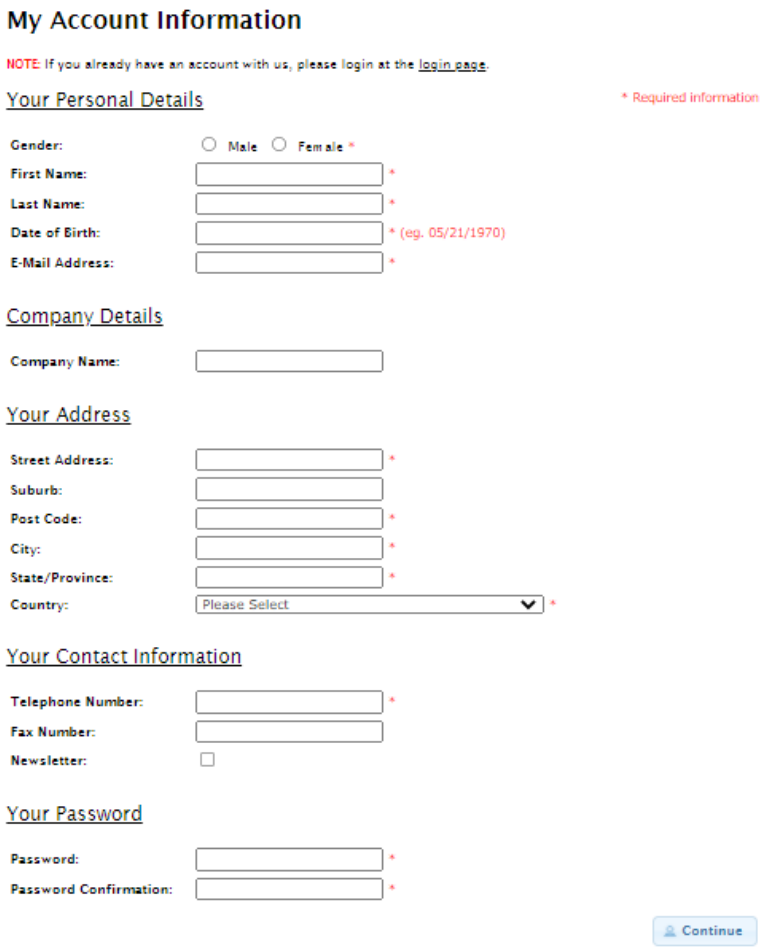


***Figure 2.15**: Register user page*

5. If the user is registered successfully, we should see the following screen:

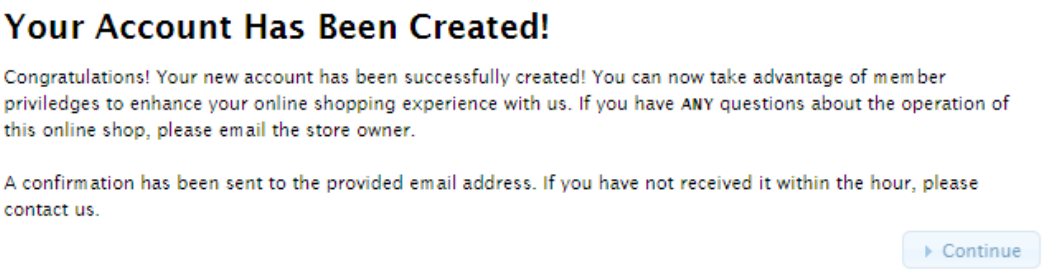


***Figure 2.16**: Account creation confirmation message*

6. Please note that once the account is created, we should be able to use the e-mail address and password combination to login into the application and explore other scenarios. Please also note that we should disable automatic password saving as browser behavior, as this hampers automation.

The next ideal step after registering the user is to login into the application, using the credentials used to register. We will be using the username **bpb@bpb.com**, and password **bpb@123** to login to the application using the following steps:

1. Open the application using URL—**http://practice.bpbonline.com/**
2. Click on the **My Account** link as shown in the following image:

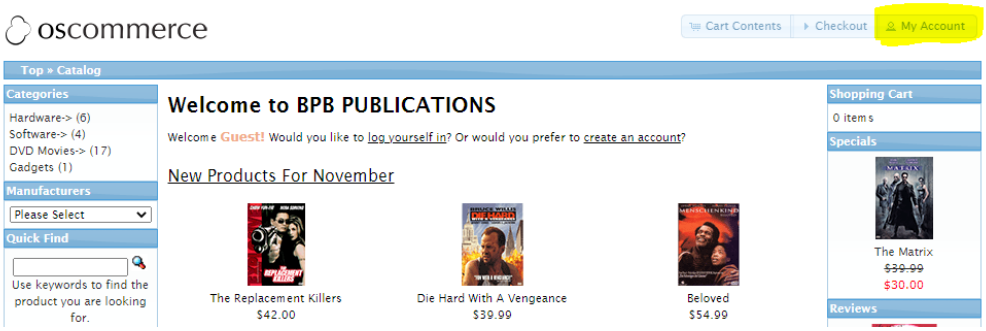


***Figure 2.17**: My Account link*

3. On the sign in page , provide the username, and password credentials, and click the **Sign In** button:

***Figure 2.18**: Account login*

4. If logged in successfully, we should be able to see the **My Account Information** page, where we can see links that allow us to change our

profile information, view orders, modify the address, change the password, and so on. The following image shows this:

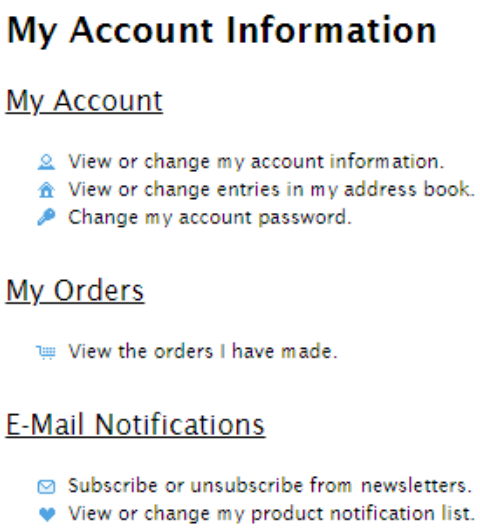


***Figure 2.19**: My Account Information page*

5. Our next step is now to log off from the application to go back to the main page. To perform the log-off operation, we will first click on the **`Log Off`** link.

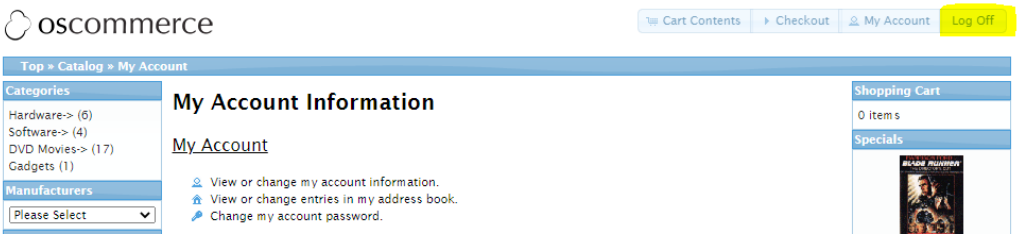


***Figure 2.20**: Log Off link*

6. The next step is to click on **`Continue`** to proceed with the log-off process.

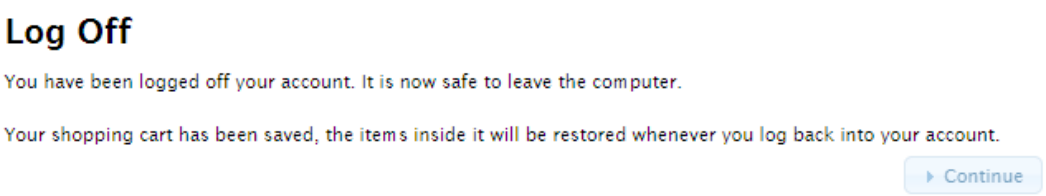


***Figure 2.21**: Finalizing Log Off*

Thus, we are able to login and log off from the application successfully. In this book, to understand the various concepts around Selenium, this would be the default scenario which we will use. We need to also note here that there are other scenarios also available in the application, such as change profile information, buy the product, view the cart, and so on, and so more, and it is left to the reader to explore them and note down their steps.

As and where, in the book, if we require other scenarios, those are then defined in those chapters.

Other applications

In this book, the default application we use is the BPB practice application. But sometimes, to explore and understand some concepts, we may use the following applications:

- Internet Herokuapp, the URL of the application is **https://the-internet.herokuapp.com/**. This application is maintained by Elemental Selenium and provides us with various small examples to explore the working of different types of HTML elements. It is a great application to practice, and use to understand how Selenium handles various Web elements.

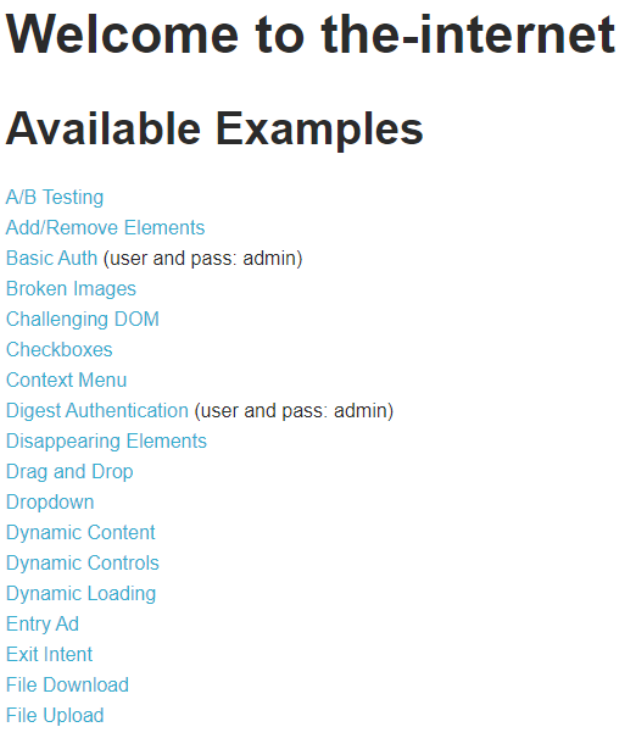


***Figure 2.22**: Internet Heroku app*

- The other application URLs that we can use to practice concepts are any eCommerce website, a website that displays information about movies, books, and home decor. In this book, as and where we use some other applications to understand the scenario, the URL and the flow are explained.

Thus, we had a look at the applications that we can use to be under test to explore how we can automate the scenarios of them using Selenium.

Conclusion

In this chapter, we have seen how we can set our system to set up Java, Eclipse and use it to set up a project. We then verified if the project actually worked. Once confirmed on that front, we downloaded and set up Selenium jars. To ensure our project could use the Selenium jars and manipulate the browsers, we downloaded and set up the browser drivers, which will act as a communication bridge and allow Selenium to communicate with the browser.

We also saw a walkthrough of the practice BPB application. We saw two scenarios of registered users and login logout in detail. We also discussed some other applications, which we can use to understand and implement scenarios as we learn Selenium.

In the upcoming chapter, we will talk about the three important pillars of Selenium implementation. The WebDriver, which is an interface to talk to the browser. The WebElement that provides a method to handle the HTML elements on the page, and the By class that helps in creating a locator to identify the element on the page.

Questions

1. What eclipse version have we used here?
2. What Java version have we installed in this chapter?
3. What is Photon with respect to Eclipse?
4. What is the name of the driver for Firefox?
5. From where can one download the driver for Internet Explorer?

Chapter 3
WebDriver, WebElement, and By

Selenium, the popular browser automating tool, has three main components: the Selenium IDE, WebDriver, and the Grid. In this chapter, we will understand about the three main important components of the WebDriver, which are the WebDriver interface, WebElement interface, and the By class. Any script which we write to automate the browser will use the methods available in these entities. The WebDriver interface provides us with various methods to automate the browser and fetch information associated with the Web page. The WebElement refers to the HTML element on the page. This interface provides us with various methods which help us to automate actions to handle different types of HTML elements. And finally, the By class helps us create locator strategies to locate an element on the Web page on which action can be performed.

Structure

In this chapter, we will discuss the following topics:

- WebDriver and its purpose
    - Methods of WebDriver
    - WebDriver of three browsers [IE, FF, Chrome]
    - Exceptions related to WebDriver

- WebElement interface
    - Methods of WebElement
    - Exceptions with WebElement
- About By class
    - Methods in By class
    - Understanding locators
    - Exceptions related to By class

Objectives

After studying this unit, you will be able to:

- Learn what is a WebDriver, and use its methods to automate the browser.
- Understand what is WebElement, and use its methods to work with Web elements on the Web page.
- Understand the By class used for locator creation and the different types of locator strategies.
- Set up browser drivers in the project.
- Understand the exceptions we may encounter while using WebDriver, WebElement, and By.

WebDriver and its purpose

WebDriver is the main interface that prepares a contract to establish the rules for the browser to follow so that it gets enabled for testing. The functionalities this interface focusses on are:

- Controlling the browser

  The WebDriver interface provides us with methods, which help us in controlling browser-related actions, such as the opening of a browser, the closing of the browser, managing browser window size, timeout events, and more. As we will see in the next section, where we discuss methods in more detail, we will see more about this.

- Selecting the Web elements

  The WebDriver interface provides two methods to look for the element or a list of elements of interest. These methods are **`findElement`** and **`findElements`**,

which provide us with an element or a list of elements, respectively. Once the Web element is identified, we can proceed with the action on it.

- Web page information

  The WebDriver interface provides us method which helps in fetching the current title of the page, the current URL, and the content available on the page. The information provided by these methods provides information related to the Web page and can also be used for validation.

In the next section, we will first understand how to set WebDrivers for different browsers. After that, we will look at the list of methods available with the WebDriver interface and their explanations. We will also see some implementation examples to understand the working of the methods.

Set browser drivers

Selenium 3.0 onwards as W3C added it into the regulations to be adopted by the browsers to be compatible with Selenium, the onus of providing the drivers so that Selenium can talk to you fell on the browsers corporations. So now, Chrome has to provide Chrome driver, Internet Explorer has to provide Internet Explorer driver, and Firefox provides geckodriver.

In an ideal situation, one should have the latest browser version available on the system and then download the compatible driver version available with it. Or with respect to the browser version available on your system, you can check which driver version will be suitable and download that accordingly.

So, first of all, create a folder in your system on which you have reading and writing rights. In that folder, we will be downloading the three different driver executables from the respective websites hosting them.

Browser	Version	Driver—URL—Download
Chrome	Chrome Version 83	**https://chromedriver.chromium.org/downloads**
Firefox	Geckodriver V0.26.0	**https://github.com/mozilla/geckodriver/releases**
Internet Explorer	IEDriverServer 3.150.1	**https://www.selenium.dev/downloads/**

***Table 3.1**: Three different driver executables*

Once these executables are downloaded, extract them from their files, and you will find the respective .exe files, add them to some folder location like for us; it is **`driverexes`** in the project folder.

The following image shows the driver executables available in the project:

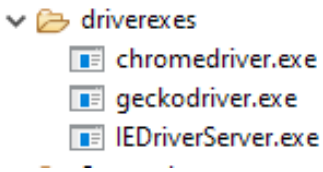


***Figure 3.1**: Driver executables for browsers*

Let us see the usage of these WebDriver executables to launch browsers Chrome, Firefox, and Internet Explorer, respectively.

Script to launch Chrome Browser with Application URL:

```
import org.openqa.selenium.*;
import org.openqa.selenium.chrome.*;
public class runScriptOnChrome {

	public static void main(String[] args) throws Exception {
		//set system property
		System.setProperty("webdriver.chrome.driver",
"driverexes\\chromedriver.exe");
		//Initialize driver object
		WebDriver driver = new ChromeDriver();
		//open chrome browser with the url.
		driver.get("http://practice.bpbonline.com/");
		Thread.sleep(5000); //wait for 5 seconds
		driver.close();
	}
}
```

As we can see in the preceding script, the command:

`System.setProperty("webdriver.driver.chrome", driverexes\\chrome driver.exe")` provides the information where chromedriver.exe file is available in the system location.

The next command—**`WebDriver driver = new ChromeDriver();`** create an instance of the **`ChromeDriver`**. We should note here for this statement to work; it is important that we have imported the correct libraries in the code file:

```
import org.openqa.selenium.*;
import org.openqa.selenium.chrome.*;
```

The next step is to use the **`get()`** method to launch the Chrome browser using the URL for the application under test—**http://practice.bpbonline.com/**. After that, we

wait for 5 seconds using **`Thread.sleep(5000)`** command. And then we use the **`driver.close()`** command to close the browser which is opened.

In the same manner, if we want to launch the Firefox browser, we will be importing the library for geckodriver, and providing the path to the geckodriver executable file. The script is as follows:

```
import org.openqa.selenium.*;
import org.openqa.selenium.firefox.*;
public class runScriptOnFirefox {
	public static void main(String[] args) throws Exception {
		//set system property
		System.setProperty("webdriver.gecko.driver",
"driverexes\\geckodriver.exe");
		//Initialize driver object
		WebDriver driver = new FirefoxDriver();
		//open firefox browser with the url.
		driver.get("http://practice.bpbonline.com/");
		Thread.sleep(5000); //wait for 5 seconds
		driver.close();
	}
}
```

The script preceding launches Firefox browser with the application URL and then closes it. We need to note that it uses the geckodriver executable file, which is already available in the **`driverexes`** folder of the project.

Now for launching Internet Explorer, we will be using the Internet Explorer driver executable and will import the respective library to work with Internet Explorer browser.

```
import org.openqa.selenium.*;
import org.openqa.selenium.ie.*;
public class runScriptOnInternetExplorer {
	public static void main(String[] args) throws Exception {
		//set system property
		System.setProperty("webdriver.ie.driver", "driverexes\\
IEDriverServer.exe");
		//Initialize driver object
		WebDriver driver = new InternetExplorerDriver();
		//open ie browser with the url.
		driver.get("http://practice.bpbonline.com/");
		Thread.sleep(5000); //wait for 5 seconds
		driver.close();
	}
}
```

The script preceding launches the Internet Explorer browser with the application URL, and then closes it. We need to note that it uses the geckodriver executable file, which is already available in the **`driverexes`** folder of the project.

In the next section, we will list out the various methods of the WebDriver, and see a few script snippets to understand the working of these to automate the browser functions.

Methods of WebDriver

The methods associated with WebDriver are written for the browser object. The interface functions are provided for the browser object. Classes such as ChromeDriver, FirefoxDriver, and other browser driver classes implement the WebDriver interface, and thus, when we create an instance of the respective browser driver class, we are able to access the methods.

Let us have a look at the methods listed in the following table:

Method	Purpose
`get()`	This function is used to open the browser with the given URL in the function.
`navigate()`	This function is used to navigate to the page provided in the function.
`close()`	This function is used to close the current browser window open with the browser session.
`quit()`	This function is used to close all the browser windows opened with the browser session.
`switchTo()`	This function is used with the browser object to switch to a new window, alert, or frame.
`getTitle()`	This function returns the text of the current title.
`getCurrentUrl()`	This function returns the current URL of the page.
	This function returns the current window id of the page.
	This function returns all the window ids which are generated for the Web pages associated with the current driver session.
`manage()`	This function has multiple utilities. It allows browser windows to maximize, minimize. It also helps in implicit time management, and so on.
`findElement()`	This function returns the first element found with the provided locator strategy used.

findElements()	This function returns all the elements matched for the given locator strategy.

***Table 3.2**: Methods of WebDriver*

We will be looking at the preceding mentioned methods based on the broad classification of actions they will be performing on the browser.

Controlling the browser

The methods which we will be looking in for the browser control are **get()**, **close()**, **quit()**, **switchTo()**, and **manage()**. We will be looking at code snippet for these and take them as examples. As we know, the **get()** method allows the opening of the browser with the given URL. The **close()** method closes the current browser instance opened. The **quit()** method closes all the windows which are opened using the current driver, and then **navigate()** method allows us to go to another Web URL. The **manage()** methods have other sub-methods; we will see the usage of it to maximize the browser, for example. Let us have a look at the code snippet where we try these. Please note that the difference between close and quit is handled in the chapter Extra Concepts, where we discuss about the **switchTo()** method of the browser in detail to handle a popup, a frame, or a new window. For now, let us look at some of the methods for browser manipulation:

```
import org.openqa.selenium.*;
import org.openqa.selenium.chrome.*;

public class browserManipulationExample {

      public static void main(String[] args) throws Exception {

            //set system property
            System.setProperty("webdriver.chrome.driver",
"driverexes\\chromedriver.exe");
            //Initialize driver object
            WebDriver driver = new ChromeDriver();
            //browser manipulation method
            driver.get("http://practice.bpbonline.com/");
            driver.manage().window().maximize();
            driver.navigate().to("http://www.selenium.dev");
            driver.close();
      }
}
```

In this code, we see the get method to launch the browser with the URL; the code **manage().window().maximize()**, will maximize the browser window. The next

command will navigate to the Selenium main website, and finally, **`driver.close()`** will close the browser. In the next section, we will modify the preceding script to add some methods which can help **`iprint`** Web page-related information like title, **`currenturl`**, and page content.

Web page information methods

The Web page information method are **`getCurrentUrl()`**, **`getPageSource()`**, and **`getTitle()`**. These methods will give Web page-related information, which we can use to find out if we are on the correct page on which we want to continue actions. These methods also allow us to perform text-based validation, where we search for text on the Web page content. Let us look at the following code snippet:

```
import org.openqa.selenium.*;
import org.openqa.selenium.chrome.*;

public class webPageInformation {

	public static void main(String[] args) throws Exception {

		//set system property
		System.setProperty("webdriver.chrome.driver",
"driverexes\\chromedriver.exe");
		//Initialize driver object
		WebDriver driver = new ChromeDriver();
		//browser manipulation method
		driver.get("http://practice.bpbonline.com/");
		System.out.println(driver.getTitle());
		if(driver.getPageSource().contains("Welcome to BPB
PUBLICATIONS")) {
			System.out.println("Page is loaded");
		}
		driver.manage().window().maximize();
		driver.navigate().to("http://www.selenium.dev");
		if(driver.getCurrentUrl().contains("selenium")) {
			System.out.println("Page is now changed to
Selenium website");
		}
		driver.close();
	}
}
```

In the preceding code, we open the application URL and then print the title of the page. We then search if the page contains a text—Welcome to BPB PUBLICATIONS

or not. Finally, we wanted to find if the navigated URL contains the word Selenium or not. The output of the preceding code is as follows:

```
BPB PUBLICATIONS
Page is loaded
Page is now changed to Selenium website
```

In the next section, we will see the usage of the **findElement**, and find elements method.

Finding elements methods

Let us say we want to identify the link **My Account** on the Web page of our BPB application. To perform this action, we will need to use the **findElement()** method available with the WebDriver. But we also need to know that this method takes an input argument. The input argument this method takes is a By object, which basically provides information about what locator strategy and data we can use to identify the element. We also need to understand that the return value of this method is an object of the **WebElement** type, on which we can perform an action like click, to basically automate clicking on **My Account** link; let us see the code snippet as follows:

```
import org.openqa.selenium.*;
import org.openqa.selenium.chrome.*;
public class findingElements {

      public static void main(String[] args) throws Exception {

            //set system property
            System.setProperty("webdriver.chrome.driver",
"driverexes\\chromedriver.exe");
            //Initialize driver object
            WebDriver driver = new ChromeDriver();
            //browser manipulation method
            driver.get("http://practice.bpbonline.com/");
            driver.manage().window().maximize();
            WebElement myAccountlink=driver.findElement(By.
linkText("My Account"));
            myAccountlink.click();
            driver.close();
      }
}
```

In the preceding code, we have found the **My Account** link using the link text locator strategy, where we have passed the text of the link as **My Account**. In the next section, we will use the **findElements** method to find all the links on the home

page and print the text associated with them on the screen. The code snippet is as follows:

```
import org.openqa.selenium.*;
import org.openqa.selenium.chrome.*;
import java.util.*;
public class findingAllLinks {

	public static void main(String[] args) throws Exception {

		//set system property
		System.setProperty("webdriver.chrome.driver",
"driverexes\\chromedriver.exe");
		//Initialize driver object
		WebDriver driver = new ChromeDriver();
		//browser manipulation method
		driver.get("http://practice.bpbonline.com/");
		driver.manage().window().maximize();
		List<WebElement> allLinks=driver.findElements(By.
xpath("//a"));
		for(WebElement lnk:allLinks) {
			System.out.println(lnk.getText());
		}
		driver.close();
	}
}
```

In the preceding code snippet, we have used an **`xpath`** locator strategy to find all links on the Web page. The method we used for **`findElements`**, which returns to us a list of all Web elements found. Once found, we used a method called as **`getText()`**, which is available with the WebElement, to print the text information associated with the method. In the next section, we will see the different functions of WebElement.

The WebElement interface

A Web element is an HTML element, which is used to create and help function a Web page. These are also known as HTML elements. An HTML element is created using HTML tags, which are predefined. They can be a link, a table, a form, a button, a text box, an image, and so on. Depending on the version of the HTML we are using, a list of new elements become available to us, which are now as well supported by the Web browsers. The latest version provided by the W3C—**https://www.w3.org/TR/html52/**, HTML 5.2 version.

Note The list of Web elements in detail can be found here—https://www.w3.org/TR/html52/semantics.html# semantics

Generic structure of WebElement

A HTML element is generally represented as follows:

```
<HTML TAG  PROP=VAL PROP=VAL .. >  TEXT </HTML TAG>
```

So at first, the HTML tag starts, it is then followed by a group of property-value pairs, and then the HTML tag closes. The HTML tags are already predefined in the W3C page, as discussed earlier. With the help of these Web elements, the Web pages are designed, and they function to achieve a business objective. The properties used to declare a Web element helps display them on the page and are later used by the automation tools to identify them uniquely on the Web page.

Methods of Web Elements

Each Web element has different functions associated with them. Some are used for presentation purposes; some are used to group together information. Elements like links, images are used for hyper referencing other pages or images. Others like span and div help create overlays. Selenium library for Web elements interface defines different methods, which we can use on the Web elements to help us achieve our business scenarios in action.

Note: Details on the methods of the Web elements are provided in the link: https://www.selenium.dev/selenium/docs/api/java/org/openqa/selenium/WebElement.html

In the following table, we are going to see the names of the various methods declared, along with the purpose they solve when we use them in the scripts.

For the approach to use them and their return values, please refer to the link shared in the preceding note.

Let us see them in the following table:

Method	Purpose
`sendKeys()`	Types context on a given object, for example, types text on a textbox.
`click()`	Performs click operation on a clickable entity, example, a link, button.
`getText()`	Fetches the text associated with a html object.

`isDisplayed()`	This is used to understand if the element is available on the page or not.
`isEnabled()`	This function is used to check if the element is functional or not.
`isSelected()`	This is used for radio and check buttons to find their selection status.
`clear()`	This is used for text fields to clear their contents.
`getAttribute()`	This function is used to provide us with the run time value associated with a given property.
`getTagName()`	It provides the html tag name for a html object.
`submit()`	It performs the form submission action.
`getRect()`	It returns the location of the Web element.
`getSize()`	This provides us the information on the size of the Web element.
`getCssValue()`	This returns the CSS value associated with the element.
`findElement()`	This returns the Web element inside another Web element.
`findElements()`	This provides a list of all Web elements available inside another Web element. For example, provide the list of all options inside select.

***Table 3.3**: Various methods and their purpose*

To understand the working of the WebElement methods, we will first see the methods which perform some action on the Web elements on the Web page like **`sendKeys`**, click, clear. Then we will see the methods which have to get word before them, which fetch some information associated with the Web elements on the Web page. After that, we will see the methods **`isSelected()`**, **`isEnabled()`**, and **`isDisplayed()`**, which provides information about the state of the method. The **`findElement()`** and **`findElements()`** method work the same as described preceding for the Web page. And we will see their usage when we discuss the handling of dropdown and Web table elements in the upcoming chapters.

Action methods on WebElement

To understand the action methods like **`sendKeys()`**, **`click()`**, **`clear()`** for WebElement, we will use the BPB application. Here, after opening the page, we will first clear the search text box, then type a search text on it, and click on the search icon. The following image shows us the highlighted part:

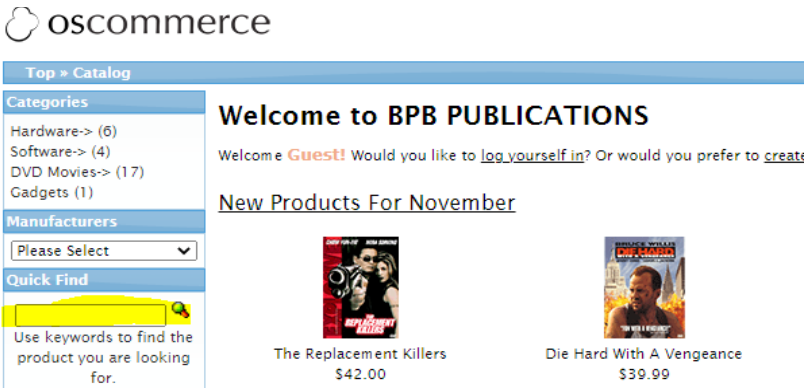


***Figure 3.2**: Home Page with search box*

The code snippet is as follows:

```
import org.openqa.selenium.*;
import org.openqa.selenium.chrome.*;
public class actionOnWebElement {

      public static void main(String[] args) throws Exception {

            //set system property
            System.setProperty("webdriver.chrome.driver",
"driverexes\\chromedriver.exe");
            //Initialize driver object
            WebDriver driver = new ChromeDriver();
            //browser manipulation method
            driver.get("http://practice.bpbonline.com/");
            driver.manage().window().maximize();
            WebElement searchField=driver.findElement(By.
name("keywords"));
            //clear action
            searchField.clear();
            //type action
            searchField.sendKeys("mouse");
            WebElement quickFind=driver.findElement(By.xpath("//
input[@title=' Quick Find ']"));
            //click action
            quickFind.click();

            driver.close();
      }
}
```

In this code, we first find how to identify the search text box, which we do use the name locator strategy, and then, we clear its content, if any, and then use the **sendKeys()** method to type text on it. Here, we need to note that the **sendKeys()** method doesn't by default clear the contents of the textbox, and if something is written before, it will start typing from after it. We then identify the quick find icon using locator strategy **xpath**, and on it perform the click action. In the next, we will see some of the methods that help us fetch information about the Web elements.

Fetching information methods on WebElements

To understand the working of the get methods like **getAttribute()**, **getTagName()**, **getText()**, **getRect()**, and **getSize()**, we will use the quick find icon from the home page of the BPB application and apply all these methods on it. Let us see the following code snippet:

```
import org.openqa.selenium.*;
import org.openqa.selenium.chrome.*;
public class fetchElementInformation {

	public static void main(String[] args) throws Exception {

		//set system property
		System.setProperty("webdriver.chrome.driver",
"driverexes\\chromedriver.exe");
		//Initialize driver object
		WebDriver driver = new ChromeDriver();
		//browser manipulation method
		driver.get("http://practice.bpbonline.com/");
		driver.manage().window().maximize();
		WebElement quickFind=driver.findElement(By.xpath("//
input[@title=' Quick Find ']"));
		System.out.println(quickFind.getAttribute("alt"));
		System.out.println(quickFind.getTagName());
		System.out.println(quickFind.getText());
		System.out.println(quickFind.getCssValue("background-
color"));
		System.out.println(quickFind.getSize().height);
		System.out.println(quickFind.getRect().height);
		driver.close();
	}
}
```

The output of the preceding code, which fetches the various information associated with the Web element QuickFind is as follows:

```
Quick Find
input
rgba(0, 0, 0, 0)
17
17
```

Checking the state of WebElement

To understand the working of the methods **`isSelected()`**, **`isEnabled()`**, and **`isDisplayed`**, we will use the URL—**https://the-internet.herokuapp.com/checkboxes**, this page displays two checkboxes, we will click on first and check the before and after **`isSelected()`** value.

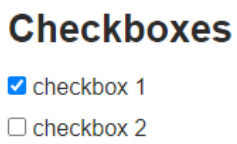


***Figure 3.3**: Checkboxes Web page*

We will print the information available with **`isEnabled()`** and **`isDisplayed()`** as well as with the checkbox. Let us see the code snippet as follows:

```
import org.openqa.selenium.*;
import org.openqa.selenium.chrome.*;
public class checkingStateOfWebElement {

      public static void main(String[] args) throws Exception {

            //set system property
            System.setProperty("webdriver.chrome.driver",
"driverexes\\chromedriver.exe");
            //Initialize driver object
            WebDriver driver = new ChromeDriver();
            //browser manipulation method
            driver.get("https://the-internet.herokuapp.com/checkboxes");
            driver.manage().window().maximize();
            WebElement firstCheckbox=driver.findElement(By.
cssSelector("#checkboxes"));
            System.out.println(firstCheckbox.isDisplayed());
            System.out.println(firstCheckbox.isEnabled());
            System.out.println(firstCheckbox.isSelected());
            firstCheckbox.click();
            System.out.println(firstCheckbox.isSelected());
            driver.close();
      }
}
```

The output of the preceding code snippet is as follows:

```
true
true
false
false
```

In the next section, we will talk briefly about the type of exception, which can come up as we handle a WebElement object.

Exception with Web Elements

While working with Web elements, different kinds of exceptions can be encountered. The most common exception is **`NoSuchElementElementException`**. This exception is generated when the element we are trying to find on the Web page with the given locator information does not exist. Another Web element-specific exception we see commonly is **`StaleElementReferenceException`**. We see this issue when the page has refreshed, and element status is no longer associated with the available driver session. Another one commonly encountered is **`ElementNotVisibleException`**; this can be easily traced; if we look at the HTML properties of the element, we will find that it has a hidden property set to true. Such elements will also return **`isDisplayed()`** as false.

Note: More details on selenium exceptions can be found here: https://www.selenium.dev/selenium/docs/api/py/common/selenium.common.exceptions.html

About By class

In the previous sections, we have seen the WebDriver object methods and the WebElement methods, which are available. Using these methods, we can work with the Web browser object and the Web elements of the page. To work with the Web elements which we see on the page, we need to first identify them uniquely on the page. The By class helps in identifying objects on the Web page. It uses the different location mechanisms based on which HTML objects can be recognized. Every HTML object is defined using an HTML tag, and HTML properties. Each HTML property is required to have a value associated with it. Using information associated with the HTML property and various techniques of the locator mechanism, we can identify the object uniquely on the Web page. Once the object is identified uniquely, we can perform an action on it.

The various ways in which Selenium allows an object to be recognized is by using the HTML property **`ID`**, **`NAME`**. It also allows identifying the object through its XPATH on-page. We can also use the CSS selector methodology, and finally, we can use the DOM approach to identify the object on the page.

Methods in By class

The Selenium Java documentation website lists down the various methods associated with the By class:

Method	Description
`static By className(java.lang.String className)`	To find an element with the help of the `class` property.
`static By cssSelector(java.lang.String cssSelector)`	To find the element by using the `css` strategy to find the element.
`boolean equals(java.lang.Object o)` `WebElement findElement(SearchContext context)`	To return the first element found, which matches the locator used.
`abstract java.util.List<WebElement> findElements(SearchContext context)`	To find the list of all elements which matches the locator used.
`static By id(java.lang.String id)`	To find the element using the `id` property of the element.
`static By linkText(java.lang.String linkText)`	To find the element using the link text value.
`static By name(java.lang.String name)`	To find the element using the name property value of the element.
`static By partialLinkText(java.lang.String partialLinkText)`	To find the element using the partial Link text value of the element.
`static By xpath(java.lang.String xpathExpression)`	To find the element by finding the xpath of the element in the HTML document.

***Table 3.4**: By class methods*

Understanding locators

Let us take an example of a Web page HTML snippet to understand locators:

```
<form name="login" action="https://5elementslearning.dev/demosite/
login.php?action=process&amp;osCsid=a7bbd18f550992ba489722cc2bc9ac35"
method="post"><input type="hidden" name="formid"
value="b5ff4c56fd83735785e1ba02a564bb50" />
```

```
    <table border="0" cellspacing="0" cellpadding="2" width="100%">
      <tr>
        <td class="fieldKey">E-Mail Address:</td>
        <td class="fieldValue"><input type="text" name="email_address"
/></td>
      </tr>
      <tr>
        <td class="fieldKey">Password:</td>
        <td class="fieldValue"><input type="password" name="password"
maxlength="40" /></td>
      </tr>
    </table>


    <p><a href="https://5elementslearning.dev/demosite/password_
forgotten.php?osCsid=a7bbd18f550992ba489722cc2bc9ac35">Password
forgotten? Click here.</a></p>
    <p align="right"><span class="tdbLink"><button id="tdb5"
type="submit">Sign In</button></span><script type="text/
javascript">$("#tdb5").button({icons:{primary:"ui-icon-key"}}).
addClass("ui-priority-primary").parent().removeClass("tdbLink");</
script></p>
    </form>
```

From the HTML part of the code, we can get the following locator information.

ID: The **id** locator is an attribute for an HTML element. For example, if we look at the preceding code, we see a sign-in button HTML, in which the HTML element has an id attribute.

```
<button id="tdb5" type="submit">Sign In</button>
```

So to identify this button on the page, we can easily use the id attribute and create a locator command as in - **driver.findElement(By.id("tdb5"))**.

NAME: The **name** locator information is also fetched from the HTML element. Name is an attribute associated with the HTML elements. For example, in the preceding HTML snippet, we see an email text box and a password textbox; for both identification, we can use the name locator.

- Email: **driver.findElement(By.name("email_address"))**
- Password: **driver.findElement(By.name("password"))**

XPATH: The **xpath** information is derived from the path it takes to traverse to reach the HTML element of interest. It is mainly used for XML documents, but we can take the HTML page as an XML page and then create the path to reach the element of choice. The xpath could be of two types

- Absolute Xpath
- Relative Xpath

The syntax of creating absolute xpath is always from the root node HTML, and for the relative path, it is from the node itself or any of the parents nodes which has a strong identification. We call those nodes landmark nodes, which have either an **ID** attribute or the **NAME** attribute. As with respect to those nodes, we can create the path to reach the node we want to work upon.

For example, the relative xpath for the email text box could be:

Email box- **//input[@name="email_address"]**.

Or if we create with respect to the form node, which is containing the email address it will be:

WRT form node- **//form/table/tr[1]/td[2]/input**

The syntax of creating an xpath is - **//htmltag[@prop=value]**.

And for absolute xpath is always from the html node - **/html/tag1/tag2/tag3…**

XPaths can also be created using a keyword **contains**. We can use any attribute or even text to create this xpath. For example, let us for the sign-in button, the xpath can be created as follows:

```
Xpath=//*[contains(text,'Sign In')]
```

The xpath locator information created would be - **driver.findElement(By.xpath("path"))**.

CSS: The **CSS** stands for the **Cascading Style Sheets**. These are used to help style the elements of the Web page. Instead of creating a style for every individual element, the CSS files are created, where for a tag type, a style is defined. We can use CSS to identify the object on the page. The syntax for creating a CSS is as follows:

```
css=#id or css=htmltag[prop=value]
```

DOM: The document object model is what any HTML Web page is based on. It basically talks about the HTML DOM being in the form of a tree-like structure where every node is in a hierarchical relationship with the other node. Generally, an HTML DOM is represented as follows:

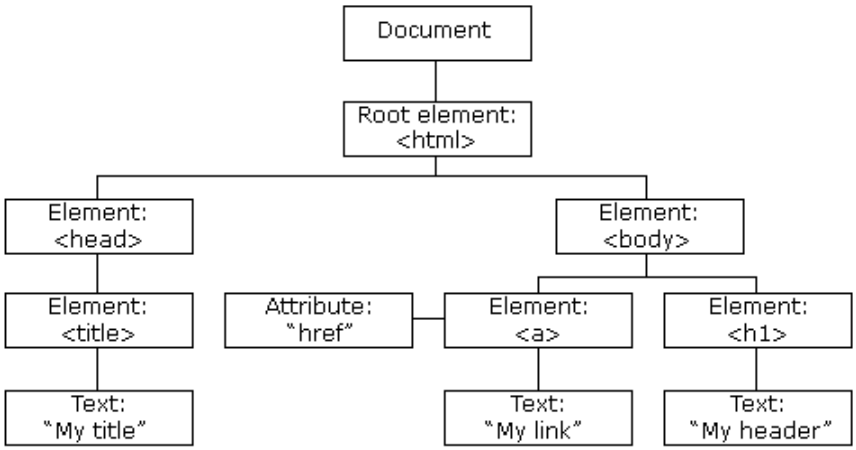


***Figure 3.4**: A HTML DOM*

To use the DOM information to identify the object of interest, we use the fact that all types of HTML tag types are allocated in the form of an array. And to identify a particular object of a certain HTML tag type, we can then use its position in the path to creating the locator information.

So over and all, we need to understand that it is the same HTML code snippet that we have available for any HTML element. And by using the various Web technologies of HTML, XML, xpath, css, and DOM, we can create locators for the object for Selenium to consume and help us identify the object uniquely on the Web page on which action can take place.

Exception with the By class

The exception which we can see when we use bad locators with the By class are as follows:

- **`InvalidSelectorException`**: This exception is thrown when an xpath selector strategy is used to find an element, the xpath used is syntactically wrong.
- **`NoSuchAttributeException`**: This exception is found when the attribute used could not be found for the element.
- **`ElementNotSelectableException`**: This exception is found when the element we are trying to work with is disabled, even though it is present in the DOM.

Conclusion

In this chapter, we discussed the interfaces of WebDriver and WebElement, and also the By class. With the help of WebDriver, we are able to drive the browser of our choice and interact with it. WebElement that are effectively the HTML elements using which the Web pages are created. Each Web element is defined using its HTML properties and values, which are also used in locator strategies to identify them while writing automation scripts. Various functions are available in Selenium Web element interface to act on the Web elements, which help us achieve our objective. The **By** class has methods that use the objects HTML properties and using them identifies the objects. We saw how the class is crucial as it helps to identify the objects on the Web page. On the identified objects, the actions will be performed. It also has methods to identify either one object or a list of objects which matches the same information. We will perform actions on these objects to achieve the purpose of our automation.

The WebElement, WebDriver, and By class create the foundation of the script designing in Selenium WebDriver. In the upcoming chapter, we will discuss about the implementation of our first script in Selenium using the learnings from WebDriver, WebElement, and the **By** class.

In the upcoming chapter, we will talk about the concept of synchronization, understand its importance, and how it is implemented.

Questions

1. What eclipse version have we used here?
2. What Java version have we installed in this chapter?
3. What is Photon with respect to Eclipse?
4. What is the name of the driver for Firefox?
5. From where can one download the driver for Internet Explorer?
6. What is the generic structure of a Web element?
7. Name some HTML tags used to create a Web element?
8. Why do we get a NoSuchElementException?
9. What is the difference between findElement() and findElements() methods? Which method do we use to type text on a textbox?
10. What is the difference between the findElement() and findElements() method?

11. What type of exception is generated if the xpath provided has wrong syntax?
12. How will you fetch at run time an object using its class name?

Chapter 4
Concept of Synchronization

In the previous chapter, we were able to finally implement our learnings about the three important pillars from the Selenium library and automate the launching of the browser with the URL of the practice application on the three browsers—Chrome, Firefox, and Internet Explorer. As we learn about the pillars of Selenium, it is important to understand one of the main pillars of test automation, which is synchronization. In this chapter, we will understand the concept and importance of having the scripts execution synchronized with the application processing of commands. So let us begin.

Structure

The chapter is divided into the following sections:

- What is synchronization, and why is it important
- Types of synchronization
- How to implement implicit wait
- How to implement explicit wait
- How to detect scripts that may have synchronization issues.

Objectives

After studying this chapter, you should be able to identify where waits are required in the script and accordingly add either the implicit wait or the explicit wait. You should also be able to identify if the script is failing because of a synchronization issue and fix it. We will also be able to decide where we can use the synchronization in the script.

Understanding synchronization and its importance

The dictionary meaning of synchronization means an activity of two or more things at the same time or rate. When we are executing a scenario ourselves without the help of any tool, we can synchronize our actions as per the speed of the Web application, as we can see and understand using our cognitive abilities, whether we should wait for an action to complete or proceed with the next instruction.

However, when we replace the same set of actions with an unattended mode, which basically means, when we write a script to perform the same actions, the script itself lacks any cognitive abilities to judge whether to wait for the action to complete or proceed with the next step.

In the case where we do not introduce the concept of synchronization in our scripts, we are causing an error in the script, where it may start failing. As the tool driving the script may fall out of sync with speed, the Web application is able to process it. By taking an example here, let us say we launched our application—practice application using the URL on chrome browser as shown in the following figure:

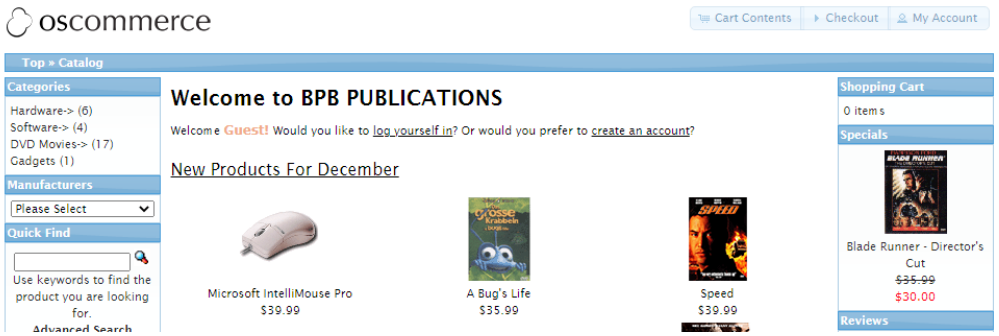


***Figure 4.1**: Example of launching an application*

Now, we perform a click operation on **`My Account`** link; then the next step for us is to type the email address as shown in *figure 4.2*:

***Figure 4.2**: My Account page of the application*

There could exist a few situations explained as follows:

- **Happy situation**: After we click on **`My Account`** link, the page gets loaded in time, and we are able to type on the email address text field.
- The other situations are explained as follows, where for any of the reasons mentioned, we are not able to complete the happy situation:
    - The page is not loaded in time, and we run out of time to perform the action. In attended mode, where a human is performing the action, can judge these using cognitive abilities and take appropriate action.
    - The page does get loaded, but the text field is not there.
    - The page does not get loaded, and there is an error that could be caused because of a client-side issue or server-side issue.
    - The page is not loaded as the internet stopped working.

A human tester, when performing the preceding steps in person, when encounters an unhappy situation, will be able to take appropriate action and make a decision whether what is being encountered should be logged as a defect, or should the person wait for more for the page to load, or report issue to Web administrator as the website did not work because of internet issues. But when a script takes the place of the human tester, we have to build certain cognitive abilities into it so that it can act appropriately in the preceding situations, as we would expect someone with cognitive abilities to do. Also, one of the approaches or it should be enforced here is that maybe the first and foremost thing to do is take care of the synchronization between these two independent processes.

There are two independent processes that take place as we use a tool to drive the browser and perform actions on the Web application. The tool Selenium used here will execute the command to drive the Web browser and perform an action on the Web application, and the Web application will execute those actions as per its processing speed. If we do not synchronize these two processes, our script will fail, and we will not be able to effectively and with reliability execute our scripts to test

the application. The reason for this would be that from the script, Selenium would try to identify the element on the Web page, but for various reasons listed in the unhappy situation, the object or the page itself may not be loaded and may cause inconsistent behavior with our scripts written to test the application.

In the following section, we will learn how we can implement synchronization in the script to ensure the two independent processes are aware of each other's existence.

Types of synchronization

We have learned so far the importance of synchronization. In this section, we will learn how we implement it in the test scripts and what the approaches are for it. There are two approaches as follows:

- Implicit wait
- Explicit wait

Implicit wait is also known as global wait. It is generally implemented immediately after the WebDriver object is initialized. Once implemented, this wait holds true for every statement in the script.

Explicit wait is known as a local wait. It is used in places in the script which may require more wait time. Generally, when we are using the explicit wait, we set the implicit wait to zero, else they may both interfere with each other. When we use explicit wait, we need to have a condition in which we will be waiting to be completed, or the wait will timeout in the specified number of seconds. To create a condition for the explicit wait, there is a class available in Selenium known as the **`ExpectedConditions`**. The class has a large number of implemented methods, which we can use to form a condition. We will know more about it in the section for explicit wait.

In the next section, we will see the implementation.

Scenario

To implement the implicit and explicit wait in action, we will pick up a scenario of login from the application using valid user information. Let us first understand the following steps for it.

Login

1. Open the application using the URL—**http://practice.bpbonline.com/**
2. Click on **`My Account`** link.
3. Provide username—**`admin@admin.com`**, and password—**`admin@123`**.

4. Click on **`Sign in`** button.
5. Click on **`Log off`** link
6. Click on **`Continue`** button

As per the preceding steps, we launch the application on a browser and click on the **`My Account`** link (as shown in the following figure).

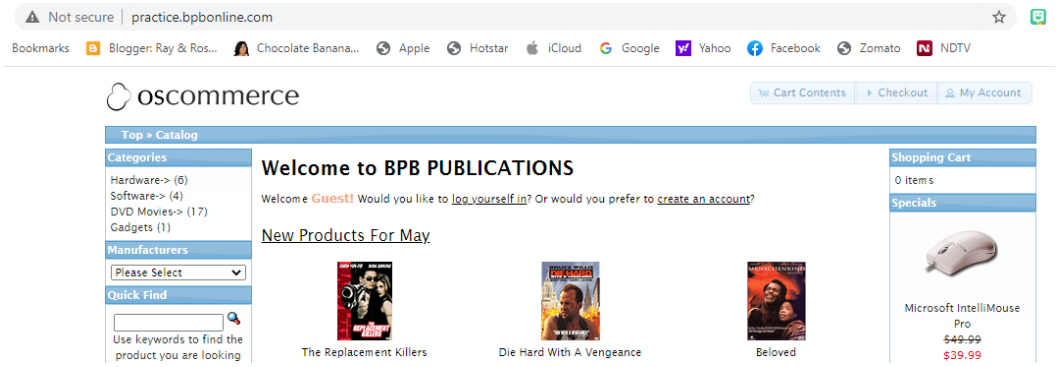


***Figure 4.3**: Home page of application*

We then provide a valid username and password information and then click on the **`Sign In`** button as shown in the following figure.

***Figure 4.4**: Login to the application*

If the information is correct, we will see the **`My Account Information`** page as shown in the following figure:

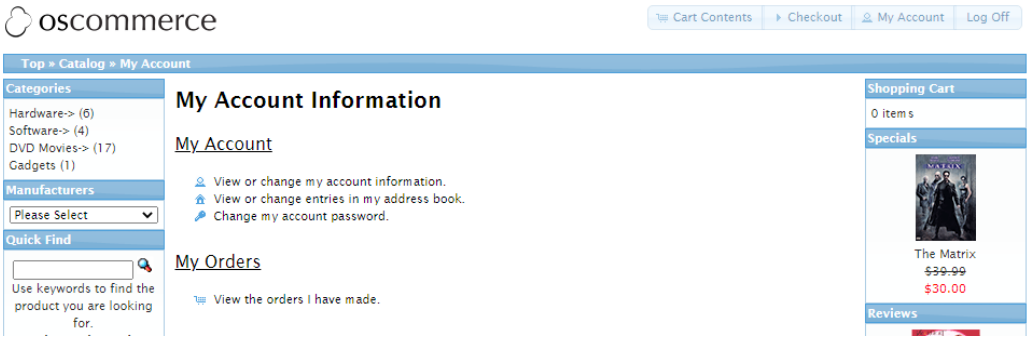


***Figure 4.5**: Page after Login*

Here, we then click on **`Log Off`** and continue button to come back to the home page:

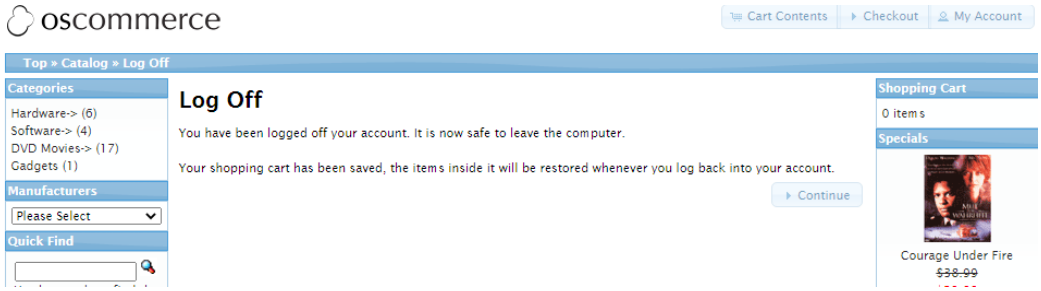


***Figure 4.6**: Logout page of the application*

In the preceding scenario, we need to notice the changes of the page in the application as we perform an action on different elements. For example, if we click on the **`My Account`** link, we go to the page where we need to fill in the information for username and password. And when we click the sign in button, we go to the **`My Account Information`** page, where we see the log-off link. We click on the log-off link and see the continue button, and after clicking it, we are back to the home page.

In each step, our script will have to be made smart to wait for an element to appear. If the element does not appear within a given time period; then, throw an exception of Time Out and end the execution. The first type of wait which we will see is implicit wait, which is introduced in the script and acts for every statement in the script. It is also known as a global wait. After that, we will see an explicit wait, which is known as local wait and is introduced at the script in a place, which requires more time to wait than the global time set. And as explained earlier, we do not use these waits together as they might interfere with each other. So, when we are using explicit wait, we set the implicit wait time to zero.

Implementing implicit wait

To implement the implicit wait, we use the following command:

```
//add implicit wait
```

driver.manage().timeouts().implicitlyWait(10, TimeUnit.SECONDS);

Here, the driver is an instance of the **`ChromeDriver`** and uses its method manage. The manage method has timeouts, where we chose the option of **`ImplicitlyWait`**. The arguments that we pass to this method are the time unit which we want to use and the quantity of that unit. So here, we choose to implicitly wait for the 10 seconds. This command which is used will work for every statement written in the script. It will wait for an element to be available for the most 10 seconds; if within 10 seconds, the element has appeared, it will perform the action and move on to the next statement in the script. If for some reason, the element is not available and 10 seconds have passed, this situation then leads to an exception of **`TimeOut`**.

Let us see the following login script using implicit wait:

```
package codefiles;

import java.util.concurrent.TimeUnit;
import org.openqa.selenium.*;
import org.openqa.selenium.chrome.*;

public class demoOfImplicitWait {

      public static void main(String[] args) throws Exception {

            //set system property
            System.setProperty("webdriver.chrome.driver",
"driverexes\\chromedriver.exe");

            //Initialize driver object
            WebDriver driver = new ChromeDriver();

            //add implicit wait
            driver.manage().timeouts().implicitlyWait(10,
TimeUnit.SECONDS);
            //login.
driver.get("http://www.practice.bpbonline.com/");
            driver.findElement(By.linkText("My Account")).click();
```

```
            driver.findElement(By.name("email_address")).
sendKeys("bpb@bpb.com");
                driver.findElement(By.name("password")).
sendKeys("bpb@123");
                driver.findElement(By.id("tdb1")).click();

                //logout
                driver.findElement(By.linkText("Log Off")).click();
                driver.findElement(By.linkText("Continue")).click();
                driver.close();
        }
}
```

Implementing explicit wait

We implement explicit wait or local wait in situations where a certain section of our script requires more time than the global timeout set. In these situations, it is not advisable to increase the entire global timeout because if for some reason the script starts failing, for every step, we will be waiting as per the duration set in the global timeout. We need to understand that there exists a thin line between synchronization and the speed of the script. And it takes a little time and a few executions to set up the balance of them.

To implement explicit wait, we will be using Selenium's **`WebDriverWait`** interface and the **`ExpectedConditions`** interface. Before we move further, let us first talk about these two entities.

The **`WebdriverWait`** interface provides us with a method until(). This method takes an argument of type **`ExpectedConditions`**. Expected conditions are a class of Selenium WebDriver, which provides us with a large number of methods to detect an event happening. For example, **`elementToBePresent`**, **`elementToBeClickable`**, **`alertWindowPresent`**, and so on and so forth.

Tip: More such information for ExpectedConditions methods are available in https://www.selenium.dev/selenium/docs/api/java/org/openqa/selenium/support/ui/ExpectedConditions.html.

The **`WebDriverWait`** interface takes an instance of the driver, a timeout, and an object of **`ExpectedConditions`**, as it gets implemented. Let us take a scenario where we want to implement a local wait or the explicit wait. We will take the login

scenario and put a **`WebDriverWait`** for the appearance of **`Log Off`** link as shown in the following script. Only if the **`Log Off`** link appears, we will go ahead and proceed with logout.

```
package codefiles;

import java.util.concurrent.TimeUnit;
import org.openqa.selenium.*;
import org.openqa.selenium.chrome.*;
import org.openqa.selenium.support.ui.ExpectedConditions;
import org.openqa.selenium.support.ui.WebDriverWait;

public class demoOfExplicitWait {

      public static void main(String[] args) throws Exception {

            //set system property
            System.setProperty("webdriver.chrome.driver",
"driverexes\\chromedriver.exe");

            //Initialize driver object
            WebDriver driver = new ChromeDriver();

            //add implicit wait
            driver.manage().timeouts().implicitlyWait(10, TimeUnit.
SECONDS);

            //login.
            driver.get("http://practice.bpbonline.com/");
            driver.findElement(By.linkText("My Account")).click();
            driver.findElement(By.name("email_address")).
sendKeys("bpb@bpb.com");
            driver.findElement(By.name("password")).
sendKeys("bpb@123");

            driver.findElement(By.id("tdb1")).click();
            driver.manage().timeouts().implicitlyWait(0, TimeUnit.
SECONDS);

            //explicit wait
            new WebDriverWait(driver, 10).until(ExpectedConditions.
presenceOfElementLocated(By.linkText("Log Off")));

            //logout
            driver.manage().timeouts().implicitlyWait(10, TimeUnit.
SECONDS);
            driver.findElement(By.linkText("Log Off")).click();
            driver.findElement(By.linkText("Continue")).click();
            driver.close();
      }
}
```

In the preceding code, we should also notice that before using **`WebDriverWait`**, we set the implicit wait to 0, and after explicit wait usage is over, we set it back to 10 seconds. We need to note here is that we do not use both implicit and explicit wait at the same time in the code. If we are implementing explicit wait, we make the implicit wait to 0. The reason for this is that these waits can interfere with each other and can cause run time execution issues in our code.

Encountering exceptions

As we use the wait commands of two kinds with Selenium, we can encounter the following exceptions:

`No Such Element Found`, and the **`TimeOutException`**. The **`NoSuchElementFound`** exception is seen when the script waits for an element to appear on the page with the given locator technique but is unable to find it within the given time duration. The **`TimeOutException`** is generated when the script is unable to find the element, with the given locator, it causes **`NoSuchElementFound`** exception, which further calls the **`TimeOutException`**, as we are unable to find the element in the given duration. So, the expected conditions method **`presenceOfElementLocatedBy`** (as shown in the following code) throws the **`NoSuchElementFound`** exception, which further causes the timeout exception to come.

To see this, we can deliberately try and find a link using the link text JUNK. Since junk does not exist, after 10 seconds of waiting using **`WebDriverWait`**, we will see the **`TimeOutException`**, which internally is generated because this element does not exist.

```
  //explicit wait
  new WebDriverWait(driver, 10).until(ExpectedConditions.
  presenceOfElementLocated(By.linkText("Junk")));
```

Exception raised as shown in the following script:

```
Exception in thread "main" org.openqa.selenium.TimeoutException:
Expected condition failed: waiting for presence of element located
by: By.linkText: Junk (tried for 10 second(s) with 500 milliseconds
interval)
      at org.openqa.selenium.support.ui.WebDriverWait.
timeoutException(WebDriverWait.java:95)
      at org.openqa.selenium.support.ui.FluentWait.until(FluentWait.
java:272)
      at codefiles.demoOfExplicitWait.main(demoOfExplicitWait.java:33)
Caused by: org.openqa.selenium.NoSuchElementException: no
such element: Unable to locate element: {"method":"link
text","selector":"Junk"}
  (Session info: chrome=90.0.4430.212)
```

Conclusion

Hence, we have learned in this chapter about the need for waits in our automation scripts, the types of waits in Selenium, and how we can implement it.

In the upcoming chapter, we will discuss about working with different types of HTML elements like form elements, Web tables, and dropdown elements.

Questions

1. Why do we need synchronization in our automation scripts?
2. Explain the meaning of implicit wait and how is it implemented?
3. What is ExpectedConditions class in Selenium?
4. Which exception comes when WebDriverWait fails?

Chapter 5
Working with WebElements—Form, Table, and Dropdown

In the earlier chapters, we discussed about the Selenium libraries, the methods available with them to handle browser, Web element, and create locators. We also discussed about how we can use the concept of waits to ensure reliability with our scripts, at the same time, taking care of the speed of the execution. In this chapter, we will discuss about working with different HTML elements, the type of action we perform on them during automation, and how Selenium allows us to work on them. We will discuss the form HTML elements, the Web table element, and the dropdown in Part 1.

Structure

The chapter is divided into the following sections:

- Working with form elements such as textbox, radio button, checkbox, and so on
- Working with the Web Table element
- Working with the Dropdown element

Objectives

After studying this chapter, you will be able to perform the following:

- Automate the handling of the form HTML elements
- Understand and work with the Web table elements
- Understand and work with the dropdown elements

Working with form elements

Form is an HTML tag, which contains different types of HTML elements. The form element is used in HTML websites to capture user input. The information is generally sent to the server for further work. The form element in HTML is represented as **`<form>,`** and it acts as a container for different elements such as text boxes, radio buttons, submit buttons, checkboxes, and so on. All these different elements are represented with the HTML tag **`<input>`**. The different types of input elements are available as follows:

Type	HTML information	Description
Text	`<input type="text">`	Displays a text input field.
Radio	`<input type="radio">`	Displays a radio button.
Checkbox	`<input type="checkbox">`	Displays a checkbox.
Submit	`<input type="submit">`	Displays a submit button.
Button	`<input type="button">`	Displays a button.

***Table 5.1**: Different types of input elements*

In our practice application, the scenario of change profile information is where we will see most of the input elements from *table 5.1*. Let us explore that section. For the change profile scenario, we will have to perform the following steps:

1. Open the application using URL: **http://practice.bpbonline.com/index.php**.
2. Click on the **`My Account`** link as shown in *figure 5.1*:

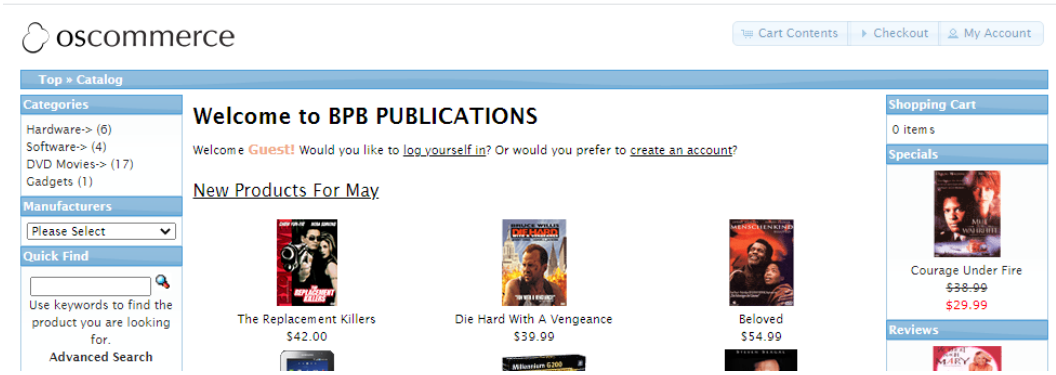


***Figure 5.1**: My Account link*

3. On the **`Sign In`** page, provide the login details: username is **`bpb@bpb.com`** and password is **`bpb@123`**. Click on the **`Sign In`** button as shown in *Figure 5.2*:

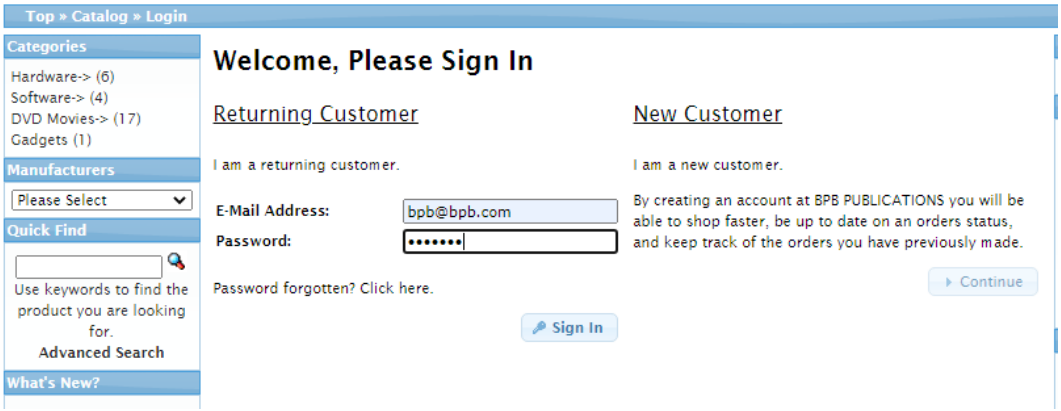


***Figure 5.2**: Sign In page*

In the preceding figure, we see the following input elements: text and password.

4. After signing in, you reach the **`My Account Information`** page. Here, we need to click on the link which says, **`View or change my account information`**.

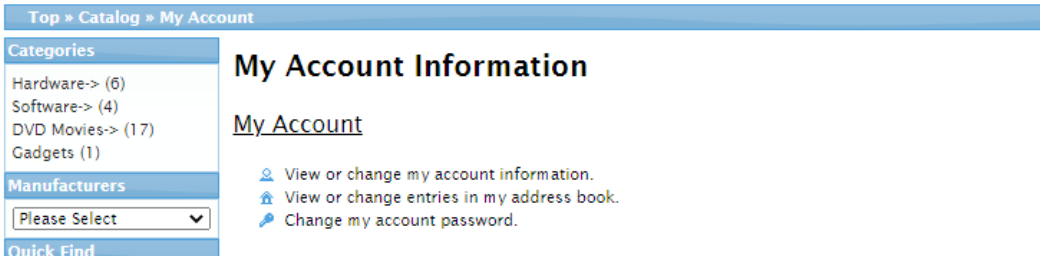


***Figure 5.3**: My Account Information page*

5. This will take us to the **`Change Profile`** page. On this page, we see the input element text and radio.

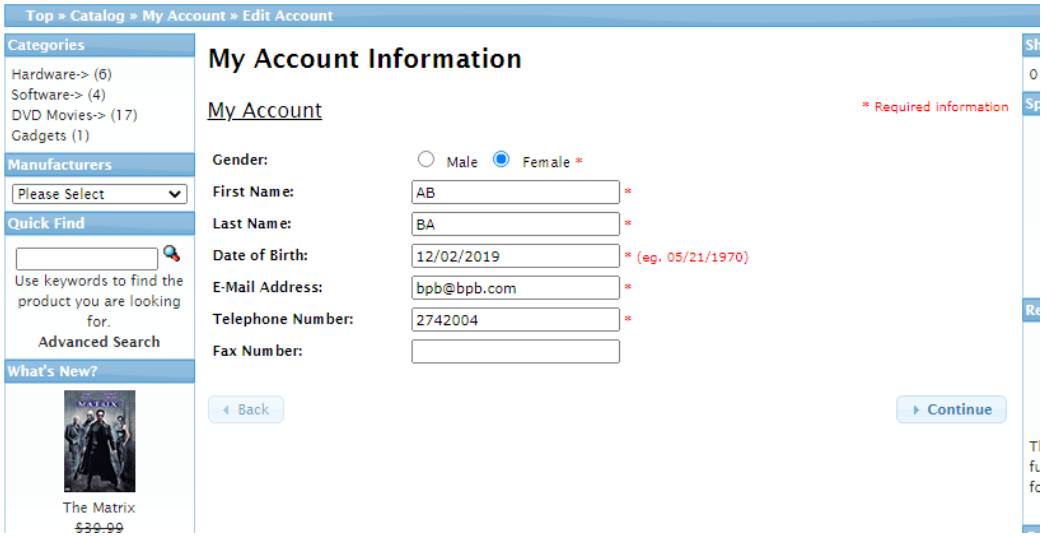


***Figure 5.4**: Change Profile page*

6. We can edit any information in the profile section; let us say we want to modify the **`Telephone Number`** and **`Gender`**. If the **`Male`** radio button is selected, we go ahead and click on **`Female`** radio button or vice versa. We will then clear the contents of the field and then type in the telephone number, and click on the **`Continue`** button for the change profile action to complete.

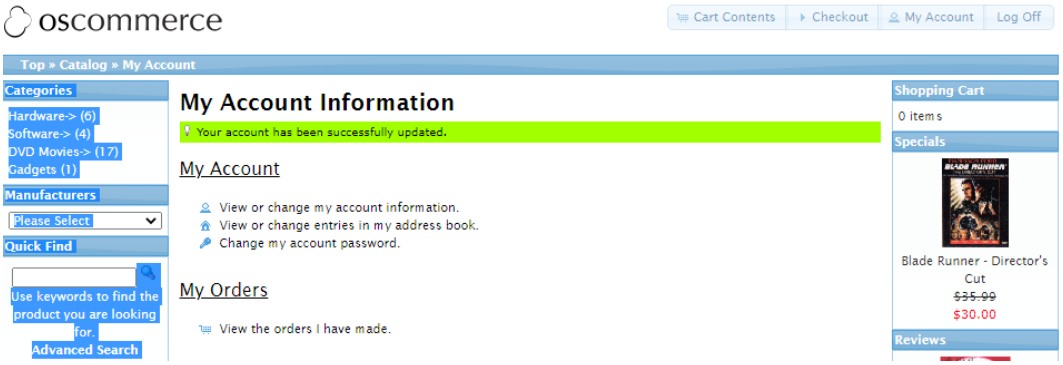


***Figure 5.5**: My Account Information page*

7. We can now click on the **`Log Off`** link and continue to log off from the account, and go back to the home page:

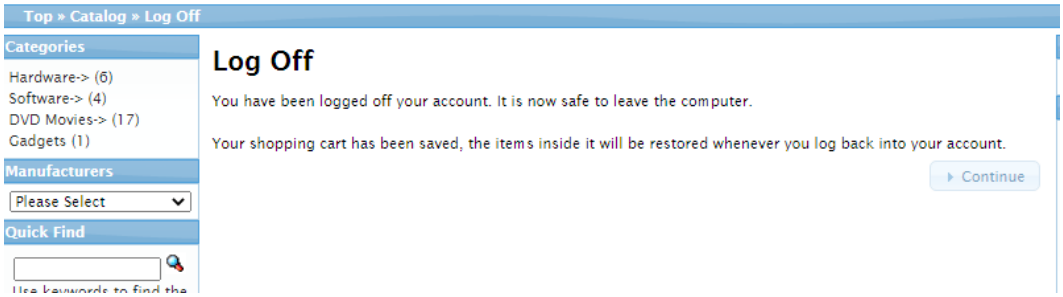


***Figure 5.6**: Log off link*

Please note that in all these steps, we are not adding any validations. Selenium library does not, by default, have any validation support, as that is not a browser action. To add validation to the preceding scenario and make it into a test case, we will have to use a unit testing framework. We can choose either JUnit or TestNG. In our upcoming chapters, we will be exploring TestNG as the unit test framework to be used while creating test automation scripts with Selenium using Java as a programming language. For the time being, we will simply use If condition statements to verify an action being completed and print a message on the console, as shown in the following script:

```
package codefiles;

import java.util.concurrent.TimeUnit;
import org.openqa.selenium.*;
import org.openqa.selenium.chrome.*;

public class handlingInputElements_changeProfile {

	public static void main(String[] args) throws Exception {

		//set system property
		System.setProperty("webdriver.chrome.driver", "driverexes\\
chromedriver.exe");

		//Initialize driver object
		WebDriver driver = new ChromeDriver();

		//add implicit wait
		driver.manage().timeouts().implicitlyWait(10, TimeUnit.SECONDS);

		//login.
		driver.get("http://www.practice.bpbonline.com/");
		driver.findElement(By.linkText("My Account")).click();
	    driver.findElement(By.name("email_address")).sendKeys("bpb@bpb.
com");
		driver.findElement(By.name("password")).sendKeys("bpb@123");
		driver.findElement(By.id("tdb1")).click();

		//change profile
		driver.findElement(By.linkText("View or change my account
information.")).click();
		//change gender-if male is selected, select female else
select male.
		if(driver.findElement(By.xpath("//input[@value='m']")).
isSelected()){
			driver.findElement(By.xpath("//input[@value='f']")).
click();
		}else {
			driver.findElement(By.xpath("//input[@value='m']")).
click();
		}
		//change phone
		driver.findElement(By.name("telephone")).clear();
		driver.findElement(By.name("telephone")).
sendKeys("23838393");
		driver.findElement(By.id("tdb5")).click(); //continue button
		if(driver.getPageSource().contains("account has been
successfully updated")) {
			System.out.println("Change profile successful");
		}else {
			System.out.println("Profile information not changed");
		}

		//logout
		driver.findElement(By.linkText("Log Off")).click();
		driver.findElement(By.linkText("Continue")).click();
		driver.close();
	}
}
```

As seen in the preceding script, we perform the change profile scenario by working with different types of input elements encountered in the logic action and change profile action. Next, we will look at how to work with the HTML Web Tables.

Working with Web Tables

The Web Tables in the HTML page are created using the **<table>** tag. It is a container tag, which means this HTML tag contains other HTML elements. The **<table>** tag contains other HTML elements which are the table row **<tr>** and each table row contains a table data **<td>**. The **<td>** is what we see as a table cell in the HTML page of the application. The table cell act like the body element of the HTML page, which means it can contain any HTML element within it. Sometimes a table tag can also contain a table body represented by **<tbody>** and the table heading tag, which is represented as **<th>**. More information on this can be read from this link—**https://www.w3schools.com/html/html_tables.asp**.

A Web table in HTML page looks like as follows:

```
<table id = table1>
  <tr>
    <td>1</td>
    <td>2</td>
    <td>3</td>
  </tr>
  <tr>
    <td>4</td>
    <td>5</td>
    <td>6</td>
  </tr>
</table>
```

This table on the Web page will appear as follows:

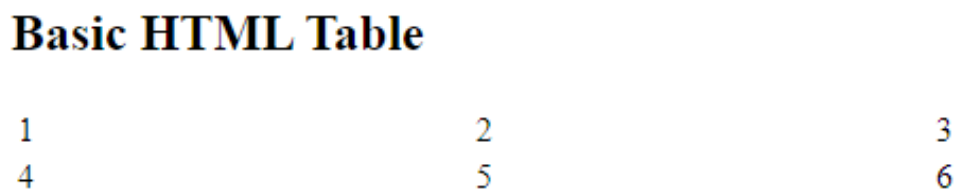


***Figure 5.7**: Output*

We find the Web Tables generally used in HTML pages to represent information in a tabular manner, which appears neat and organized. The Web Table can contain static information, or at the time of page rendering, it can pick up dynamic information. This depends on the Web application. In this chapter, we will see how we can iterate a Web table, and fetch information from each cell, and print that on the console. This is one of the examples to work with Web tables; there could very well be others. The idea is to understand how to handle them and then implement this depending on the requirement we have for automation.

For this, we will look at the home page of the application. We see the product listed as the following image shows:

***Figure 5.8**: The home page of the application*

If we look at this using the chrome inspect, we will see the table element which contains this information.

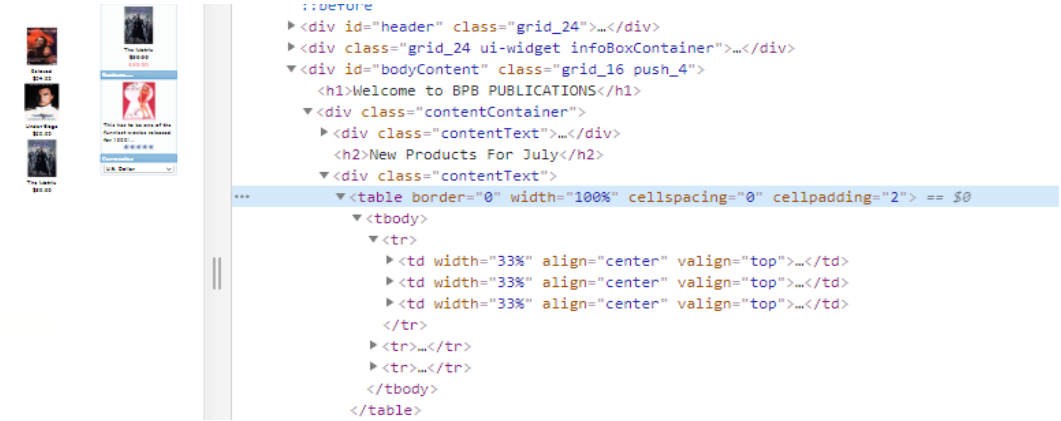


***Figure 5.9**: The Chrome inspects*

Let us look at the information contained in one table cell of this product table:

```
<td width="33%" align="center" valign="top">
  <a href="http://practice.bpbonline.com/product_info.php?products_id=4">
    <img src="images/dvd/replacement_killers.gif" alt="The Replacement Killers" title="The Replacement
    illers" width="100" height="80">
  </a>
  <br>
  <a href="http://practice.bpbonline.com/product_info.php?products_id=4">The Replacement Killers</a>
  <br>
  "$42.00"
</td>
```

***Figure 5.10**: HTML content of a table cell*

As we see in *figure 5.10*, the table cell contains two anchor tags and a text that shows the price of the product. One anchor tag displays the image of the product, and the other takes us to the product page. In the script, which we will write, we will iterate through this table and print the contents of the table. The steps which we will do in our script are as follows:

1. Open the Web application using the URL: **http://practice.bpbonline.com/index.php**.
2. Identify the table element using the HTML tag property as there is only one table on the page, and that does not contain any attribute which we can use to identify it.

```
<table border="0" width="100%" cellspacing="0" cellpadding="2">
  <tbody>
```

***Figure 5.11**: The table element*

3. Once we have the table element, we will find out how many tables row it contains by using the **`findElements`** method and look for the **`<tr>`** tag, and then, we will take each row and find out how many table data tags it contains, using the **`findElements`** method only.
4. Once we have found the table cell, we will print its text contents on the console.
5. Here, it is a good idea to note that the Web Table of the HTML page works as a two-dimensional array, and like it, we have rows and columns, and we need two loops to iterate through it.

Let us look at the following code now:

```
package codefiles;

import java.util.List;
import java.util.concurrent.TimeUnit;
import org.openqa.selenium.*;
import org.openqa.selenium.chrome.*;


public class workingWithwebTables_producttable {

	public static void main(String[] args) throws Exception {

		//set system property
		System.setProperty("webdriver.chrome.driver",
"driverexes\\chromedriver.exe");

		//Initialize driver object
		WebDriver driver = new ChromeDriver();

		//launch application
		driver.get("http://practice.bpbonline.com/index.php");

		//add implicit wait
		driver.manage().timeouts().implicitlyWait(10, TimeUnit.
SECONDS);


		//create the table element
		WebElement productTable = driver.findElement(By.
tagName("table"));

		//Fetch all table rows
		List<WebElement> rows = productTable.findElements(By.
xpath("//*/tbody/tr"));
```

```
            for(WebElement row : rows) {
                  //Fetch all table cols
                  List<WebElement> cols = row.findElements(By.
xpath("td"));
                  for(WebElement col: cols) {
                        //print cell content
                        String content = col.getText();
                        System.out.println(content);
                  }
            }

      }
}
```

The output of the preceding program prints information on the console, which is as follows:

```
The Replacement Killers
$42.00
Die Hard With A Vengeance
$39.99
Beloved
$54.99
Samsung Galaxy Tab
$749.99
Matrox G200 MMS
$299.99
Under Siege
$29.99
Speed
$39.99
Microsoft Internet Keyboard PS/2
$69.99
The Matrix
$30.00
```

***Figure 5.12**: Output of the program*

So, here we have seen how we can handle a Web Table and fetch contents present in the table cell. In the last section of this chapter, we will talk about the dropdown element.

Working with dropdown

The dropdown element is represented using the **<select>** tag. It is also a container tag and it contains **<option>** as the child tag. An example of the dropdown is shown as follows:

```
<select name="countries" >
  <option value="japan">JAP</option>
  <option value="India">IND</option>
  <option value="France">FRA</option>
  <option value="Australia">AUS</option>
</select>
```

***Figure 5.13**: Dropdown example*

When the preceding HTML renders on the Web page, it is displayed as follows:

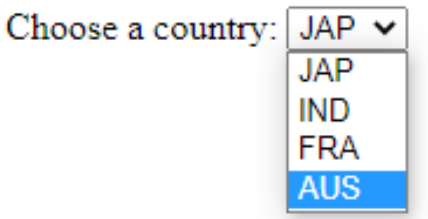


***Figure 5.14**: Dropdown HTML view*

To know more about the dropdown element, refer to the link—**https://www.w3schools.com/tags/tag_select.asp**.

The dropdown element in Selenium is not handled using the **WebElement** interface. Rather the dropdown element has its own separate class known as the Select class. Its details are available here: **https://www.selenium.dev/selenium/docs/api/java/org/openqa/selenium/support/ui/Select.html**

The reason for a separate class is that there are some very specific actions associated with the dropdown element, which are not available with the other Web elements. To understand these actions, we need to first understand the dropdown element HTML information. Let us look at the preceding dropdown HTML again:

```
<select name="countries" >
  <option value="japan">JAP</option>
  <option value="India">IND</option>
  <option value="France">FRA</option>
  <option value="Australia">AUS</option>
</select>
```

***Figure 5.15**: HTML content of dropdown*

As we see in this HTML code, the select tag is a parent tag inside which the child tag of option is available. The option tag contains a field called as **`value,`** and a text is associated with it. This is the text which we see on the Web page. For example, **`<option value="japan">JAP</option>`** code displays JAP on the Web page. But at the backend, this is the information associated with it. Keeping this in mind, we will not look at the different methods available to us with **`Select`** class and look at some of them at the code level.

Method	Description
`selectByValue()`	This will select the element from the dropdown based on the value attribute of the option tag.
`selectByVisibleText()`	This will allow us to select an element based on the visible text associated with the option tag.
`selectByIndex()`	This will allow us to select the element based on its position in the dropdown list.
`deselectByValue()`	This will deselect the element from the dropdown based on the value attribute of the option tag.
`deselectByVisibleText()`	This will allow us to deselect an element based on the visible text associated with the option tag.
`deselectByIndex()`	This will allow us to deselect the element based on its position in the dropdown list.
`getOptions()`	This method will return us all the options available in the dropdown list.
`getFirstSelectedOption()`	This method will return us the first selected option from the list.
`isMultiple()`	This method returns true or false based on whether the dropdown allows multiple selections of elements or not.

***Table 5.2**: Different methods available to us with select class*

In our application, we can find many dropdown lists available. We will pick the dropdown available on the main page, which displays the manufacturers. Let us have a look at it.

```
<div class="ui-widget-header infoBoxHeading">Manufacturers</div>
▼<div class="ui-widget-content infoBoxContents">
  ▼<form name="manufacturers" action="http://practice.bpbonline.com/index.php" method="get">
    ▼<select name="manufacturers_id" onchange="this.form.submit();" size="1" style="width: 100%"> == $0
      <option value selected="selected">Please Select</option>
      <option value="6">Canon</option>
      <option value="4">Fox</option>
      <option value="8">GT Interactive</option>
      <option value="9">Hewlett Packard</option>
      <option value="5">Logitech</option>
      <option value="1">Matrox</option>
      <option value="2">Microsoft</option>
      <option value="10">Samsung</option>
      <option value="7">Sierra</option>
      <option value="3">Warner</option>
    </select>
```

***Figure 5.16**: Dropdown available on the main page*

We can see in the preceding figure that this list is **`<select>`** element in the HTML, and it has various **`<option>`** tags available with it. Let us look at the complete HTML associated with it.

```
▼<select name="manufacturers_id" onchange="this.form.submit();" size="1" style="width: 100%"> == $0
  <option value>Please Select</option>
  <option value="6">Canon</option>
  <option value="4">Fox</option>
  <option value="8">GT Interactive</option>
  <option value="9">Hewlett Packard</option>
  <option value="5" selected="selected">Logitech</option>
  <option value="1">Matrox</option>
  <option value="2">Microsoft</option>
  <option value="10">Samsung</option>
  <option value="7">Sierra</option>
  <option value="3">Warner</option>
</select>
```

***Figure 5.17**: Select tag*

We see that this is not a multiple selection dropdown. It contains 11 option tags, where the first options tag represents **`Please Select`**. We should notice that the value attribute contains numerals, such as **"1"**, **"2"**, until **"10"**. And the display text associated with each option element is the manufacturer name that we see on the Web page. We will take the following scenario to create an automation script:

1. Launch the Web application using the URL: **http://practice.bpbonline.com/index.php.**
2. Find all the manufacturers from the dropdown table.
3. For each manufacturer found, select it from the dropdown and print the names of all products associated with it on the console.

4. In Step 3, we need to understand that the product page shows information in the form of a Web table; so, we will have to use the information from the Web table section discussed earlier to show how to iterate through the table and print the contents associated with it.

5. As we write this code, we will see with Selenium that when we move from one manufacturer to another, there is a page refresh happening. Due to this, we will be getting a **`StaleElementException`** in our code. To handle it, we will have to identify the dropdown element again.

6. To work with the dropdown element, we need to import the package: **`org.openqa.selenium.support.ui.Select`**.

Let us now look at the following code:

```
package codefiles;

import java.util.*;
import java.util.concurrent.TimeUnit;
import org.openqa.selenium.*;
import org.openqa.selenium.chrome.*;
import org.openqa.selenium.support.ui.Select;

public class workingWithdropdownelement_manufacturer {

      public static void main(String[] args) throws Exception {
            //set system property
            System.setProperty("webdriver.chrome.driver",
"driverexes\\chromedriver.exe");

            //Initialize driver object
            WebDriver driver = new ChromeDriver();

            //launch application
            driver.get("http://practice.bpbonline.com/index.php");

            //add implicit wait
            driver.manage().timeouts().implicitlyWait(10, TimeUnit.
SECONDS);
            //create the dropdown element
            Select manufacturers = new Select(driver.findElement(By.
xpath("//select[@name='manufacturers_id']")));
            //getting all option elements available within the
dropdown
```

```
            List<WebElement> allmanfs=manufacturers.getOptions();
            allmanfs.remove(0); //removing Please Select from the
list.

            //creating an arraylist for all manufacturers name
            ArrayList<String> allNames = new ArrayList<String>();
            for(WebElement man : allmanfs) {

                  allNames.add(man.getText());
            }
//selecting each manufacturer one by one
            for(String manname : allNames) {
                  //selecting element by its visible text
                  manufacturers.selectByVisibleText(manname);
                  //handling the stale element exception as page
refreshes.
                  manufacturers = new Select(driver.findElement(By.
xpath("//select[@name='manufacturers_id']")));
                  if(driver.getPageSource().contains("There are no
products available in this category.")) {
                        System.out.println("The manufacturer has
no products");
                  }else {
                        //create the table element
                        WebElement productTable = driver.
findElement(By.className("productListingHeader"));

                        //Fetch all table rows
                        System.out.println("\n\nThe manufacturer
- "+manname +" products are listed--");
                        List<WebElement> rows = productTable.
findElements(By.xpath("//*/tbody/tr"));
                        for(WebElement row : rows) {
                              //Fetch all table cols
                              List<WebElement> cols = row.
findElements(By.xpath("td"));
                              for(WebElement col: cols) {
                                    //print cell content
                                    String content = col.
getText();
                                    System.out.println(content);
                              }
                        }
                  }
```

```
            }
            driver.close();
        }
}
```

So as seen in the preceding code, where appropriate comments are also added for better understanding, we are able to print out the output with the manufacturer name and the products associated with them. The output is as follows:

```
The manufacturer Canon has no products

The manufacturer - Fox products are listed--
Product Name+ Price Buy Now
Courage Under Fire $38.99  $29.99
Buy Now
Die Hard With A Vengeance $39.99
Buy Now
Speed $39.99
Buy Now
Speed 2: Cruise Control $42.00
Buy Now
There's Something About Mary $49.99
Buy Now


The manufacturer - GT Interactive products are listed--
Product Name+ Price Buy Now
Disciples: Sacred Lands $90.00
Buy Now
The Wheel Of Time $99.99
Buy Now
Unreal Tournament $89.99
Buy Now


The manufacturer - Hewlett Packard products are listed--
Product Name+ Price Buy Now
Hewlett Packard LaserJet 1100Xi $499.99
Buy Now
The manufacturer Logitech has no products


The manufacturer - Matrox products are listed--
Product Name+ Price Buy Now
```

```
Matrox G200 MMS $299.99
Buy Now
Matrox G400 32MB $499.99
Buy Now
The manufacturer - Microsoft products are listed--
Product Name+ Price Buy Now
Microsoft IntelliMouse Explorer $64.95
Buy Now
Microsoft Internet Keyboard PS/2 $69.99
Buy Now
The Replacement Killers $42.00
Buy Now

The manufacturer - Samsung products are listed--
Product Name+ Price Buy Now
Samsung Galaxy Tab $749.99
Buy Now

The manufacturer - Sierra products are listed--
Product Name+ Price Buy Now
SWAT 3: Close Quarters Battle $79.99
Buy Now

The manufacturer - Warner products are listed--
Product Name+ Price Buy Now
A Bug's Life $35.99
Buy Now
Beloved $54.99
Buy Now
Blade Runner - Director's Cut $35.99  $30.00
Buy Now
Fire Down Below $29.99
Buy Now
Frantic $35.00
Buy Now
Lethal Weapon $34.99
Buy Now
Microsoft IntelliMouse Pro $49.99  $39.99
Buy Now
Red Corner $32.00
Buy Now
The Matrix $39.99  $30.00
```

```
Buy Now
Under Siege $29.99
Buy Now
Under Siege 2 - Dark Territory $29.99
Buy Now
You've Got Mail $34.99
Buy Now
```

We can try more functions associated with the dropdown class depending upon the scenario we have in hand to automate.

Conclusion

In this chapter, we have learned to handle the form elements, the Web table element, and the dropdown element. In our upcoming chapter, we will learn to handle the frame, alert and how to work with advance keyboard and mouse actions.

Questions

1. Which function of WebElement will we use to find out if the radio button is selected or not?
2. Is Web table a container element?
3. List the three different methods to select an element in a dropdown list?
4. When do we get StaleElementException?

Chapter 6
Working with WebElement—Alert, Frame, IFrame, and Window

In the previous chapter, we saw handling of the different types of the HTML elements, such as the input elements, the Web table, and the dropdown element. In this chapter, we will learn about how we can automate the handling of the JavaScript alerts, also known as popups, the Frame element, the IFrame element, the HTML window, and the action class, which is used to handle advance mouse and keyboard actions. To understand the concepts of this chapter, we will have to explore another Web application, which is hosted on this link—**https://the-internet.herokuapp.com/**. This website is maintained by Elemental Selenium and widely used by the Selenium community to try and understand the working of the different types of HTML elements using Selenium. Some of our examples will be based on it.

Structure

The chapter is divided into the following sections:

- Working with JavaScript alerts
- Working with Frame and IFrame HTML element
- Working with HTML window
- Working with action class

Objectives

After reading this chapter, you will be able to:

- Use the **`Alert`** class in Selenium to handle the popups.
- Work with the IFrame element, and handle objects which are inside an IFrame.
- Automate scenarios when a new window opens in the application.

Working with JavaScript alerts

The JavaScript alerts or popups are found in various Web application scenarios. These popups can provide information or request a user action. Until we handle the popup, we cannot proceed with any other action on our Web application. There are three kinds of JavaScript alerts are as follows:

- Alert box
- Confirm box
- Prompt box

An alert box is used to provide information to the user, and the user generally handles it by closing or clicking on the **`OK`** button.

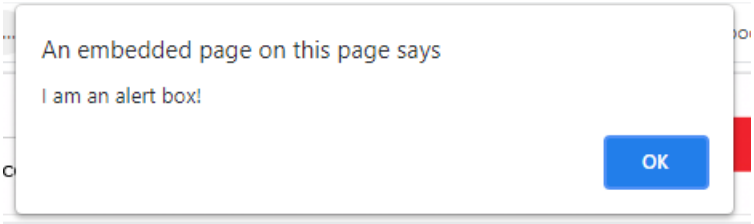


Figure 6.1: *Alert box*

The confirm box is used for the user to verify an action or deny it. It generally has an **`OK`** or a **`Cancel`** button. And the user can handle it by clicking on either.

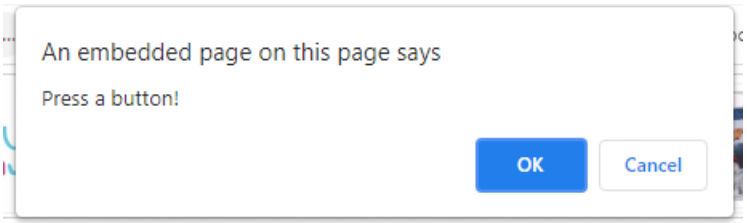


Figure 6.2*: Confirm box*

Finally, a prompt box is used in the situation when we want some information to be provided by the user. The user has to input something and then can either click on **`OK`** or on the **`Cancel`** button.

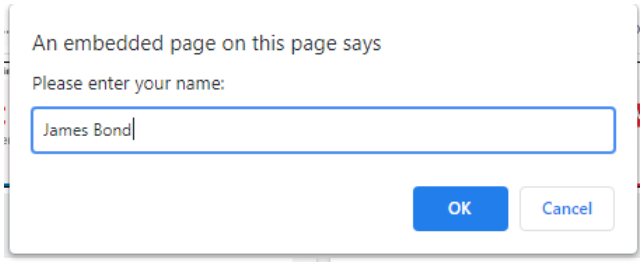


***Figure 6.3**: Prompt box*

To handle these alerts, there is a separate interface called as **`Alert`** in the Selenium java bindings. The details are available at: **https://www.selenium.dev/selenium/docs/api/java/org/openqa/selenium/Alert.html**

The **`Alert`** interface has four methods, as shown in the following table:

Method name	Action
`Accept`	This method will accept the alert. It will cause click on `OK` button
`Dismiss`	This method will cancel the alert. It will cause click on the `Cancel` button
`getText`	This method will fetch the text displayed on the `Alert`.
`sendKeys`	This method will send the keys which we press on the keyboard to the `Alert`.

***Table 6.1**: Methods of alert interface*

To understand the handling of the alert, we will take help from the internet Heroku application and see them working. The website is **https://the-internet.herokuapp.com/javascript_alerts**. On this, we will find all three types of alerts available.

To handle the first type alert box, we will perform the following actions:

1. Launch the application on Chrome using the URL: **https://the-internet.herokuapp.com/javascript_alerts**

2. Click on the button shown in the figure:

***Figure 6.4**: Confirm box type of pop up*

3. Handle the alert which pops up:

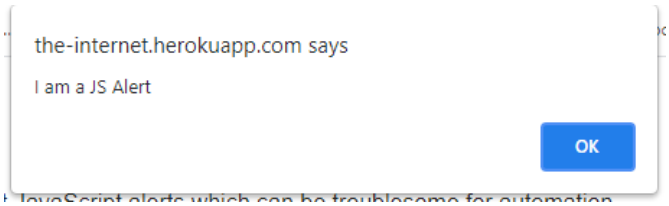


***Figure 6.5:** the first type Alert box*

4. If the alert is handled, we will get a success message on the Web page:

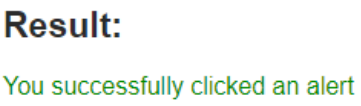


***Figure 6.6**: Result message on the page*

Let us see the following code for the preceding set of steps to work with the alert box type of popup.

```
Package codefiles;
import org.openqa.selenium.*;
import org.openqa.selenium.chrome.*;

public class demoOfAlertBox {

      public static void main(String[] args) throws Exception {

            //set system property
            System.setProperty("webdriver.chrome.driver",
"driverexes\\chromedriver.exe");
            //Initialize driver object
            WebDriver driver = new ChromeDriver();

            //open chrome browser with the url.

            driver.get("https://the-internet.herokuapp.com/
javascript_alerts");
```

```
            driver.findElement(By.xpath("//button[normalize-
space()='Click for JS Alert']")).click();
            Thread.sleep(2000); //wait for alert to appear
            Alert alertBox= driver.switchTo().alert(); //accept the
alert
            alertBox.accept();
            if(driver.findElement(By.xpath("//p[@id='result']")).
getText().contains("You successfully clicked an alert")){

                  System.out.println("Alert was handled");
            }else {
                  System.out.println("Alert was not handled");

            }
            driver.close();
      }

}
```

Let us now work with the **Confirm** box type of popup. The following steps will be done for the **Confirm** box:

1. Launch the Web application on Chrome using the URL: **https://the-internet.herokuapp.com/javascript_alerts**
2. Click on the second button:

***Figure 6.7**: Confirm box type of pop up*

3. On this, let us dismiss the alert by clicking on the **Cancel** button.

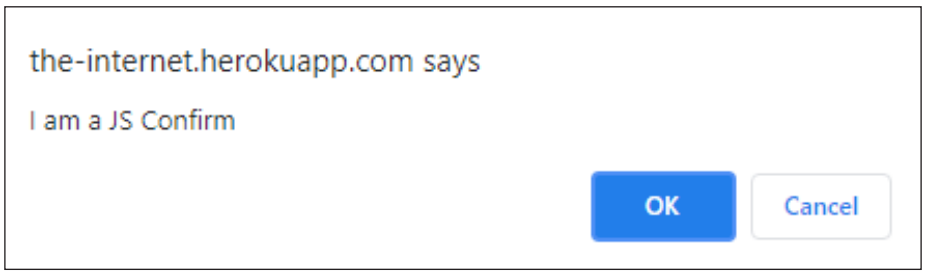


***Figure 6.8**: Dismiss the Alert box*

4. Our Web page will show the following message if the alert was dismissed:

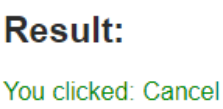


***Figure 6.9**: Message obtained on dismissing the Alert box*

Let us see the script to implement the preceding set of statements as code:

```
package codefiles;
import org.openqa.selenium.*;
import org.openqa.selenium.chrome.*;

public class demoOfConfirmBox {

	public static void main(String[] args) throws Exception {

		//set system property
		System.setProperty("webdriver.chrome.driver",
"driverexes\\chromedriver.exe");
		//Initialize driver object
		WebDriver driver = new ChromeDriver();
		//open chrome browser with the url.
		driver.get("https://the-internet.herokuapp.com/
javascript_alerts");
		driver.findElement(By.xpath("//button[normalize-
space()='Click for JS Confirm']")).click();
		Thread.sleep(2000); //wait for alert to appear
		Alert alertBox= driver.switchTo().alert(); //accept the
alert
		alertBox.dismiss();  //dismissing the alert
		if(driver.findElement(By.xpath("//p[@id='result']")).
getText().contains("Cancel")){

			System.out.println("Alert was handled");
		}else {
			System.out.println("Alert was not handled");

		}
		driver.close();
	}

}
```

In the final type of alert handling, we will work with the Prompt Box. For this, we will perform the following steps:

1. Launch the Web application using the URL: **https://the-internet.herokuapp.com/javascript_alerts**
2. Click on the third button:

***Figure 6.10**: Final type of alert handling*

3. We will here fetch the text associated with the alert, and send the text **`Hello how are you`** to it.

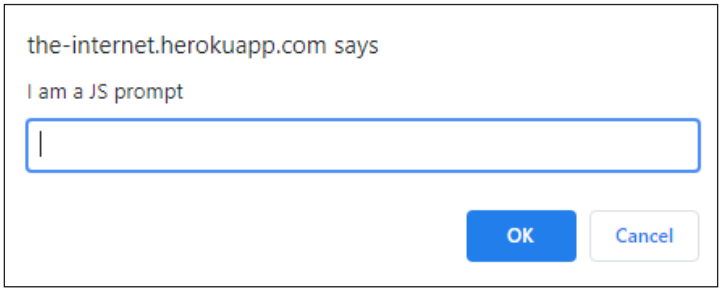


***Figure 6.11**: Fetching the text associated with the alert*

4. We finally accept the alert, and see the message sent to the prompt box on the Web page once the alert is closed.

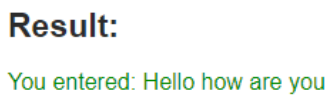


***Figure 6.12**: Result obtained*

Let us write the code to handle the preceding steps:

```
package codefiles;

import org.openqa.selenium.*;
import org.openqa.selenium.chrome.*;


public class demoOfPromptBox {

	public static void main(String[] args) throws Exception {

		//set system property
		System.setProperty("webdriver.chrome.driver",
"driverexes\\chromedriver.exe");
		//Initialize driver object
		WebDriver driver = new ChromeDriver();
		//open chrome browser with the url.
		driver.get("https://the-internet.herokuapp.com/
javascript_alerts");
		driver.findElement(By.xpath("//button[normalize-
space()='Click for JS Prompt']")).click();
		Thread.sleep(2000); //wait for alert to appear
		Alert alertBox= driver.switchTo().alert(); //accept the
alert
		String textOnAlert = alertBox.getText(); //fetch the
text from alert
		System.out.println(textOnAlert);
		alertBox.sendKeys("Hello how are you"); //sending keys
to alert
		alertBox.accept();
		Thread.sleep(2000);
		if(driver.findElement(By.xpath("//p[@id='result']")).
getText().contains("are you")){

			System.out.println("Alert was handled");
		}else {
			System.out.println("Alert was not handled");

		}

		driver.close();
	}

}
```

In all the preceding codes, where we handle the alert, we see there is a function from the driver which we use called as **`switchTo().alert`**. The **`switchTo()`** is a function associated with the **`Webdriver`** object, and it allows us to switch our focus to an alert, a frame, or a new window. Here, we have used it for the alert.

We have seen what are the different types of Java script alerts, also known as popups, and how we can handle them using the Alert interface of Selenium. In the next section, we will talk about working with Frame and IFrame.

Working with Frame and IFrame

The frame object in the HTML allows us to divide the HTML window into multiple sections, where each section can have a Web page within itself. The frame tag is no longer supported by HTML 5; instead, we will use the IFrame tag. An IFrame tag is called as an inline frame, and it allows us to embed another HTML page in the current document. An example from the HTML perspective is as follows:

```
<!DOCTYPE html>
<html>
<body>

<h2>Demo of IFRAME</h2>
<p>A page in another web page</p>

<iframe src="demo_iframe.htm" height="300" width="200" title="demoiframe">
</iframe>

</body>
</html>
```

In the Web browser, the Web page will look as follows:

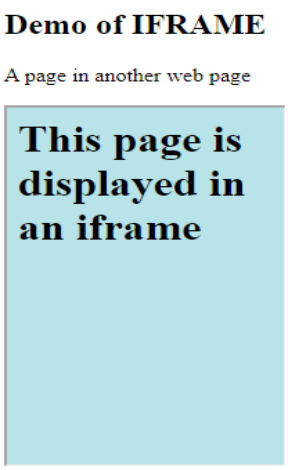


***Figure 6.13**: Web page showing IFrame*

If we have an HTML page, which contains an IFrame, and we are interested in working with a Web element that is on the HTML page inside the IFrame; then, we will first have to switch our focus using the **`driver.switchTo()`** command to the IFrame. Once our driver is pointing to the IFrame, we can now access and work with the Web elements of the Web page inside the IFrame. To show this, we will take an example of the IFrame available at the internet Heroku application. The URL of the application is: **https://the-internet.herokuapp.com/iframe**.

The scenario that we decide to automate here is described as follows:

1. Open the application using the URL: **https://the-internet.herokuapp.com/iframe**
2. Once the page is launched, fetch the contents displayed in the WYSIWYG editor displayed on the page

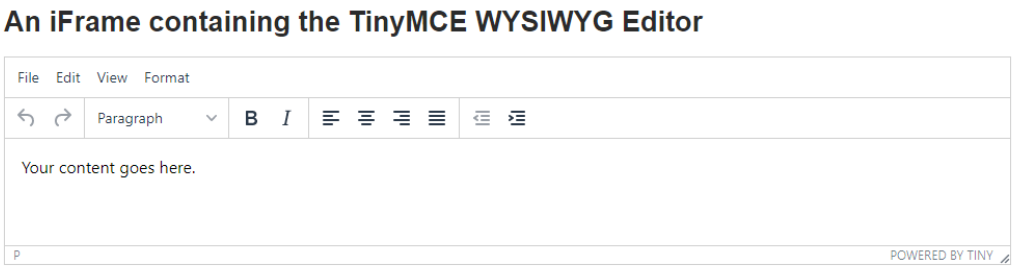


***Figure 6.14**: IFrame with WYSIWYG editor*

3. Print it on console

When we try and inspect the WYSIWYG editor on the page to see the backend HTML, we find that this is inside an IFrame tag, which looks as follows:

```
<html class="no-js" lang="en">
  <!--<![endif]-->
▶ <head>…</head>
▼ <body>
  ▶ <div class="row">…</div>
  ▼ <div class="row">
      ::before
    ▶ <a href="https://github.com/tourdedave/the-internet">…</a>
    ▼ <div id="content" class="large-12 columns">
        <!-- CDN hosted by Cachefly -->
        <script src="https://cdn.tiny.cloud/1/mcq6n68fc0vsuywr7nx7awf11byv91g6ihxw5okkzj8aykz6/tinymce
        5/tinymce.min.js" referrerpolicy="origin"></script>
      ▶ <script>…</script>
      ▼ <div class="example">
          <h3>An iFrame containing the TinyMCE WYSIWYG Editor</h3>
          <textarea id="mce_0" aria-hidden="true" style="display: none;">Your content goes here.
          </textarea>
        ▼ <div role="application" class="tox tox-tinymce" aria-disabled="false" style="visibility: hidd
        n; height: 200px;"> flex
          ▼ <div class="tox-editor-container"> flex
            ▶ <div data-alloy-vertical-dir="toptobottom" class="tox-editor-header">…</div>
            ▼ <div class="tox-sidebar-wrap"> flex
              ▼ <div class="tox-edit-area"> flex
                ▼ <iframe id="mce_0_ifr" frameborder="0" allowtransparency="true" title="Rich Text Area
                Press ALT-0 for help." class="tox-edit-area__iframe"> == $0
                  ▼ #document
                      <!DOCTYPE html>
                    ▼ <html>
                      ▶ <head>…</head>
                      ▶ <body id="tinymce" class="mce-content-body " data-id="mce_0" contenteditable="tr
                        e" spellcheck="false">…</body>
```

***Figure 6.15**: HTML associated with IFrame page*

So, to fetch the contents from the editor, we will first have to switch the focus to the IFrame, and only then the driver object will be able to access the Web elements present on the Web page, which is inside the IFrame. The code will look as follows:

```
package codefiles;

import org.openqa.selenium.*;
import org.openqa.selenium.chrome.*;

public class demoOfIFrame {

	public static void main(String[] args) throws Exception {

		//set system property
		System.setProperty("webdriver.chrome.driver",
"driverexes\\chromedriver.exe");
		//Initialize driver object
		WebDriver driver = new ChromeDriver();
		//open chrome browser with the url.
		driver.get("https://the-internet.herokuapp.com/iframe");
		//switch the focus to the iframe, we identify in here
the iframe using its id property
		driver.switchTo().frame(driver.findElement(By.id("mce_0_
ifr")));
		//Your content goes here is associated with the paragraph
html element
		String contents = driver.findElement(By.xpath("//p")).
getText();
		System.out.println(contents);
		driver.switchTo().defaultContent(); //switch back to the
parent html document
		driver.close();
	}

}
```

In the preceding code, we will see that we used the IFrame id property to identify it. We then used the driver **`switchTo`** command to shift our focus on the Web page available inside the IFrame. Once the driver points to it, we can then access the "p" HTML element text and display it on the console. Once our work is done, we switch the focus back to the main HTML page using the method: **`switchTo().defaultContent()`**.

Thus, we conclude the working with IFrame element and the Web page in it using Selenium.

Working with HTML window

Many times, we come across applications, which on launching open a new page, like an eCommerce website opens an advertisement page on a new window or a new tab. To handle this situation using automation, we will again have to use the driver **`swtichTo`** function. This function allows to switch the focus of the WebDriver object to the new window or tab. We can then work on the page of the new window and then close it and come back to the main website.

To work with the windows, we need to understand that every window is associated with a unique window handle. This window handle is used to identify it and associate the driver object with it. To work with all windows opened with the same driver instance, we can store them in a collection called a set. A set is a collection in Java that allows to store unique data values. We then use an Iterator to iterate through all values in the set. Let us see an application that allows us to open a new window.

The scenario which we will automate is as follows:

1. Open the application using the URL: **https://the-internet.herokuapp.com/windows**
2. After that, we click on the link as shown in the following image:

***Figure 6.16**: Link to be clicked*

3. When we click on **`Click Here`**, it will open a new tab with a new page which looks as follows:

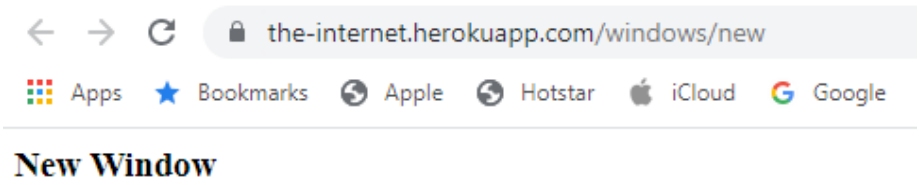


***Figure 6.17**: New Window launched*

4. We will verify if the new page contains the text **`New Window`**.

5. We will now close the window and go back to the main parent window and close it also.

The code for the preceding steps is as follows:

```
package codefiles;

import java.util.Set;
import org.openqa.selenium.*;
import org.openqa.selenium.chrome.*;

public class demoOfWindow {

	public static void main(String[] args) throws Exception {

		//set system property
		System.setProperty("webdriver.chrome.driver",
"driverexes\\chromedriver.exe");
		//Initialize driver object
		WebDriver driver = new ChromeDriver();
		//open chrome browser with the url.
		driver.get("https://the-internet.herokuapp.com/windows");
		//Click on Click Here
		driver.findElement(By.linkText("Click Here")).click();
		Set<String> allWindowHandles = driver.
getWindowHandles();
		for(String handle: allWindowHandles) {
			System.out.println(handle);
		}
		//convert the set into an array so that we can access
the window.
		//at array 0 position, the main window handle will be
there, and after that based on when a new window is openned
		//it will take the next position in the array
		Object[] windows= allWindowHandles.toArray();
		//Switch to new window
		driver.switchTo().window(windows[1].toString());
		if(driver.getPageSource().contains("New Window")) {
			System.out.println("we are now on new window");
			System.out.println("closing the child window");
			driver.close();
		}
		//switch to the main window
		System.out.println("closing the main window");
		driver.switchTo().window(windows[0].toString());
		driver.close();

	}

}
```

We can see in the code that once we have fetched in a set all the window handles, we are using the for each loop to print the window handles on the console. After that, we convert the set to an Object Array. Here, we need to realize that the first element in the array will be the handle of the parent window. The next element in the array will be the next window which is launched, and so on. Since for our scenario, there are only two windows, at 0 index we have the handle of the parent window, and at index 1 we have the handle of the child window.

We then use the **`driver.switchTo().window`** command to switch to the child window by passing the second index of the array. We then find if it contains a required text and closes it. We then switch to the parent window using the first element of the array and close it too.

Thus, in the preceding code, we have seen how we can work with windows in Selenium.

Conclusion

In this chapter, we have seen how we can work with HTML elements which are container elements. The container HTML elements are those which contain other HTML elements. We saw how we can work with Web tables, dropdown elements. We also saw the usage of the **`switchTo`** method associated with the webdriver, which allows us to switch the focus to a new alert, an IFrame, or a new window. Overall, with the lessons learnt in *Chapters 8* and *9*, we should be able to handle the different types of HTML elements we can see in a Web application. In the upcoming chapter, we will get the concepts on **`Action`** class, Screenshot capture, WebDriver manager, and so on.

Questions

1. Which class do we use to work with dropdown elements?
2. Which method of alert is same as clicking on the OK button?
3. What is a window handle?
4. What is the purpose of an IFrame in a Web page?

Chapter 7

Extra Concepts—Actions, Screenshot, WebDriverManager

Introduction

In this chapter, we will discuss about a few extra concepts available in Selenium. We will talk about the Action class, which allows us to handle advance mouse and keyboard actions. We will see how we can capture screenshots during execution. We will also talk about a recently introduced third-party package in Selenium called as `WebDriverManager`, which comes in handy while working with Selenium to fetch at turn time the correct browser driver for the browser chosen for execution.

Structure

In this chapter, we will discuss the following topics:

- Actions
- Screenshot
- WebDriverManager

Objectives

The objective of this chapter is to a few extra concepts while trying to automate Web applications when using Selenium. After this chapter, you will be able to:

- Automate advance keyboard and mouse operations using the **`Action`** class.
- Able to capture screenshots at the time of execution for better reporting.
- Finally, use the concept of WebDriver manager to take care of the driver browser interaction for our automation scripts.

Actions

While working with Web applications, we may come across scenarios where we need to perform some advance user gestures using keyboard and mouse. Like drag and drop, mouse over, key up and key down, and so on. To perform these actions, we should use the action interface available with Selenium. The interface is available in the interactions package—**org.openqa.selenium.interactions**. It also helps us to create a composite action, which could be a combination of two or more actions to be performed together. There are various methods available with the **`Action`** class; more details can be found on this link—**https://www.selenium.dev/selenium/docs/api/java/org/openqa/selenium/interactions/Actions.html**.

We will see the different methods which are available with the action class, along with some code examples in *table 7.1*.

Methods of action class

The various methods of the action class the following are shared the list of some commonly used. For more details, please refer to the link shared in the previous paragraph.

Method name	Description
`click()`	Performs click at the current mouse location. It is generally used when we are creating a composite action. It has a `click(WebElement ele)` method variant which clicks on the given Web element.
`clickAndHold()`	Performs a click operation at the current mouse position and holds it. It has a `clickAndHold(WebElement ele)` method variant, which clicks on a given Web element and holds it.
`doubleClick()`	Performs double click at the given mouse position. It has a `doubleClick(WebElement ele)` method variant, which performs a double click action on the method.
`contextClick()`	This method performs right-click at the given mouse position. They are also known as `contextClick`. It has a method variant that takes `WebElement` as an input argument. `contextClick(WebElement ele)`, this will perform `contextClick` at the Web element.

`dragAndDrop()`	This method takes as argument a source element and a target element. It drags the source element to the target element. This method is represented as `dragAndDrop(WebElement source, WebElement target)`. This method also has a variant, which takes as an input argument the position to which the element needs to be dragged. `dragAndDrop(WebElement src, int offsetX, int offsetY).`
`moveToElement()`	This method takes as argument a `WebElement`, and moves the mouse to the middle of the element. Its syntax is `moveToElement(WebElement target)`. This method has a variant `moveToElement(WebElement target, int offsetX, int offsetY)`. This method moves to offset with respect to the element's viewpoint center.
`build()`	This method takes all the previous actions and combines them to create a composite action, which will then be performed.
`perform()`	This method is used to finally perform the action or the composite action created.

***Table 7.1**: Methods of the action class*

We will now see some code example samples for a few of the preceding action methods listed.

The first example we will see is drag and drop. To see the example, we will take the available website URL: **https://jqueryui.com/droppable/** as we can see in the website based on the following figure.

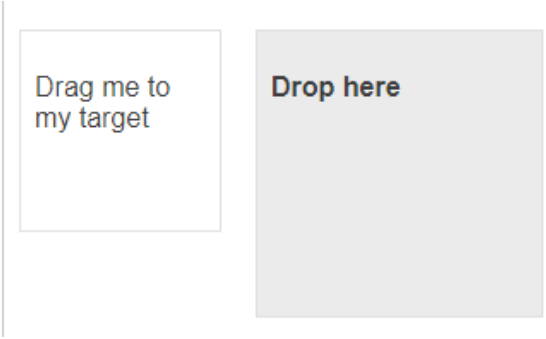


***Figure 7.1**: Drag and drop example*

We will have to drag the left box and put it on the right box. After we have performed this action, we will see the following figure:

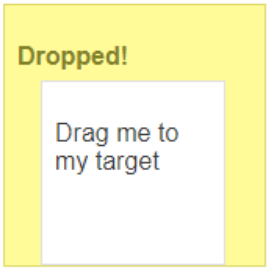


***Figure 7.2**: Output obtained by dragging the left box on the right box*

To automate the previous action, we will need the locators for both objects. As per the HTML information available on this page, we will see that these objects can be recognized using the available id property.

```
▼<div id="draggable" class="ui-widget-content ui-draggable ui-draggable-handle" style="position: relative;
">
    <p>Drag me to my target</p> == $0
  </div>
▼<div id="droppable" class="ui-widget-header ui-droppable">
    <p>Drop here</p>
  </div>
```

***Figure 7.3**: ID property*

We need to also realize that these objects are available inside an iframe; the HTML is as follows:

```
▼<iframe src="/resources/demos/droppable/default.html" class="demo-frame">
  ▼#document
    <!DOCTYPE html>
  ▼<html lang="en">
    ▶<head>…</head>
    ▼<body>
      ▶<div id="draggable" class="ui-widget-content ui-draggable ui-draggable-handle" style="position: relative;
      ">…</div>
      ▶<div id="droppable" class="ui-widget-header ui-droppable">…</div>
      </body>
    </html>
</iframe>
```

***Figure 7.4**: Objects available inside an iframe*

Let us write the steps to automate the previous scenario, and for this, we will use the following action class.

1. Launch the application using URL: **https://jqueryui.com/droppable/**.

2. Scroll the page down using the **`JavaScriptExecutor`** class available with Selenium. This class allows to use of a **`javascript`** command and execute it directly during the execution run of the browser with Selenium.
3. Switch to the iframe which contains the objects.
4. Create an action class object, and call the method **`draganddrop`**, pass the target element, which is the box on the left, and the source element, which is the box on the right.
5. Call the method to perform on the preceding **`draganddrop`** method to complete the action.
6. Close the application.

The code for the preceding steps are as follows:

```
package codefiles;
import java.util.concurrent.TimeUnit;

import org.openqa.selenium.*;
import org.openqa.selenium.chrome.*;
import  org.openqa.selenium.interactions.*;

public class demoDragAndDropAction {

      public static void main(String[] args) throws Exception {
            //set system property
            System.setProperty("webdriver.chrome.driver",
"driverexes\\chromedriver.exe");
            //Initialize driver object
            WebDriver driver = new ChromeDriver();
            driver.manage().timeouts().implicitlyWait(10,
TimeUnit.SECONDS);
            driver.manage().window().maximize();
            //open chrome browser with the url.
            driver.get("https://jqueryui.com/droppable/");
            //to perform scroll on an application using Selenium
            JavascriptExecutor js = (JavascriptExecutor) driver;
            js.executeScript("window.scrollBy(0,document.body.
scrollHeight)");
            //switch to the iframe
            driver.switchTo().frame(driver.findElement(By.
className("demo-frame")));
```

```
//source element
                WebElement drag=driver.findElement(By.
id("draggable"));
                //target element
                WebElement drop=driver.findElement(By.
id("droppable"));
                //Create action object
                Actions act=new Actions(driver);
                //call the drag drop action
                act.dragAndDrop(drag, drop).perform();
                //wait for 5 seconds before closing the browser
                Thread.sleep(5000);
                //close the browser
                driver.close();

        }

        private static Actions Actions(WebDriver driver) {
                // TODO Auto-generated method stub
                return null;
        }

}
```

In the next example, we will pick where we have to perform a composite action. Composite action is one where more than one action is clubbed together and performed using the build method first. For this example, we will see the Web URL available at—**https://the-internet.herokuapp.com/key_presses**.

On this page, there is a textbox on which we can press keys like Tab, Space, and as we do that, the key that we press gets displayed on the page. In the script, we will first press the Tab key, wait for 5 seconds, and then press the Space key. We will first have to create a composite action using the build **`method()`** and then call perform. The script for the same are as follows:

```
package codefiles;
import java.util.concurrent.TimeUnit;

import org.openqa.selenium.*;
import org.openqa.selenium.chrome.*;
import  org.openqa.selenium.interactions.*;

public class demoCompositeAction {

      public static void main(String[] args) throws Exception {
            //set system property
            System.setProperty("webdriver.chrome.driver",
"driverexes\\chromedriver.exe");
            //Initialize driver object
            WebDriver driver = new ChromeDriver();
            driver.manage().timeouts().implicitlyWait(10, TimeUnit.
SECONDS);
            driver.manage().window().maximize();
            //open chrome browser with the url.
            driver.get("https://the-internet.herokuapp.com/key_
presses");
            WebElement textBox=driver.findElement(By.id("target"));
            //Create action object
            Actions act=new Actions(driver);
            //press tab key first and then wait for 5 seconds, and
then press space
            act.sendKeys(textBox, Keys.TAB).pause(5000).
sendKeys(textBox,Keys.SPACE).build().perform();
            //wait for 5 seconds before closing the browser
            Thread.sleep(5000);
            //close the browser
            driver.close();
      }
}
```

Thus, we saw the different action methods and also saw an example of composite action.

Screenshot

The screenshot interface in Selenium allows us to capture the screen at the time of script execution and store it in different ways. Screenshots are a great way to capture issues while script execution. They help with logging better bug reports so

that developers can easily identify by looking at the image where the issue might have been. To use the screenshot method **`getScreenShotAs`**, which stores file using the file class. This class is available in the commons.io jar package, which is available by Apache. We will need to add this package to our project so as to use the method.

To show the code example for the screenshot, we will be taking the scenario for user login and logout using an external data file. The file which we will use is a CSV file, which stores data using a comma separator. The file will contain two records, one with valid user credentials and the other with invalid user credentials. We will capture the screenshot for user login for both valid scenarios and invalid scenarios to log the working of the application.

Our steps will be as follows:

1. Create a CSV file with two records, one with valid user credentials and the other with invalid user credentials; the file is saved in the **`DataFiles`** folder in the project with the extension of **`.csv`**.

```
bpb@bpb.com,bpb@123
junk@bpb.com,junkpwd
```

***Figure 7.5**: Records with user credentials*

2. We then write the script to automate the steps to login and logout of the application are as follows:
    a. Open the application.
    b. Click **`My Account`** link.
    c. Type username, using the data read from CSV file.
    d. Type password, using the data read from the CSV file.
    e. Click **`Sign In`**.
    f. If the user logs in successfully, capture the screenshot and then click on log off and continue the link.
    g. If the user does not log in successfully, then capture the screenshot and close the application.
3. To capture the screenshot, we will call the **`getScreenShot`** method of the screenshot interface. This interface will be initialized using the browser object created. We will store it as a **`File`** object and then convert that file into a **`.jpg`**. The class we use for it is the **`FileUtils`** class in the Commons IO jar package.

4. The screenshots will be stored in the **`Screenshots`** folder, which we need to create prior to our project. So our project tree looks like the following figure:

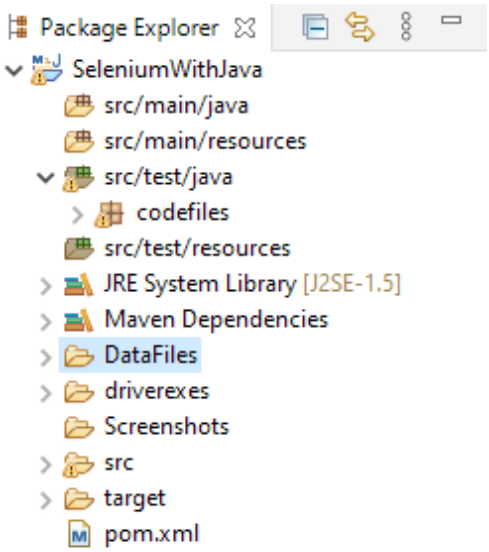


***Figure 7.6**: Project tree*

The following code shows the script which is generated from the preceding steps written:

```
package codefiles;
import org.openqa.selenium.*;
import org.openqa.selenium.chrome.*;
import java.util.concurrent.TimeUnit;
import java.io.*;
import org.apache.commons.io.FileUtils;

public class demoOfScreenShot {

      public static void main(String[] args) throws Exception {
            //Read the contents of the csv file
            FileReader readerObj = new FileReader("DataFiles\\
logininformation.csv"); //stream object
            BufferedReader bufReader = new
BufferedReader(readerObj); //input stream
            String line=bufReader.readLine(); //fetch the first
line of the file.
```

```
            //set system property
            System.setProperty("webdriver.chrome.driver",
"driverexes\\chromedriver.exe");
            //Initialize driver object
            WebDriver driver = new ChromeDriver();
            //add implicit wait
            driver.manage().timeouts().implicitlyWait(10,
TimeUnit.SECONDS);
            //login.
            driver.get("http://www.practice.bpbonline.com/");
            //Reading the files line by line until we reach end of
file
            while(line != null) {
                  //spliting the line into two sub strings using
comma as seperator.
                  //at array position 0 will be username, and at
1 it will be password
                  String[] loginDetails=line.split(",");
                  driver.findElement(By.linkText("My Account")).
click();
                  driver.findElement(By.name("email_address")).
sendKeys(loginDetails[0]);
                  driver.findElement(By.name("password")).
sendKeys(loginDetails[1]);
                  driver.findElement(By.id("tdb1")).click();
                  if(driver.getPageSource().contains("My Account
Information")) {
                        File scrnsht =  ((TakesScreenshot)
driver).getScreenshotAs(OutputType.FILE);
                        String fname=
"Screenshots\\"+loginDetails[0]+".jpg";
                        FileUtils.copyFile(scrnsht, new
File(fname));
                        //log off action
                        driver.findElement(By.linkText("Log
Off")).click();
                        driver.findElement(By.
linkText("Continue")).click();
                  }else {
/if user is not valid, then just capture the screenshot
                        File scrnsht =  ((TakesScreenshot)
driver).getScreenshotAs(OutputType.FILE);
```

```
                        String fname=
"Screenshots\\"+loginDetails[0]+".jpg";
                        FileUtils.copyFile(scrnsht, new
File(fname));
                    }
                    //read the next line of the file.
                    line=bufReader.readLine();
                }
                //close the browser
                driver.close();
        }
}
```

The previous code will execute two times for the valid user credential and the invalid user credential. After the execution is complete, we can find in the Screenshots folder two images captured for valid and invalid users. Let us see the screenshots captured here.

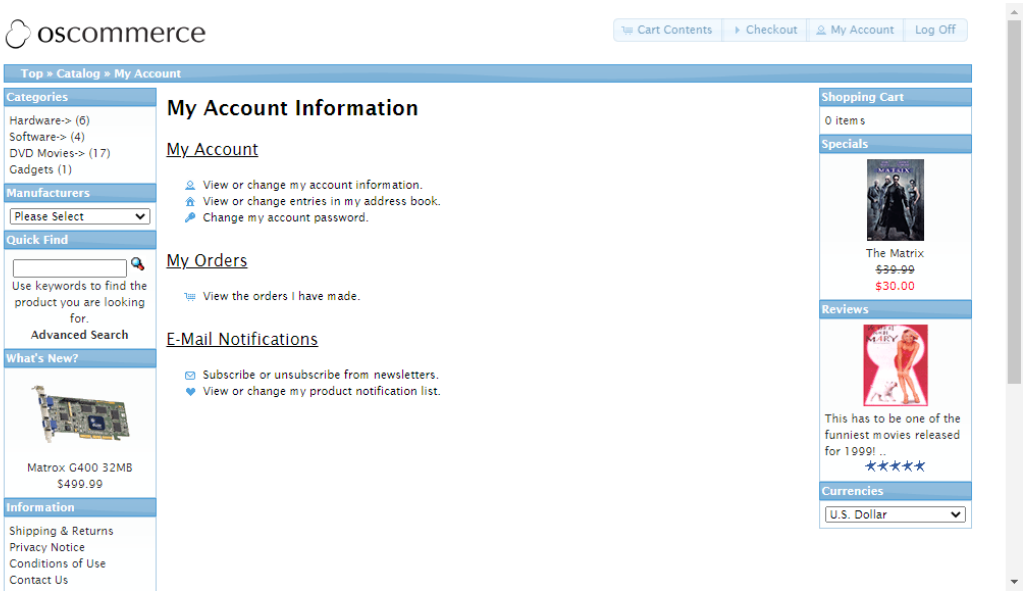


***Figure 7.7**: Valid user*

The figure shown here is during the invalid user credential flow:

***Figure 7.8**: Invalid user*

As mentioned previously, screenshot capturing during execution helps us log effective and detailed issues if found any. This helps the team members to fix the issues with the help of visual descriptions captured and reported in times of error.

WebDriverManager

The **`WebDriverManager`** allows us to manage at runtime the driver executable, which will be required by the browser for automation with Selenium. Using this package, we do not have to download and keep track of the driver versions with respect to the browser version we have on the system where test automation execution is going to take place. Using this library, the tasks to manage the drivers for the respective browser as per their type and version get simplified.

This library was developed by Boni Garcia, **https://github.com/bonigarcia**, and is maintained by a group of contributors. More details about it are available here—**https://github.com/bonigarcia/webdrivermanager**.

In this section, we will write down a sample code, which uses the **`WebDriverManager`** to launch and work with the browser. We will write the steps to launch the three different browsers in three scripts and then wait for 5 seconds and close them. The browsers we will choose are Chrome, Firefox, and Internet Explorer.

Launch Chrome using **`WebdriverManager`** as shown here:

```
package codefiles;
import io.github.bonigarcia.wdm.*;
import org.openqa.selenium.*;
import org.openqa.selenium.chrome.*;

public class demoWebDriverManagerChrome {

	public static void main(String[] args) throws Exception {
		//set up chromedriver with webdrivermanager
		WebDriverManager.chromedriver().setup();
		//create instance of chrome
		WebDriver driver= new ChromeDriver();
		driver.get("http://www.5elementslearning.dev/demosite");
		Thread.sleep(2000);
		driver.close();
	}
}
```

Launch Firefox using **`WebDriverManager`** as shown here:

```
package codefiles;
import io.github.bonigarcia.wdm.*;
import org.openqa.selenium.*;
import org.openqa.selenium.firefox.*;

public class demoWebDriverManagerFirefox {

	public static void main(String[] args) throws Exception {
		//set up firefox driver[geckodriver] with webdrivermanager
		WebDriverManager.firefoxdriver().setup();
		//create instance of firefox
		WebDriver driver= new FirefoxDriver();
		driver.get("http://www.5elementslearning.dev/demosite");
		Thread.sleep(2000);
		driver.close();
	}
}
```

The last we see is launching Internet Explorer using **`WebdriverManager`** as shown here:

```
package codefiles;
import io.github.bonigarcia.wdm.*;
import org.openqa.selenium.*;
import org.openqa.selenium.ie.*;

public class demoWebDriverManagerIE {

	public static void main(String[] args) throws Exception {
		//set up iedriver with webdrivermanager
		WebDriverManager.iedriver().setup();
		//create instance of ie
		WebDriver driver= new InternetExplorerDriver();
		driver.get("http://www.5elementslearning.dev/demosite");
		Thread.sleep(2000);
		driver.close();
	}
}
```

Thus, we see in this section how we can use the **`WebDriverManager`** to work with different browsers. We also saw that we did not have to download or manage the drivers for these browsers.

Conclusion

In this chapter, we learn about a few extra concepts which we can use with Selenium to help us write better scripts. We saw the usage action class, which helps us work with mouse and keyboard actions. We saw how we can screenshot at run time, save it as a file in the system. Finally, we saw a third-party useful library called as **`WebDriverManage,`** which eases the task to manage the respective driver for the browser we wish to use for execution by making available the correct driver for it at run time. In the upcoming chapter, we will look at the concept of a unit testing framework called as TestNG, which is extremely helpful in designing test automation scripts using Selenium.

Questions

1. Which package contains the action interface?
2. What does build() method does in the action interface?
3. Which method is used to capture the screenshot?
4. What is the use of the WebDriverManager?

Chapter 8
What is TestNG

TestNG stands for **Testing Next Generation**. It is one of the most widely used testing frameworks, specifically in test automation projects made using Java as a programming language. The other test framework many uses is JUnit, but TestNG is more compatible for test projects. The main website for TestNG is **https://testng.org/doc/**. In this chapter, we will explore the basics of TestNG and see some of its functionalities with respect to being used in test automation projects with Selenium.

Structure

In this chapter, the following topics will be covered:

- Designing a TestNG test
- Adding assertions to TestNG test
- Reports in TestNG

Objectives

The objective of this chapter is to understand the feature and functionalities of TestNG as a test framework to be used in designing test automation projects using Selenium. TestNG is one of the most widely used units test frameworks besides

JUnit. Although JUnit is mainly used by developers as they design their code, TestNG has features that are more apt for the testers to use as they design their code for automation. We will see what a TestNG suite looks like, discuss about various annotations available with TestNG, understand how we can execute a suite file, and discuss the different modes available for execution. Finally, we will explore reports which are made available by default with TestNG.

Introduction

TestNG is a testing framework that is used for a broad range of test activities, from unit testing to integration testing. It has various features and a wide range of annotations, which are useful to design a test suite. Any test suite designed in TestNG has an XML file that contains information about the tests, execution criteria and can also contain test configuration parameters information to be used by the test during execution. Let us see now how can we install TestNG in the eclipse environment.

Installation

In this section, we will cover the TestNG installation in the eclipse environment. There are various ways to install TestNG, which are mentioned in this link: **https://testng.org/doc/download.html**. We will see how we can install it using the Help of the Eclipse IDE. Let us go step by step.

1. Open the Java project in eclipse, where we have been working in the previous chapters. On the main menu, click on **`Help`** and select **`Install New Software...`**.

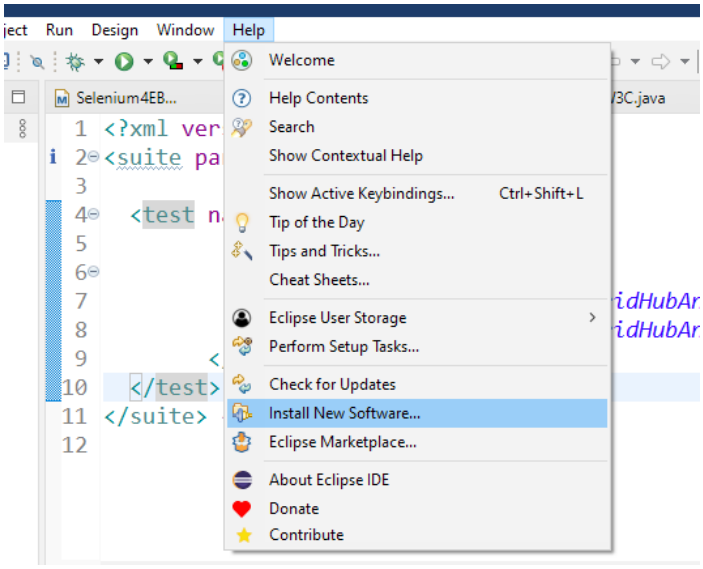


***Figure 8.1**: Installing TestNG*

2. Once the Install window opens, there we need to type in a URL in the **`Work with`** a text box. The URL to provide is **https://testng.org/testng-eclipse-update-site**, and press *Enter* or click on the **`Add...`** button:

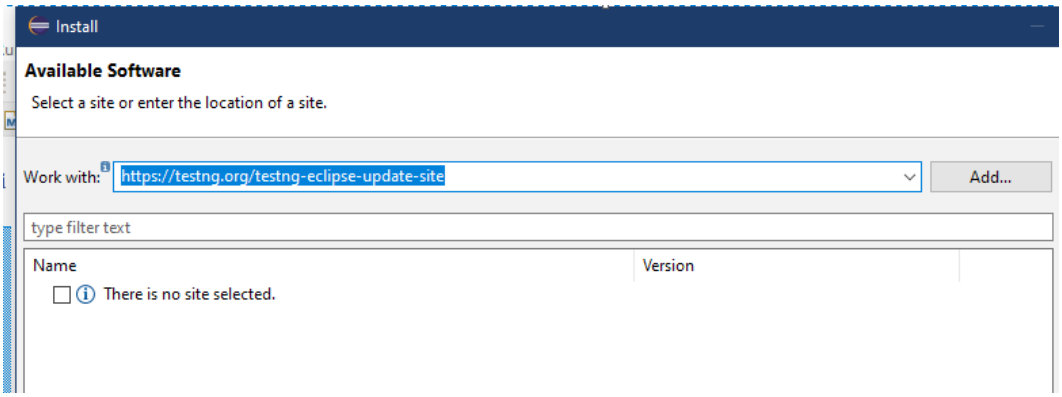


***Figure 8.2**: Provide URL for TestNG*

3. Once we do that, we will see the latest version of the TestNG, which is available. At the time of writing the book, the version, which is available, is 7.4.

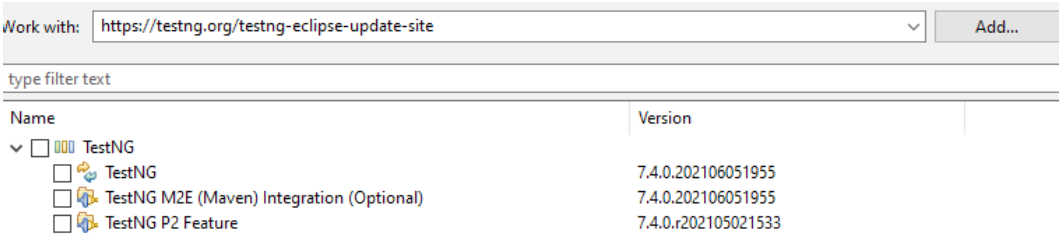


***Figure 8.3**: Select TestNG only from the list*

4. Select the first option, and click on **`Next`** button to proceed with the installation. It takes a while as it calculates the various dependencies.

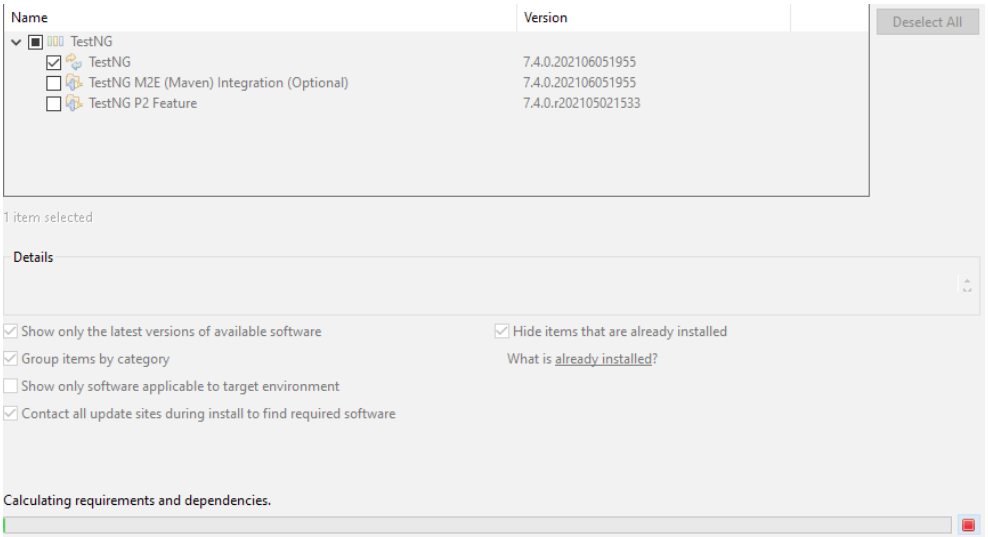


***Figure 8.4**: Installation begins*

5. Once done, it will require you to accept a license agreement. Accept and proceed to complete the installation.

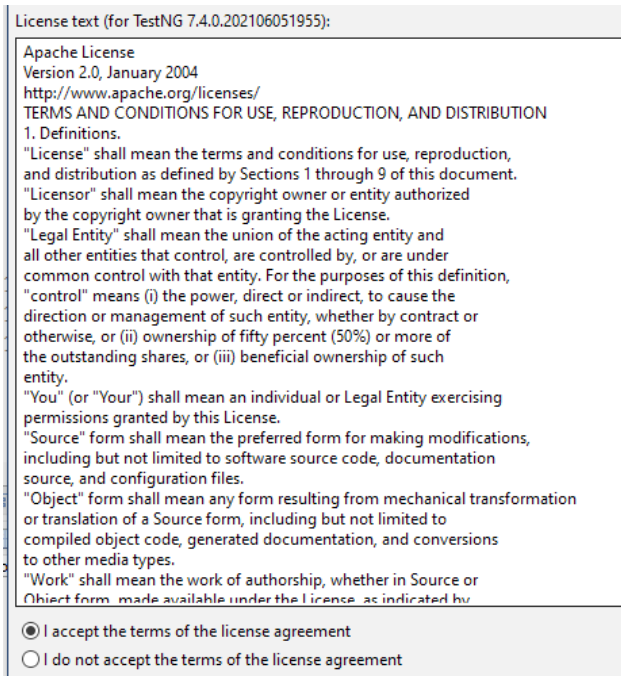


***Figure 8.5**: Accept the license agreement*

6. Once TestNG is installed on the eclipse, it will require a restart of an eclipse. After doing it, we need to add the TestNG library to the Java project. For this, right-click on the project and select **`Properties`**. Select **`Java Build Path`** option.

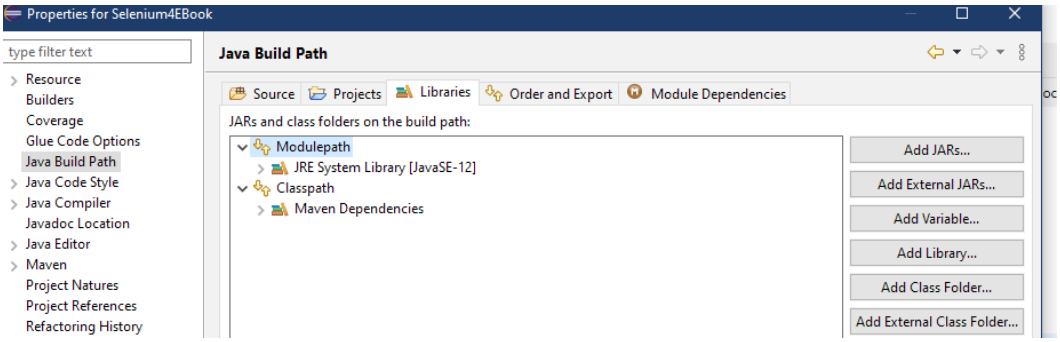


***Figure 8.6**: Java build path*

7. Click on **`Modulepath`**, and select **`Add Library`**. Select the TestNG from the list of libraries shown.

8. Once TestNG is added, it will show as an option generally just above the properties when we right-click the project. We can select it to create a TestNG class.

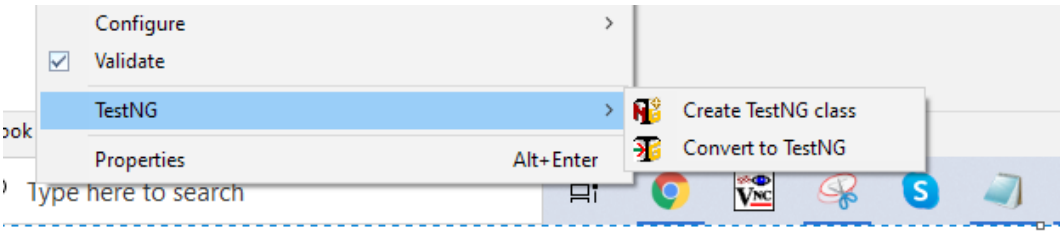


***Figure 8.7**: Creating TestNG class*

9. Select the option to **`Create TestNG class`**; it will show us a window where we need to select the annotation, we would need in our file and name the TestNG suite XML file.

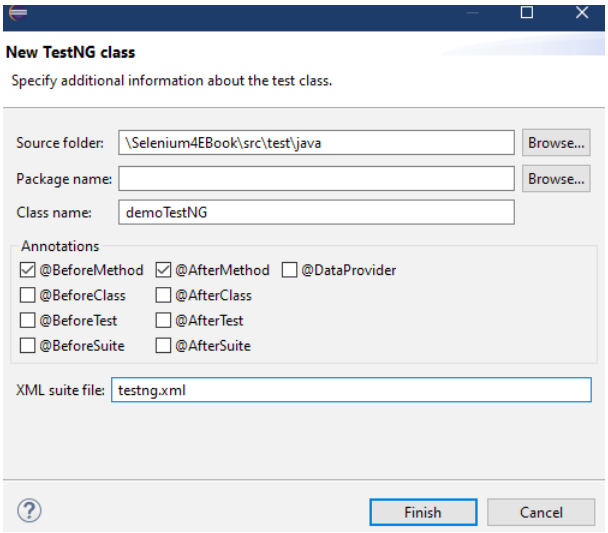


***Figure 8.8**: Creating TestNG XML*

10. As it can be seen from the preceding image, we provide the class name as **`demoTestNG`**, select the **`@BeforeMethod`**, and **`@AfterMethod`**. We also name the suite file as **`testng.xml`**. On clicking **`Finish`**, we will file a Java file created with the TestNG structure using annotations. And a **`testng.xml`** file is available as well.

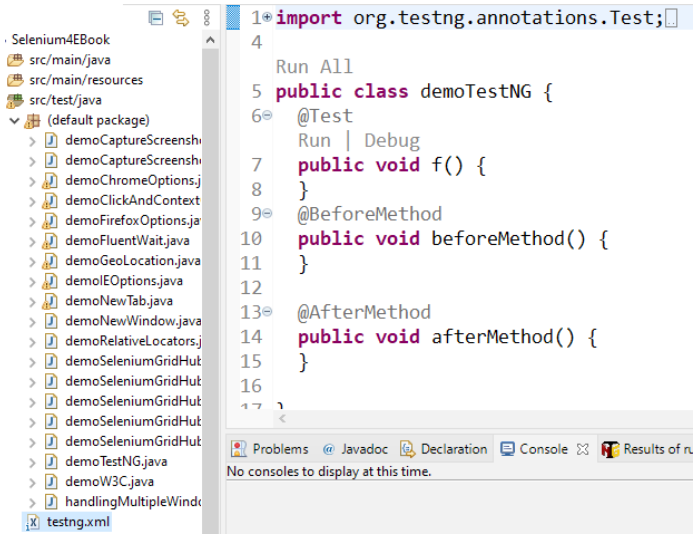


***Figure 8.9**: TestNG class*

11. We are now ready to add the code into the preceding file and execute it as a TestNG suite using the **`testng.xml`** file. But before we do that, let us first understand the structure by understanding the various annotations available in TestNG and also the suite XML file.

In the next section, we will learn about the different annotations TestNG has and the structure **`testng`** suite file has.

Structure

A TestNG suite consists of an XML file and at least one TestNG class file, which is a Java file containing a **`@test`** annotation that is used by a method in the program. Let us first discuss the different types of annotation which are available with TestNG, which can help us design our framework for the test. The annotations in TestNG can be broadly classified into two categories **`@BeforeAnnotation`** and **`@AfterAnnotation`**. The following table lists and describes them:

Annotation	Description
`@BeforeSuite`	Any statement written within this annotation will execute at the beginning of the suite execution.
`@AfterSuite`	Any statement written within this annotation will execute at the end as the suite execution finishes.
`@BeforeTest`	Any statement written in this annotation will execute before the test within the suite execution begins.
`@AfterTest`	The statements written in this annotation will execute after the test is completed. And it will execute after the test for every test in the suite file.
`@BeforeGroups`	The statement in this annotation will execute for the test mentioned in the group.
`@AfterGroups`	The statement in this annotation will execute after the test completes mentioned in the group.
`@BeforeClass`	The statement in this annotation will begin before the test mentioned in the class code.
`@AfterClass`	The statement in this annotation will execute after the test method of the class is over.
`@BeforeMethod`	The code mentioned in this annotation will execute before test method.
`@AfterMethod`	The code mentioned in this annotation will execute after the test method.

***Table 8.1**: TestNG annotations*

In the preceding table, we saw the various TestNG annotations, and a small description for each of them. Before we learn how to use them, we need to understand the TestNG suite XML file. A sample TestNG suite file look as follows:

```
<suite name="RegressionSuite" verbose="1" >
  <test name="Chrome" >
    <classes>
       <class name="loginLogout" />
    </classes>
  </test>
   <test name="Regression">
    <classes>
      <class name="loginLogout"/>
      <class name="changeProfile"/>
    </classes>
  </test>
</suite>
```

As shown in the preceding XML file snippet, we have a suite tag as the root node. It can contain one or more than one test tag. A test tag contains the classes which are the Java **`testng`** class files created and will execute as we put the suite for execution. The suite **`.xml`** file follows the hierarchy shown in *figure 8.10*.

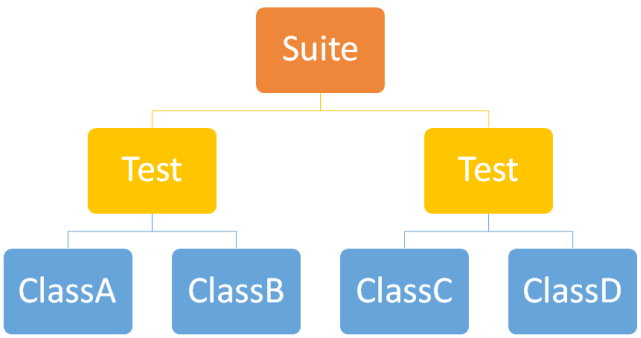


***Figure 8.10**: TestNG hierarchy*

This image shows that the suite node is the root node, and it will contain at least one test node. A test node will contain at least one class node. The class contains information of the **`testng`** class file where the actual code for test is written for execution. In another structure of a suite file, we can also have group information mentioned. The group information states which class files need to be included or

excluded for a test during execution. The sample suite file with group information will look as follows:

```
<suite>
 <test name="Regression1">
   <groups>
     <run>
       <exclude name="regressionTest"  />
       <include name="UITest"  />
     </run>
   </groups>
     <classes>
     <class name="loginLogout"/>
    <class name="buyProduct"/>
    <class name="findOrders"/>
    </classes>
 </test>
</suite>
```

This sample suite file states the information about the groups which should be included during execution, and the ones which should be excluded. The group information is passed as a test method parameter information in the actual TestNG class file. A sample test method that contains group information will look as follows:

```
  @Test(groups = "UITest")
  public void loginLogout() {
        //code to execute
  }
```

The test method in the class file has available various parameters which can be passed to it. The following table lists and describes a few of them:

Parameter	Description
dataProvider	This parameter helps us the data provider name to the test, which helps in parameterizing the test.
dependsOnMethods	Here, we can list all the methods on which execution the method will depend on.
groups	A test method can be part of one or more groups. If the group is included in the TestNG file, the method will execute. If the group is excluded, the test method will not execute.

`priority`	By default, the test methods execute in the alphabetical order of their names. If this is set, the lower priority test methods will execute first.
`alwaysRun`	The test method will always run, even if the `dependsOnMethod` fails.

***Table 8.2**: Parameters of test method*

Note: More information on the parameters are available on this link to explore: https://testng.org/doc/documentation-main.html# annotations.

Assertions in TestNG

Assertions are a way to validate the steps which we have performed to understand if they work or do not work. If they work, we mark the test case as passed; else, we mark the test case as a failure. There is an inbuilt assertion module in TestNG, which has various methods that we can use in our script design to assert the actions we have done.

Note: The assertion methods which are available for TestNG are listed here: https://www.javadoc.io/doc/org.testng/testng/latest/org/testng/asserts/Assertion.html.

We will here see some of the methods from the preceding link, which we will be using in our scripts. As the scenario demands, we can choose other methods mentioned in the link.

Method	Description
`assertEquals(arg 1, arg 2)`	The `assertEquals` method takes various forms, where it compares different types of entities with the other. Here, the arguments passed could be integers, class objects, arrays, and so on. And this method returns true if the arguments are equal to each other.
`assertNotEquals(arg1, arg2)`	The `assertNotEquals` method takes various forms, where it compares different types of entities with the other. Here, the arguments passed could be integers, class objects, arrays, and so on. And the method returns true if the arguments are not equal to each other.
`assertNull(arg)`	This method returns true if the argument passed for the method is Null.

`assertTrue(boolean condition, string message)`	If the Boolean condition evaluated returns true, then the message passed will be printed.
`fail(string message)`	This method will mark the test as a fail and print the message.

***Table 8.3**: Assertion methods in TestNG*

We will use these methods in the scripts we design using TestNG to mark the test method as pass or fail depending on the condition being evaluated as the application is under execution using Selenium.

Result and reporting in TestNG

When a TestNG suite file [.xml] is executed, it generates three types of result files. These files carry the same information but are represented in different manners. The result is stored in the test-output folder, which generates after execution. The following image shows the folder structure which gets displayed in the project tree after you have clicked refresh after the execution of the suite file is completed.

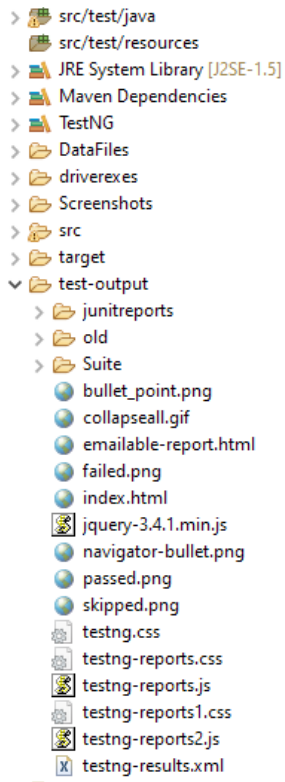
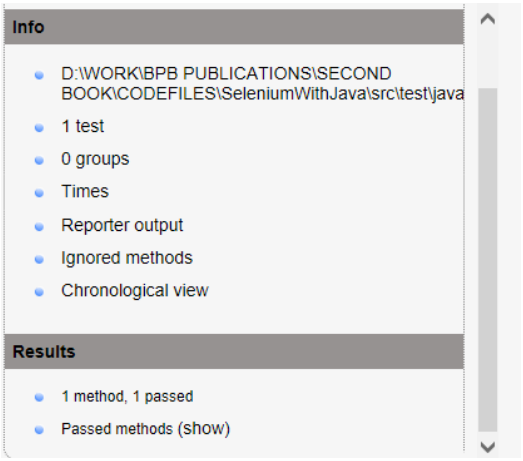
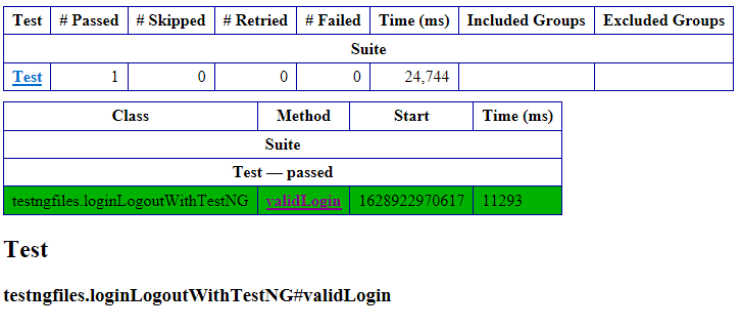


***Figure 8.11**: Result folder of Test NG—test-output*

The files that get generated when the TestNG suite executes, and their description is available as follows:

Result file name	Description
`index.html`	The `index.html` is a file available in the test-output folder. It contains minimal information about the suite file and the methods which have passed or failed in the file. The following image shows how this file looks like. 
`emailable-report`	The e-mailable-report file contains information in a tabular manner and displays the methods in rows. The methods which pass are marked as green, and those which have failed are marked as red. It also contains information for the execution time. The following image shows how the file looks like. 

`Test [html file]`	This result file is available in the `Suite` folder. The suite folder is created using the name for a suite, which we pass in the testng.xml file. By default, the name of the folder will be a suite, and the result HTML file name will be `Test.html`. The result file will be generated for every test inside the suite, and its name will be the name of the test. By default, the file which we will see will have the name `Test.html`. The file contains information about suite execution, start date and time, groups information, if any, and the method pass and fail status. **Test** Tests passed/Failed/Skipped: 1/0/0 Started on: Sat Aug 14 12:05:57 IST 2021 Total time: 24 seconds (24744 ms) Included groups: Excluded groups: *(Hover the method name to see the test class name)* **PASSED TESTS** **Test method** / **Exception** / **Time (seconds)** / **Instance** **validLogin** Test class: testngfiles.loginLogoutWithTestNG / 11 / testngfiles.loginLogoutWithTestNG@26275bef

***Table 8.4**: Result files of TestNG*

As the test suite is executed, we can pass on some reporting information in these result files to make them more meaningful and readable for the stakeholders. To log the messages in the result files, we can use the object of TestNG reporter [**https://javadoc.jitpack.io/com/github/cbeust/testng/master/javadoc/org/testng/Reporter.html**] and call the log method, which takes an input argument as a string. In the following section, when we design a TestNG test and suite eventually, we will see its usage.

Design TestNG test

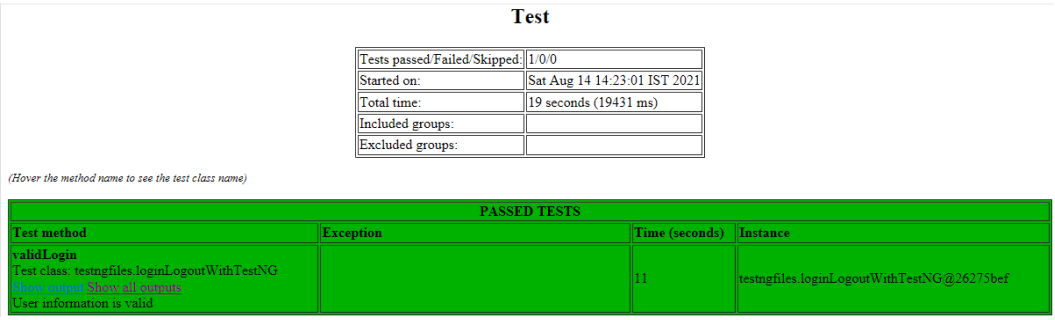
In this section, we will explore creating a TestNG class file. The scenario which we will pick for this is the login logout scenario. We will create two class files with two methods, one for login to the application successfully and another to test incorrect login. We will then call these files from our TestNG suite file.

Let us first see the code for the **`loginLogout`** TestNG class file. We will from here now start using **`webdrivermanager`**; as in the upcoming chapters, we will start talking more about the various components that come together to help us design and write better test automation suites. You can choose to still work with driver executables if you wish to.

```
package testngfiles;

import org.testng.annotations.*;
import io.github.bonigarcia.wdm.*;
import java.util.concurrent.TimeUnit;
import org.testng.Reporter;
import org.openqa.selenium.*;
import org.openqa.selenium.chrome.*;

public class loginLogoutWithTestNG {
      WebDriver driver;

  @BeforeMethod
  public void setupBrowser() {
        //set up chromedriver with webdrivermanager
        WebDriverManager.chromedriver().setup();
        //create instance of chrome
        driver= new ChromeDriver();
        driver.manage().timeouts().implicitlyWait(10, TimeUnit.SECONDS);
  }
@Test
  public void validLogin() {
            //login.
            driver.get("http://www.practice.bpbonline.com/");
            driver.findElement(By.linkText("My Account")).click();
            driver.findElement(By.name("email_address")).
sendKeys("bpb@bpb.com");
            driver.findElement(By.name("password")).
sendKeys("bpb@123");
            driver.findElement(By.id("tdb1")).click();
            if(driver.getPageSource().contains("My Account
Information")) {
                  //log off action
driver.findElement(By.linkText("Log Off")).click();
                  driver.findElement(By.linkText("Continue")).
click();
                  Reporter.log("User information is valid");
                  }else {
                        Reporter.log("User information is
invalid");
                  }
```

```
    }

    @AfterMethod
    public void cleanUp() {
         //close the browser
         driver.close();
    }

}
```

As we can see in the preceding TestNG test design, we have a **`@BeforeMethod`** annotation where we have set up the browser object. We are creating an instance of the **`webdrivermanager`** object and asking it to be set to chrome. In the **`@Test`** method, we have written the code to perform the login logout in our application. In this test, we have passed valid user login details. And finally, in the **`@AfterMethod`**, we wrote the code for the browser to close. We also used the TestNG Reporter method log to provide information in the result files based on whether test passed or failed. Let us see the **`Test.html`** file after the suite execution is completed. The following image gives us a snapshot of it.



Test

Tests passed/Failed/Skipped:	1/0/0
Started on:	Sat Aug 14 14:23:01 IST 2021
Total time:	19 seconds (19431 ms)
Included groups:	
Excluded groups:	

(Hover the method name to see the test class name)

PASSED TESTS			
Test method	**Exception**	**Time (seconds)**	**Instance**
validLogin Test class: testngfiles.loginLogoutWithTestNG Show output Show all outputs User information is valid		11	testngfiles.loginLogoutWithTestNG@26275bef

***Figure 8.12**: Suite execution result file*

We need to understand here is that we used a **`testng.xml`** file, which is the suite file for executing the TestNG test. The suite XML file which we used is as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<suite parallel="false" name="Suite">
  <test name="Test">
    <classes>
      <class name="testngfiles.loginLogoutWithTestNG"/>
    </classes>
  </test> <!-- Test -->
</suite> <!-- Suite -->
```

In this **testng.xml** file, we can see that the name given to our test suite is **Suite**, and the name of the test is **Test**. Our test contains only one class file for execution, and we passed its name so that the code mentioned in the file, in order of the annotation, can get executed. So first, the code in **@BeforeMethod** will execute, then in the **@Test** method, and finally, **@AfterMethod** will execute.

We need to understand here that based on our code structure, our test automation suite design, we can and should design our class files using proper annotations and use them in the test suite XML file. In our next section, we will see how we can pass data to our tests using the parameters tag in the suite XML file, and also see how we can use the **@DataProvider** annotation to parameterize our test method in the class file.

Passing data in TestNG test

Data management and structuring form an integral part of the test automation suite design. The TestNG framework provides us with different ways to manage and handle passing data to the test. We have available with us the parameters tag in the suite XML file and the **@DataProvider** annotation, which we can use in the class file, to parameterize the **@Test method**. Let us see in this section the situation where we can use these two entities to make our test design better.

Let us take a test suite design, where we have more than one test available. We need to pass the information of the browser as a configuration setting, mentioning before suite execution begins which browser the test should use. It could be also possible that we design a test suite file in a manner that a group of tests take one particular browser for execution. To achieve this, we need to use the **Parameter** tag, which is available for us to use during the suite XML file design.

To use it, we will create a test suite, which contains more than one test, and pass the browser information as a configuration setting. To implement this, we will see how our suite file will look like and where and how we will be using the **Parameter** information which will be passed to the test from the suite file.

For example, let our test suite contain two test files, one for valid user login and the other for the invalid user login, and we pass browser execution information from the suite file. To show this, we created a **baseFile**, which has a method to set the driver object based on the browser name passed to it. We then call the method in our TestNG class file and using the parameter tag in the suite file, we pass the browser name, which for this example, we are going to use chrome. Let us see the code files as follows:

baseFile

```
package testngfiles;
import io.github.bonigarcia.wdm.*;
import java.util.concurrent.TimeUnit;
import org.openqa.selenium.*;
import org.openqa.selenium.chrome.*;

public class baseFile {

	static WebDriver driver;

	public static WebDriver setDriver(String browserName) {

		if(browserName.equals("chrome")) {
			WebDriverManager.chromedriver().setup();
			driver= new ChromeDriver();
			driver.manage().timeouts().implicitlyWait(10,
TimeUnit.SECONDS);
		}else {
			driver=null;
		}

	  return driver;
	}
}
```

`loginLogoutWithParams`–Java code is as follows:

```
package testngfiles;

import org.testng.annotations.*;
import org.testng.Reporter;
import org.openqa.selenium.*;

public class loginLogoutWithParams {
	WebDriver driver;

  @BeforeMethod
  @Parameters ({"browser"})
  public void setupBrowser(String brow) {
	driver= baseFile.setDriver(brow);
  }

  @Test
  public void validLogin() {
```

```
            //login.
            driver.get("http://www.practice.bpbonline.com/");
            driver.findElement(By.linkText("My Account")).click();
            driver.findElement(By.name("email_address")).
sendKeys("bpb@bpb.com");

            driver.findElement(By.name("password")).
sendKeys("bpb@123");
            driver.findElement(By.id("tdb1")).click();
            if(driver.getPageSource().contains("My Account
Information")) {
                  //log off action
                  driver.findElement(By.linkText("Log Off")).
click();
                  driver.findElement(By.linkText("Continue")).
click();
                  Reporter.log("User information is valid");
                  }else {
                        Reporter.log("User information is
invalid");
                  }

  }

  @AfterMethod
  public void cleanUp() {
       //close the browser
       driver.close();
  }
}
```

The TestNG suite XML file look as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<suite parallel="false" name="DemoSuite">
 <parameter name="browser"  value="chrome"/>
  <test name="LoginLogout">
    <classes>
      <class name="testngfiles.loginLogoutWithParams"/>
    </classes>
  </test> <!-- Test -->
</suite> <!-- Suite -->
```

So in these code files, we saw how we can pass the browser information in the parameter file. Next, we will see how we can pass the information using the data provider in the test method.

DataProvider

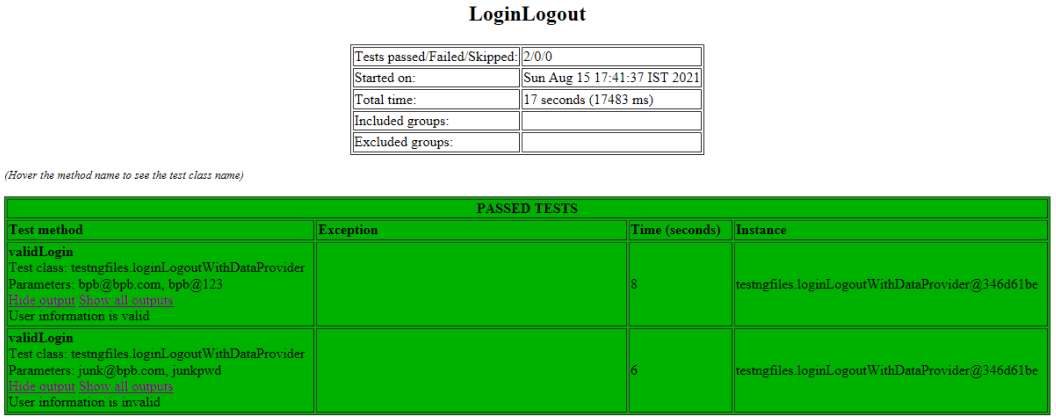
The **@DataProvider** annotation takes data values in the form of a two-dimensional array object. We can pass any data type, whether it is an integer, a double or an object of a class. The name which we use for the **DataProvider** is passed to the **@Test** method. There could be more than one data provider available in a class file. In an example, which we will take to understand the usage of the **DataProvider**, we will pick data for user details from a CSV file. The code for this will be written in the **@DataProvider** annotation. We will then use this to parameterize the **@Test** method. Let us look at the following code.

loginLogoutWithDataProvider–Java code file:

```
package testngfiles;

import org.testng.annotations.*;
import org.testng.Reporter;
import java.io.*;
import org.openqa.selenium.*;

public class loginLogoutWithDataProvider {
      WebDriver driver;

  @BeforeMethod
  @Parameters ({"browser"})
  public void setupBrowser(String brow) {
      driver= baseFile.setDriver(brow);
  }

  @DataProvider (name = "logininformation")
  public Object[][] dpMethod() {

        String data[][]=new String[2][2];//declaring the 2D array
with dimensions
        int i=0;
        try {
            //parsing a CSV file into BufferedReader class constructor
BufferedReader br = new BufferedReader(new FileReader("D:\\WORK\\BPB
PUBLICATIONS\\SECOND BOOK\\CODEFILES\\SeleniumWithJava\\DataFiles\\
logininformation.csv"));
            String line=br.readLine();
            while (line!= null)
```

```
            //returns a Boolean value
            {
              String[] userInfo = line.split(",");
              data[i][0]=userInfo[0];
              data[i][1]=userInfo[1];
              i++;
                 line=br.readLine();
            }
          }
          catch(IOException e) {
            e.printStackTrace();
          }

      return data;
   }

  @Test(dataProvider = "logininformation")
  public void validLogin(String user, String pwd) {
            //login.
            driver.get("http://www.practice.bpbonline.com/");
            driver.findElement(By.linkText("My Account")).click();
            driver.findElement(By.name("email_address")).
sendKeys(user);
            driver.findElement(By.name("password")).sendKeys(pwd);
            driver.findElement(By.id("tdb1")).click();
            if(driver.getPageSource().contains("My Account
Information")) {
                  //log off action
                  driver.findElement(By.linkText("LogOff")).click();
                  driver.findElement(By.linkText("Continue")).click();
                  Reporter.log("User information is valid");
                  }else {
                        Reporter.log("User information is
invalid");
                  }

  }

  @AfterMethod
  public void cleanUp() {
      //close the browser
      driver.close();
  }

}
```

In the preceding file, we pass the browser information through **`Parameter`**, and use the **`@DataProvider`** annotation to read information from the CSV file and pass it to the **`@Test method`**. We will note that the test gets executed twice. Let us look at the result file after the execution is over as follows:



LoginLogout

Tests passed/Failed/Skipped:	2/0/0
Started on:	Sun Aug 15 17:41:37 IST 2021
Total time:	17 seconds (17483 ms)
Included groups:	
Excluded groups:	

(Hover the method name to see the test class name)

PASSED TESTS			
Test method	**Exception**	**Time (seconds)**	**Instance**
validLogin Test class: testngfiles.loginLogoutWithDataProvider Parameters: bpb@bpb.com, bpb@123 Hide output Show all outputs User information is valid		8	testngfiles.loginLogoutWithDataProvider@346d61be
validLogin Test class: testngfiles.loginLogoutWithDataProvider Parameters: junk@bpb.com, junkpwd Hide output Show all outputs User information is invalid		6	testngfiles.loginLogoutWithDataProvider@346d61be

***Figure 8.13**: Result file of suite execution with data provider*

Thus, in the preceding code, we saw how we can use the **`@DataProvider`** to parameterize the **`@Test`** method and read data from a CSV file.

Conclusion

TestNG is a widely used framework and is one of the most commonly used components while designing test automation suites using open-source solutions like Selenium to drive the browser. In this chapter, we have covered what TestNG is and how we can install it in eclipse. We saw the meaning of different annotations, understood the test suite XML file. We saw the different result files generated after execution and how we can log information to it. Finally, we saw how we can pass data to our TestNG test. In the upcoming chapter, we will discuss about the concept of the page object model and how it can be used to design our test.

Questions

1. What are the different annotations available with TestNG?
2. Which method is used to log information to the result file?
3. What are the two ways to pass data to a TestNG test?

Chapter 9
Concept of Page Object Model

Test automation of Web applications is a complex process, and when we try to automate the process of testing a Web application using open-source technologies like Selenium, we need to create infrastructure at the code level to handle many artifacts which we need in our code. The object of the Web application, on which we will perform events to achieve the end result, is one of the main artifacts. Handling the information related to objects and managing it outside the test scripts is crucial for the success of our test automation effort. In this chapter, we will learn about how using a design pattern; we can handle object information and how Selenium class can be of help as well.

Structure

The chapter is divided into the following sections:

- Design pattern—page object model
- Code example for page object model
- Handling objects using page factory class of Selenium
- Code example for page factory

Objectives

The objective of this chapter is to understand the need for implementing a page object model at our code level. Specifically, when we are working with open-source technologies like Selenium, which by default does not come up with any mechanism to handle and manage the object information which we use in the test scripts. Many commercial tools such as UFT, test complete, and so on come with a feature set called an object repository which helps manage object information from the web application that we use in the test script. But as we have learned, Selenium is only a tool to drive the Web browser and does not come up with these thrills and frills. We need to use good coding practices, design patterns to handle these artifacts for our code. In this chapter, we will explore it.

Page object model

Martin Fowler first described this term under the name window driver, but it has been popularized as a page object by the Selenium tool and is generally used under this name. Will sincerely recommend reading about page object in detail from the following two links—Martin Fowler—**https://martinfowler.com/bliki/PageObject.html**.

And another is from the Selenium website documentation—**https://www.selenium.dev/documentation/guidelines/page_object_models/**.

To understand page object, let us first understand the two basic contents of a Web page. We say a Web page consists of two important entities:

- The business logic of the page.
- And the objects available on the page that help us implement the business logic.

As the application develops over time, we will find that the business logic does not undergo as much change as the page object information might. As we design code, we need to write it in such a manner that we separate the object information from the business logic of the page. We will also notice that as we generate test automation code, we will come across scenarios that occur in more than one test. For example, in our Web application, the login logout scenario will occur in login logout action, the change profile action, change password action, buy product action, and many more. We can achieve the preceding using two ways; one is by designing ourselves POM at the code level, which will be explained first. And the second approach is to use page factory, which is a class defined by Selenium to help us implement POM. This will be explained after.

Implementing page object model

Let us take an example to understand the implementation of the page object model for our demo site application login logout scenario. To understand the concept, let us take a page—sign in, which we see after we click on the **`My Account`** link from the main page or our application URL—**http://practice.bpbonline.com/**.

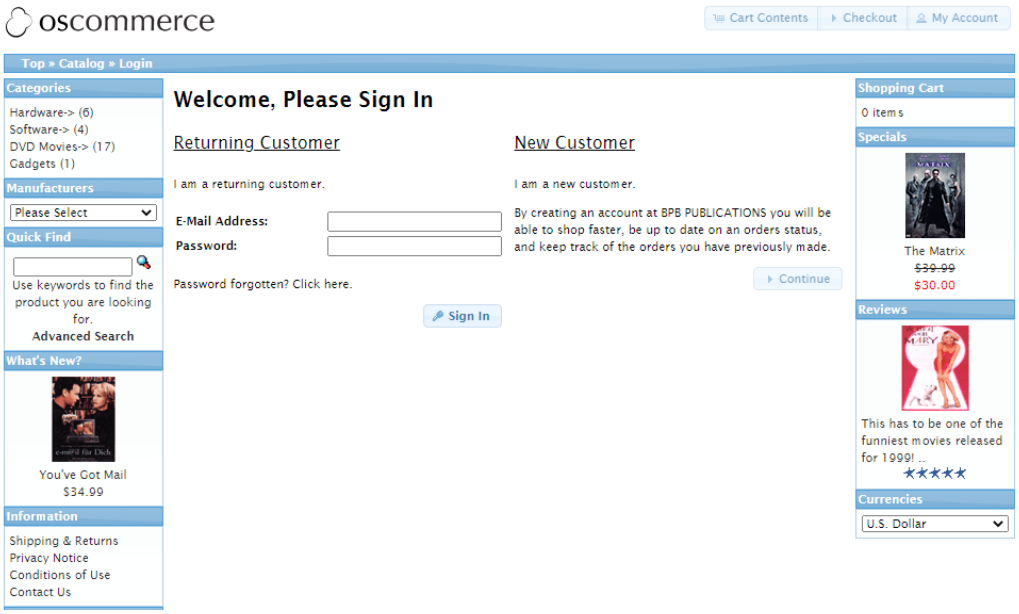


***Figure 9.1**: Sign in page*

If we look at the preceding page, we see a few objects on the page, and there is a business objective associated with the page. The following describes the business objective of the page:

- Allow a user with a valid credential to login into the application by providing a username and password.
- Do not allow a user with invalid details to login to the application.
- Allow a user who has forgotten password to retrieve its credential by clicking on the **`Password forgotten, Click here`** link.
- Allow a new user to register to the application by clicking on the **`Continue`** button, which takes to the register user page.

If we look at the objects which are available on this page, which help us achieve the business objectives are as follows:

- **`Email Address`**: Textbox
- **`Password`**: Textbox
- **`Sign In`**: Button
- **`Password forgotten`**: Link
- **`Continue`**: Button

As the application develops, we should be aware that there is a possibility that object information will get changed, where the HTML properties can change, new objects can be added, old ones removed. For example, it is possible that the **`Password forgotten`** is changed into a button, or the **`Continue`** button is changed into a link, and so on. But the business objective of this page is less likely to undergo change.

The concept of page object model says that we separate the object information from the business logic of the page. And maintain objects of the page separately from our test script code. This helps us in introducing modularity at the code level because it is possible as we develop and design test scripts, the same objects as they are part of some common business flow, are called again and again. And if in some build those objects undergo changes, we will have to open all test scripts to make the change if the page object model concept is not implemented. Implementing a page object model helps us maintain and manage object information separately as well.

To move forward, let us take the login logout scenario and create files to implement the page object model for it. For this, we create a new class file and give it an appropriate name—**`login_pom`**. In this file, the first thing we will do is create By class objects for each of the HTML objects of the page. And then, we will create functions for every action we want to perform on those objects. For example, let us take the **`My Account`** link:

Code for **`By`** object for **`My Account`** link:

```
By myaccount_link= By.linkText("My Account");
```

And the next is we create the function for the click action, which we will want to perform on the **`My Account`** link:

```
public Login_Pom clickMyAccount() {
             driver.findElement(myaccount).click();
          // Return the current page object as this action doesn't
navigate to a page represented by another PageObject
           return this;
       }
```

We should notice here that the return method of the function is the class itself; the reason for this is, we need to have the page reference made available to us after the

action is taken so that the next action can take place on it.

Now, let us take all the objects of the page and see the code as follows:

```
package pageobjectmodel;

import org.openqa.selenium.By;
import org.openqa.selenium.Keys;
import org.openqa.selenium.WebDriver;


public class Login_Pom {
       private WebDriver driver;
       //Constructor of the class
       public Login_Pom(WebDriver driver) {
        this.driver = driver;
     }

 // The login page contains several HTML elements that will be
represented as WebElements.
    // The locators for these elements should only be defined once.
       By myaccount = By.linkText("My Account");
        By usernameLocator = By.name("email_address");
        By passwordLocator = By.name("password");
        By loginButtonLocator = By.id("tdb1");
 // This will click on the MyAccount link
        public Login_Pom clickMyAccount() {
              driver.findElement(myaccount).click();
           // Return the current page object as this action doesn't
navigate to a page represented by another PageObject
            return this;
        }


    // The login page allows the user to type their username into the
username field
    public Login_Pom typeUsername(String username) {
        driver.findElement(usernameLocator).sendKeys(username);
        return this;
    }
 // The login page allows the user to type their password into the
password field
 public Login_Pom typePassword(String password) {
        driver.findElement(passwordLocator).sendKeys(password);
```

```
    return this;
     }

     // The login page allows the user to submit the login form
     public Login_Pom submitLogin() {
         driver.findElement(loginButtonLocator).submit();
         return this;
     }

     public boolean validateLogin(String srchTxt) {
         if (driver.getPageSource().contains(srchTxt)){
              return true;
         }else{
              return false;
         }
     }

  // Conceptually, the login page offers the user the service of
being able to "log into"
     // the application using a user name and password.

     public Login_Pom loginAs(String username, String password) {
         // The PageObject methods that enter username, password &
submit login have already defined and should not be repeated here.
         typeUsername(username);
         typePassword(password);
         return submitLogin();
     }
}
```

In the same manner, we can implement the page object for the log-off page scenario. For log-off page to appear, we need to be first logged in to the application. And then, the process is we click on the **`Log Off`** link and then the continue link to log off from the application. The following image shows the flow:

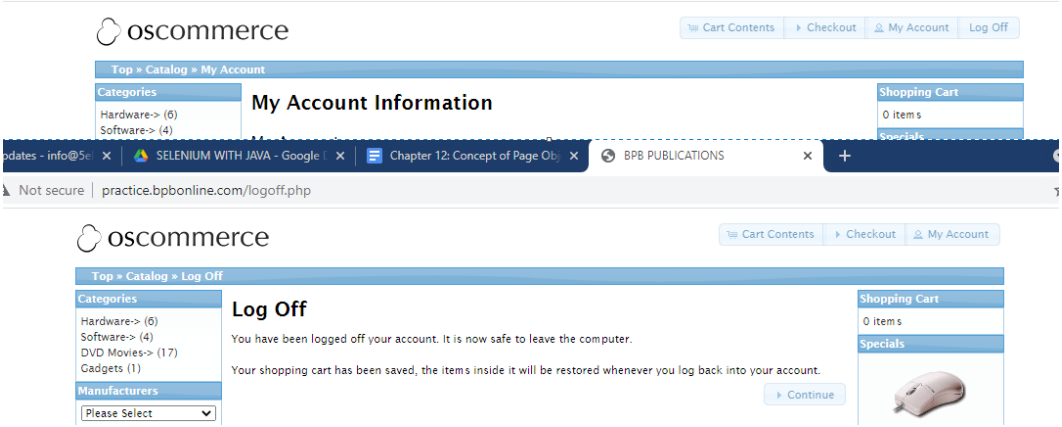


***Figure 9.2**: Log Off page scenario*

So, for the log-off page object model, we will be using the log-off link and the **`Continue`** link. Let us see the following code:

```
package pageobjectmodel;

import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;


public class Logout_Pom {
       private WebDriver driver;

    public Logout_Pom(WebDriver driver) {
        this.driver = driver;
       }

       By logOff = By.linkText("Log Off");
        By continueButton = By.linkText("Continue");

       public Logout_Pom clickLogOff() {
            // This is the only place that "knows" how to enter a
username
            driver.findElement(logOff).click();
            return this;
        }

        public Logout_Pom clickContinue() {
            // This is the only place that "knows" how to enter a
username
```

```
            driver.findElement(continueButton).click();
            return this;
        }

    public Logout_Pom logOff() {
      clickLogOff();
      return clickContinue();
    }
}
```

We will now create a third class, where the business logic and validations for the login logout process will be written. And in this class, instead of writing down the steps where we first identify the Web element and then perform an action on it, we will call the objects from the **login_pom** and the **logout_pom** and access the methods defined there. Let us see the following code:

```
package pageobjectmodel;

import java.util.concurrent.TimeUnit;

import org.junit.*;
import org.openqa.selenium.*;
import org.openqa.selenium.chrome.*;

import io.github.bonigarcia.wdm.WebDriverManager;


public class loginLogoutUsingPOM {
        private WebDriver driver;

       @Before
        public void setUp() throws Exception {
             //set up chromedriver with webdrivermanager
            WebDriverManager.chromedriver().setup();
            driver = new ChromeDriver();
          driver.manage().timeouts().implicitlyWait(30, TimeUnit.
SECONDS);
        }

       @Test
       public void test() throws Exception{
             driver.get("http://practice.bpbonline.com/");
             Login_Pom login = new Login_Pom(driver);
             Logout_Pom logout = new Logout_Pom(driver);
```

```
                login.clickMyAccount();
                login.loginAs("bpb@bpb.com", "bpb@123");
                login.validateLogin("My Account Information");
                logout.logOff();

        }

        @After
        public void cleanup(){
                driver.quit();
        }
}
```

In this code, we can see the following lines:

```
1   Login_Pom login = new Login_Pom(driver);
2   Logout_Pom logout = new Logout_Pom(driver);
3   login.clickMyAccount();
4   login.loginAs("abc@demo.com", "demo@123");
5   login.validateLogin("My Account Information");
6   logout.logOff();
```

Lines 1 and 2 are where we define the object for the login page object model class and the logout page object model class. In Line 3, we call the **clickMyAccount** link method. This method, as we are aware, is declared in the **login_pom** class:

```
public Login_Pom clickMyAccount() {
            driver.findElement(myaccount).click();
          // Return the current page object as this action doesn't
navigate to a page represented by another PageObject
            return this;
       }
```

We need to also note here an important step. The WebDriver is declared in the calling class, and when we instantiate the object of **login_pom,** the driver object is passed. It is in the **login_pom** constructor method, where we see that the WebDriver object in **login_pom** is assigned the driver object passed. The method is shown as follows:

```
//Constructor of the class
      public Login_Pom(WebDriver driver) {
        this.driver = driver;
    }
```

The same is also true for the **logout_pom** class. Lines 4 and 5 are where the actual login will take place, and the action is validated. Step 6 calls the log-off method from the **logout_pom**, where we call the clicking of log-off link method and the continue link method. The same is shown here:

```
  public Logout_Pom logOff() {
      clickLogOff(); //method in logout_pom
      return clickContinue(); //method in logout_pom
    }
```

clickLogOff

```
public Logout_Pom clickLogOff() {
            // This is the only place that "knows" how to enter a
username
            driver.findElement(logOff).click();
            return this;
        }
```

clickContinue

```
  public Logout_Pom clickContinue() {
            // This is the only place that "knows" how to enter a
username
            driver.findElement(continueButton).click();
            return this;
        }
```

Now, in any test script scenario, where we need to call the login and logout action, for example, in **Change Profile** or **Buy Product**, we can simply call the methods from the **login_pom** and **logout_pom**. Also, if any of the objects used in these classes undergo any change, the same needs to be done only in these classes and nowhere else in the code.

The benefits of implementing page object models for our test automation code are many. We achieve modularity, code abstraction, reduce chances of error, less code lines, separate object information management, separation of business logic and application objects. Thus, it is important that any good test automation project using Selenium implements the page object model concept.

Implementing page factory

Selenium provides us with a class to implement page object model in our code. The class to do is called as page factory. We also use an annotation enum which is described using **FindBy**. To implement **PageFactory** class, we do not use the **By** class to define the objects. Instead, we use the annotation available in **FindBy**, which

allows us to define the way we wish to find something, such as **`NAME`**, **`ID`**, **`LinkText`**, and so on, as well as what value we are searching for in that property. An example of **`FindBy`** for **`My Account`** link will look as follows:

```
@FindBy(linkText = "My Account")
WebElement myAccLnk;
```

So in the **`login`** class for the **`PageFactory`**, the code will look as follows:

```
package pageobjectmodel;

import org.openqa.selenium.*;
import org.openqa.selenium.support.FindBy;
import org.openqa.selenium.support.PageFactory;

public class loginWithPageFactory {
        WebDriver driver;

        @FindBy(linkText = "My Account")
        WebElement myAccLnk;

        @FindBy(name = "email_address")
        WebElement uname;

        @FindBy(name = "password")
        WebElement pwd;

        @FindBy(id = "tdb1")
        WebElement signBtn;

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

        public void clickAccount() {
               myAccLnk.click();
       }

        public void typeUser(String user) {
               uname.clear();
               uname.sendKeys(user);
        }
```

```
        public void typePwd(String password) {
               pwd.clear();
               pwd.sendKeys(password);
        }

        public void clickSign() {
               signBtn.click();
        }

        public void login(String u, String p) {
               typeUser(u);
               typePwd(p);
               clickSign();
        }
}
```

The **logout** class implemented using page factory will look as follows:

```
package pageobjectmodel;

import org.openqa.selenium.*;
import org.openqa.selenium.support.FindBy;
import org.openqa.selenium.support.PageFactory;

public class logoutWithPageFactory {
        WebDriver driver;

        @FindBy(linkText = "Log Off")
        WebElement logoffLnk;

        @FindBy(linkText = "Continue")
        WebElement continueLnk;

        public logoutWithPageFactory(WebDriver driver) {
               this.driver = driver;
               PageFactory.initElements(driver, this);
               }
        public void logOffClick() {
               logoffLnk.click();
       }
```

```
        public void ctnClick() {
               continueLnk.click();
       }


        public void logOff() {
              logOffClick();
              ctnClick();
        }
}
```

In both login using page factory code and logout using page factory code, we will see the following code line in the constructor.

PageFactory.initElements(driver, this);

The purpose of the preceding line is to initialize the page object elements using the driver object, which we have identified using the **`FindBy`** annotation in the class. Finally, the calling code will look as follows:

```
package pageobjectmodel;

import org.testng.annotations.Test;
import io.github.bonigarcia.wdm.WebDriverManager;
import org.testng.annotations.BeforeMethod;
import org.testng.annotations.AfterMethod;
import java.util.concurrent.TimeUnit;
import org.openqa.selenium.*;
import org.openqa.selenium.chrome.ChromeDriver;

public class loginLogoutWithPageFactory {

         WebDriver driver;
         loginWithPageFactory loginPF;
         logoutWithPageFactory logoffPF;

  @BeforeMethod
  public void beforeMethod() {
         //set up chromedriver with webdrivermanager
              WebDriverManager.chromedriver().setup();
              //create instance of chrome
              driver= new ChromeDriver();
```

```
            loginPF=new loginWithPageFactory(driver);
            logoffPF= new logoutWithPageFactory(driver);
          //an implicit wait given for each command to search and
object and perform operation on it
          driver.manage().timeouts().implicitlyWait(30, TimeUnit.
SECONDS);
          //maximize window
          driver.manage().window().maximize();
  }

 @Test
  public void testLogin() throws Exception {
            driver.get("http://practice.bpbonline.com/");
            //login using page factory
            loginPF.clickAccount();
            loginPF.login("bpb@bpb.com", "bpb@123");
            //logoff using page factory
            logoffPF.logOff();
   }

 @AfterMethod
  public void afterMethod() {
        driver.quit();
  }

}
```

So, the preceding code shows how we can implement page object model using the page factory concept, where we use the **`FindBy`** annotation to identify the object and then use the **`Page.init`** method to initialize the objects of the page with the driver object reference. Finally, like we created our own implementation of page object model, in the same manner, we can use the **`PageFactory`** class made available by Selenium to do the same.

Conclusion

Thus, in this chapter, we saw the importance of page object model. We understood the importance of modularity and the advantages we get when we separate object information from the business logic of the page. We saw two implementations of the POM, one by using design patterns by creating separate classes to manage object information and then a calling class that is a test script. The second implementation we saw was using the **`PageFactory`**, which is a class available to us with Selenium. In the upcoming chapter, we will learn about how we can data drive our test scenarios by reading data from a CSV file and an Excel file.

Questions

1. Explain the importance of page object model?
2. What is modularity, and why is it important for code?
3. What does @FindBy annotation do?

Chapter 10
Data Driving Test

Any Web application testing requires test data preparation. Test data forms an integral part of the test case process. As we automate our test scripts, we will find we are using data in the scripts. As the application changes or the test process may undergo changes, it is very much likely that the data used in test scripts also undergo changes. So, it is imperative that we manage data used in test scripts external to our test logic. Also, we may have some environment configuration data such as application URL, some file paths for reports, logs, and so on, which should also be managed externally. In this chapter, we will see how we can manage data for our test and use it judiciously in our scripts.

Structure

The chapter is divided into the following sections:

- Importance of data management
- Managing data in CSV file, Excel file, and using them in test

Objectives

The objective of this chapter is to understand the importance of data as an artifact when we are designing our tests. In this chapter, we will also see how we can read

data from a CSV file or an Excel file. We will understand the need to manage data separately from the test scripts, which contain the business logic. We will be covering reading from two types of data files—the CSV and the Excel. For Excel, there are many packages available we will be using the Apache POI. We need to understand here that managing data has got nothing to do with Selenium. It depends on the programming language we are using to design our test. And it depends on us how to manage the data.

Managing data using CSV

One of the most commonly used data management ways is to put data in a CSV file. CSV stands for comma-separated values. A CSV file will have data separated using commas, or it can also be any other separator like a tab. An example of a CSV file is as follows:

```
Identifier;First name;Last name
901242;Rachel;Booker
207074;Laura;Grey
408129;Craig;Johnson
934600;Mary;Jenkins
507916;Jamie;Smith
```

***Figure 10.1**: CSV file Example*

To read a CSV file, we will be using OpenCSV package available in Java. The OpenCSV package details are available here—**http://opencsv.sourceforge.net/**. We will be using the class CSVReader to read a given CSV file. To manage data, we will be creating separate files where the function for reading data from the CSV file will be written. Eventually, we will add the code to read data from the Excel file also here. Let us first write the code to read from the CSV file, and then we will implement it in a scenario.

To read from a CSV file, where data is comma-separated, we will create an object of the CSVReader class. The detail of CSVReader class information is available here—**http://opencsv.sourceforge.net/apidocs/com/opencsv/CSVReader.html**. Before we write code for reading the CSV file, we will need to add the following dependency in the pom.xml file.

```
<!-- https://mvnrepository.com/artifact/com.opencsv/opencsv -
<dependency>
    <groupId>com.opencsv</groupId>
    <artifactId>opencsv</artifactId>
    <version>5.5.2</version>
</dependency>
```

***Figure 10.2**: Dependency information for OpenCSV*

We will first create a package called as **`dataHandling`**, in the project. And in here is a Java file with the name **`DataReaders`**. Here, we will create a method to read the CSV file. The code is as follows:

```
public static List<String[]> getCSVData(String filename, int skipLines)
throws IOException{

            List<String[]> allData=null;
            try {
               FileReader filereader = new FileReader(filename);
               // create csvReader object and skip first Line
              CSVReader csvReader = new CSVReaderBuilder(filereader)
                                          .withSkipLines(skipLines)
                                          .build();
              allData = csvReader.readAll();
            }
          catch (Exception e) {
              e.printStackTrace();
          }
             return allData;
       }
```

The method takes the file name as an argument, and the second argument is to skip the count of lines. So, we should pass 0 if the CSV files do not contain any row, which has headers. And if the CSV file contains headers, we can pass 1 as the argument. So, the method **`readAll()`**will skip the first line when reading the content of the CSV file into a list of arrays of strings. The list of arrays of strings is the return value of the method. The size of the list will be equivalent to the number of lines in the CSV file, which is read. And the size of the array of strings will be the same as the column in the CSV file. Let us see its usage in a test scenario.

The scenario which we will pick is the login logout from the Web application available here—**http://practice.bpbonline.com/**.

The steps for the login logout are as follows:

1. Click **`My Account`** link:

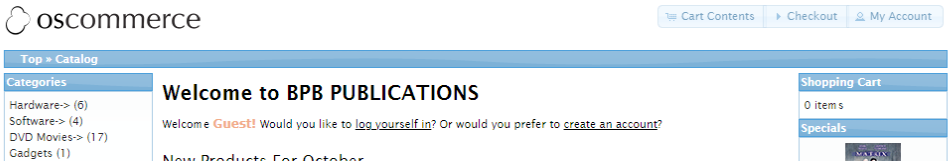


***Figure 10.3**: My Account Link*

2. Fill in username and password, and click on the **`Sign In`** button:

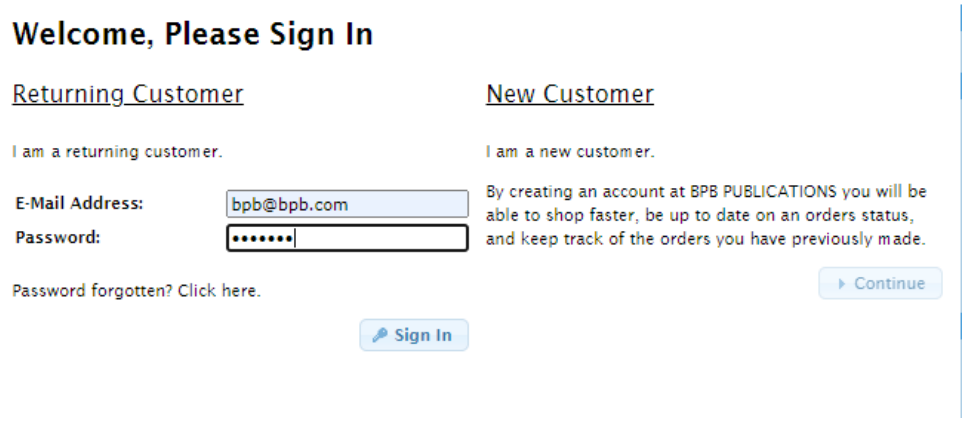


***Figure 10.4**: Fill in login details*

3. If the user details are invalid, it will throw an error message and not allow to login:

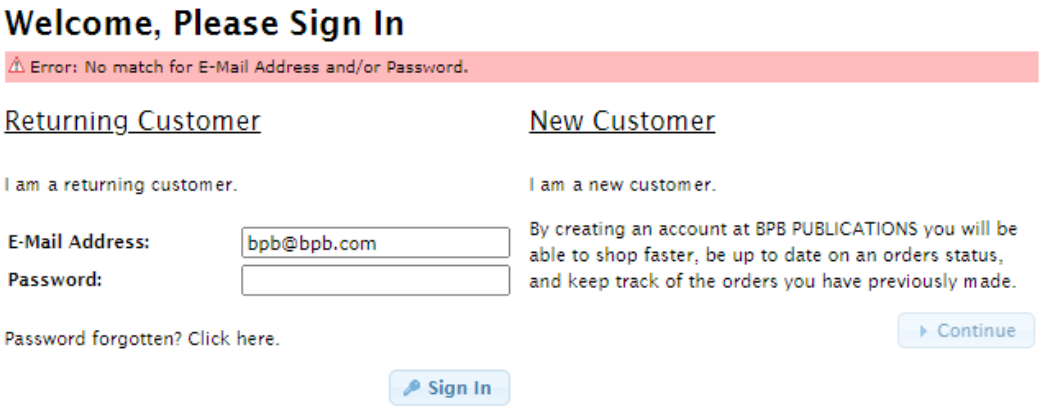


***Figure 10.5**: Error message on invalid user details*

4. If the user is valid, it will allow the user to login. We will see a **`Log Off`** link; on clicking, we can log off from the application.

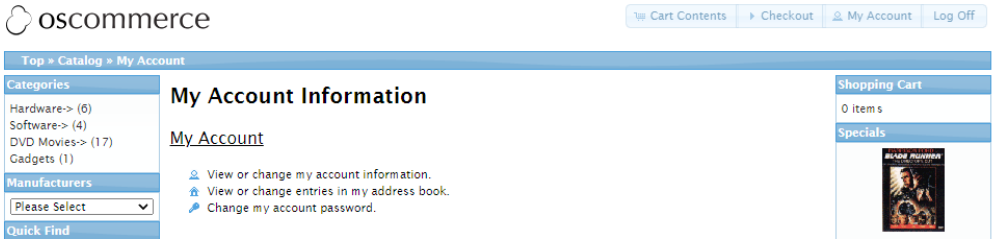


***Figure 10.6**: Log off link view*

5. To complete the log off process, we will need to click on the **`Continue`** link that will come after we click on the **`Log Off`** link.

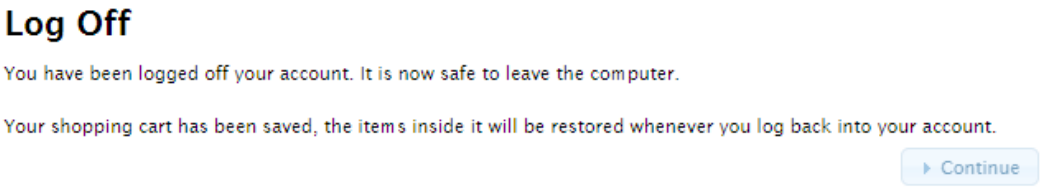


***Figure 10.7**: Continue link*

6. We will be data driving the preceding process. The CSV file data which we will be using is as follows:

```
username,password
bpb@bpb.com,bpb@123
junk,junk
```

***Figure 10.8**: CSV file*

7. We are saving this file in the location **`src\test\resources`**, in a folder called as **`Data`**. Or you can save where you feel appropriate.

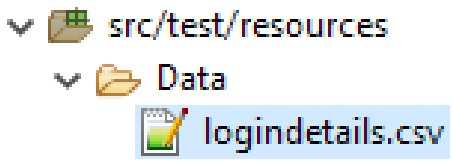


***Figure 10.9**: CSV file location*

Let us have a look at the DataProvider method of the TestNG test before we write the complete code as follows:

```
@DataProvider(name = "logincsv")
        public Object[][] loginCSVData() throws IOException  {

               List<String[]> allData=DataReaders.getCSVData("src\\
test\\resources\\data\\logininfo.csv",1);
               String[][] data=new String[allData.size()][allData.
get(0).length];
               // read data in 2D array
               int i=0;
              for (String[] row : allData) {
                   int j=0;
                  for (String cell : row) {
                      data[i][j]=cell;
                      j=j+1;
                  }
                  i=i+1;
              }
               return data;
       }
```

The return value of DataProvider is a two-dimensional array of type object. And the **`getCSVData`** returns a list of the string array. So, we will need to iterate through the list of the array returned and populate a 2D array object, which will be the DataProvider object as passed to the test method of the TestNG test file. The complete code is as follows:

```
package datadrivingtest;

import org.testng.annotations.*;
import java.io.*;
import org.openqa.selenium.*;
import org.openqa.selenium.chrome.*;
import handlingData.DataReaders;
import io.github.bonigarcia.wdm.WebDriverManager;
import java.util.*;
import java.util.concurrent.TimeUnit;

public class loginLogoutUsingCSV {
      WebDriver driver;
      @BeforeMethod
      public void setUp() {
```

```
            WebDriverManager.chromedriver().setup();
            driver = new ChromeDriver();
            driver.manage().window().maximize();
            driver.manage().timeouts().implicitlyWait(10, TimeUnit.
SECONDS);
      }

      @DataProvider(name = "logincsv")
        public Object[][] loginCSVData() throws IOException  {
            List<String[]> allData=DataReaders.getCSVData("src\\
test\\resources\\Data\\logindetails.csv",1);
            String[][] data=new String[allData.size()][allData.
get(0).length];
            // read data in 2D array
            int i=0;
            for (String[] row : allData) {
                  int j=0;
             for (String cell : row) {
                  data[i][j]=cell;
                 j=j+1;
                 }
                 i=i+1;
             }
              return data;
      }

      @Test(dataProvider="logincsv")
      public void loginUsingWrapperMethods(String uname,String
passwd) throws Exception {
            driver.get("http://practice.bpbonline.com/");
            driver.findElement(By.linkText("My Account")).click();
            driver.findElement(By.name("email_address")).
sendKeys(uname);
            driver.findElement(By.name("password")).
sendKeys(passwd);
            driver.findElement(By.id("tdb1")).click();
            driver.findElement(By.linkText("Log Off")).click();
            driver.findElement(By.linkText("Continue")).click();
      }

    @AfterMethod
    void after() {
       driver.close();
    }
 }
```

The test report for the execution is as follows:

LoginLogout

Tests passed/Failed/Skipped:	2/0/0
Started on:	Sun Aug 15 17:41:37 IST 2021
Total time:	17 seconds (17483 ms)
Included groups:	
Excluded groups:	

(Hover the method name to see the test class name)

PASSED TESTS			
Test method	**Exception**	**Time (seconds)**	**Instance**
validLogin Test class: testngfiles.loginLogoutWithDataProvider Parameters: bpb@bpb.com, bpb@123 Show output Show all outputs		8	testngfiles.loginLogoutWithDataProvider@346d61be
validLogin Test class: testngfiles.loginLogoutWithDataProvider Parameters: junk@bpb.com, junkpwd Show output Show all outputs		6	testngfiles.loginLogoutWithDataProvider@346d61be

***Figure 10.10**: Test report*

As we can see in the preceding test report, the test ran twice, and data was picked from the external CSV file. We can now change the data now in the file without impacting our test case. Thus, with this, we achieve a data-driven test case design. In our next section, we will use an Excel file to do the same.

Managing data using Excel

An Excel file is widely used to manage and save data information. It is also very popular among the testers. An Excel file is associated with a workbook. An Excel workbook is associated with worksheets. An Excel worksheet contains rows and columns. An intersection of row and columns make a cell. And it is here in the cell that the data is read from or written to. So, if we look at the following hierarchy for the Excel file, it will become easy for us to understand the Excel object that is used by the programming languages to manage them. Let us look at the following diagram:

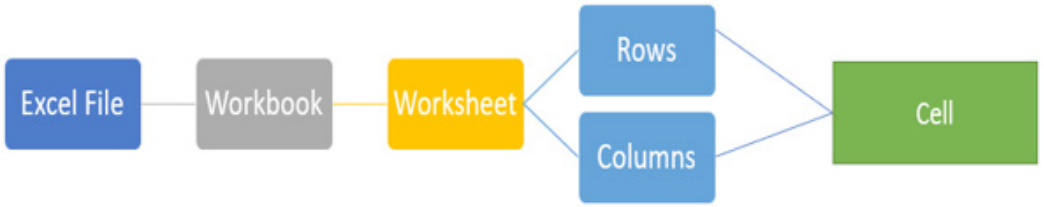


***Figure 10.11**: Excel file hierarchy diagram*

To read data from an Excel file, we will need to use the Apache POI Java package. The Apache POI package in Java allows us to manage and work with Microsoft

documents. The Apache POI package maven dependency information is as follows, which we should add in the **`pom.xml`** before we want to use it in the project.

```
<!-- https://mvnrepository.com/artifact/org.apache.poi/poi -->
<dependency>
    <groupId>org.apache.poi</groupId>
    <artifactId>poi</artifactId>
    <version>5.0.0</version>
</dependency>
```

***Figure 10.12**: Maven dependency for Apache POI*

Here, we need to note that working with the Excel file is depended on the programming language, which we have used to implement Selenium. Selenium, by default, does not come with any support to manage data in an Excel file. The Apache POI has two classes to manage different forms of Excel files. For Excel files with extension **`.xls`**, we have the HSSF class files. For working with the **`.xlsx`** Excel files, we have the XSSF package we can use. To know more about these packages, we can visit—**https://poi.apache.org/components/spreadsheet/**.

Reading data from Excel file

To read data from the Excel file, we have created a function in the **`DataReader.java`** file. Let us understand the working of the function. The function definition is as follows:

```
String[][] getExcelDataUsingPoi(String filename, String sheetname)
```

The function takes as input a filename and the **`sheetname`**. The **`sheetname`** has to match the name in the Excel workbook, which data we want to read. The **`filename`** is the file path where the file is saved. The return value of the function is a two-dimensional array of **`String`** data types. So effectively, we read the rows and columns information from the Excel file and return it as the two-dimensional arrays. In the function, we first define a workbook object which we initialize it to null. And the next is depending on the file extension; for **`.xls`**, we invoke the HSSF object, and for **`.xlsx`** we invoke the XSSF object. Let us see the code line as follows:

```
org.apache.poi.ss.usermodel.Workbook wb = null;
File file = new File(fileName);
FileInputStream fs = new FileInputStream(file);
if(fileName.substring(fileName.indexOf(".")).equals(".xlsx"))
{
wb = new org.apache.poi.xssf.usermodel.XSSFWorkbook(fs);
}
else if(fileName.substring(fileName.indexOf(".")).equals(".xls"))
{
wb = new org.apache.poi.hssf.usermodel.HSSFWorkbook(fs);
}
```

Our next action is to create the worksheet object using the workbook and then find the available rows and columns. The code for the same is as follows:

```
org.apache.poi.ss.usermodel.Sheet sh = wb.getSheet(sheetName);
int totalNoOfRows = sh.getPhysicalNumberOfRows();
int totalNoOfCols = sh.getRow(0).getPhysicalNumberOfCells();
System.out.println("totalNoOfRows="+totalNoOfRows+"," + "
totalNoOfCols="+totalNoOfCols);
```

Finally, we need to read the data from the Excel cell into two-dimensional arrays of data type String. We should note here that it is possible that the Excel cell contains data in different formats, such as number, date format, currency, and so more. We in here are converting any format into string type and storing it in the array. Later in the Java program, if we wish to handle the data in some other format, we can do. The code to read from the cell is as follows:

```
for (int i= 1 ; I <= totalNoOfRows-1; i++) {
      for (int j=0; j <= totalNoOfCols-1; j++) {
            String rawCellVal = null;
            try {
                  DataFormatter formatter = new DataFormatter();
                  rawCellVal = formatter.formatCellValue(sh.
getRow(i).getCell(j));
            }catch(Exception e) {
                                    System.out.println“"error
reading value from the row and cell-- null may b”");
            }
            if (rawCellVal == null || rawCellVal.toString().
contains“"-”")){
                  continueReading= false;
                  break;
            }
            String cellStringVal = rawCellVal.toString();
            arrayExcelData[i-1][j] = cellStringVal;
```

In the complete function of reading data from the Excel file, we have handled situations where it is possible that the object returned is null. This can happen when the worksheet does not exist, or rows and columns are not filled in, and so on. We have also handled a situation in the code where if the cell contains the value of **-1,** we will not read data from that row. We can also use another value to mark the end of reading information from the Excel file. The complete code to read information from the Excel file is as follows:

```
public static String[][] getExcelDataUsingPoi(String fileName, String
sheetName) throws IOException {

            String[][] arrayExcelData = null;
            org.apache.poi.ss.usermodel.Workbook wb = null;
            try {
                  File file = new File(fileName);
                  FileInputStream fs = new FileInputStream(file);
                  if(fileName.substring(fileName.indexOf(".")).
equals(".xlsx"))
                  {
                        wb = new org.apache.poi.xssf.usermodel.
XSSFWorkbook(fs);
                  }
                  else if(fileName.substring(fileName.indexOf(".")).
equals(".xls"))
                  {
                        wb = new org.apache.poi.hssf.usermodel.
HSSFWorkbook(fs);
                  }

                  if (wb==null)
                  {
                        //Error Sheet name not found
                        Exception exp = new Exception("WORKBOOK
CREATION ERROR - May be File **NOT** found " + sheetName );
                        throw exp;
                  }

                  org.apache.poi.ss.usermodel.Sheet sh =
wb.getSheet(sheetName);

                  if (sh==null)
                  {
                        //Error Sheet name not found
                        Exception exp = new Exception("Sheet Name
**NOT** found " + sheetName );
                        throw exp;
                  }
```

```
            int totalNoOfRows = sh.getPhysicalNumberOfRows();
                  int totalNoOfCols =
                              sh.getRow(0).
getPhysicalNumberOfCells();


                  System.out.
println("totalNoOfRows="+totalNoOfRows+","
                              + " totalNoOfCols="+totalNoOfCols);

                  arrayExcelData =
                              new String[totalNoOfRows-1]
[totalNoOfCols];

                  System.out.println("Reading excel file now");
                  // End reading the excel file if the column value
is -1
                  boolean continueReading = true;
                  for (int i= 1 ; i <= totalNoOfRows-1; i++) {
                        for (int j=0; j <= totalNoOfCols-1; j++)
{
                              String rawCellVal = null;
                              try {

                                    DataFormatter formatter =
new DataFormatter();
                                    rawCellVal = formatter.
formatCellValue(sh.getRow(i).getCell(j));
                              }
                              catch(Exception e) {
                                    // error reading cell value
or row value
                                    // looks like it may be null
                                    System.out.println("error
reading value from the row and cell - null may be");
                              }
                              if (rawCellVal == null ||
rawCellVal.toString().contains("-1"))
                              {
                                    continueReading= false;
                                    break;
                              }
```

```
                                        String cellStringVal =
rawCellVal.toString();
                                arrayExcelData[i-1][j] =
cellStringVal;
                                System.out.
print(arrayExcelData[i-1][j]+":");

                        } // inner for loop - j
                        if (continueReading == false) {
                                System.out.println("Completed
reading -1 or null found: breaking now.");
                                break;
                        }
                        System.out.println();
                } // outer for loop i
        } catch (Exception e) {
                System.out.println("EXCEPTION error in
getExcelData()");
                System.out.println(e.getMessage());
                if (arrayExcelData==null)
                {
                        IOException exp = new IOException(e.
getMessage());
                        throw exp;
                }
        }
        return arrayExcelData;
    }
```

Using Excel reading function

Let us have a look at the Excel file from which we need to read data from. The file snapshot is here:

username	password
bpb@bpb	bpb@123
junk	junk

***Figure 10.13**: User details*

We will create a **testng** test for login-logout action, the same which is explained for the CSV file reading. Here, instead of reading data from a CSV file, we will read from the Excel file. Let us look at the **DataProvider** method, which we will use in the code:

```
@DataProvider(name = "loginExcel")
        public Object[][] loginExcelData() throws IOException  {
              String[][] data=DataReaders.getExcelDataUsingPoi("src\\
test\\resources\\data\\datasheet.xlsx", "Data");
              return data;}
```

We will pass the DataProvider name **loginExcel** to our **test** method. The **test** method will iterate twice for both rows for valid user credentials and invalid user credentials. The complete code is as follows:

```
package datadrivingtest;

import org.testng.annotations.*;
import java.io.*;
import org.openqa.selenium.*;
import org.openqa.selenium.chrome.*;
import handlingData.DataReaders;
import io.github.bonigarcia.wdm.WebDriverManager;
import java.util.*;
import java.util.concurrent.TimeUnit;

public class loginLogoutUsingExcel {
       WebDriver driver;

       @BeforeMethod
       public void setUp() {
              WebDriverManager.chromedriver().setup();
              driver = new ChromeDriver();
              driver.manage().window().maximize();
              driver.manage().timeouts().implicitlyWait(10, TimeUnit.
SECONDS);
       }

       @DataProvider(name = "loginExcel")
         public Object[][] loginExcelData() throws IOException  {
               String[][] data=DataReaders.
getExcelDataUsingPoi("src\\test\\resources\\data\\datasheet.xlsx",
"Data");
               return data;
        }
```

```
      @Test(dataProvider="loginExcel")
      public void loginUsingWrapperMethods(String uname,String
passwd) throws Exception {
            driver.get("http://practice.bpbonline.com/");
            driver.findElement(By.linkText("My Account")).click();
            driver.findElement(By.name("email_address")).
sendKeys(uname);
            driver.findElement(By.name("password")).
sendKeys(passwd);
            driver.findElement(By.id("tdb1")).click();
            driver.findElement(By.linkText("Log Off")).click();
            driver.findElement(By.linkText("Continue")).click();
      }

    @AfterMethod
    void after() {
       driver.close();
    }
 }
```

Thus, the preceding code performs the login logout action on the application and runs twice, both for valid user details and invalid user details.

Conclusion

Thus, we have completed how to parameterize our test using data from external sources. We have seen two data sources in this chapter CSV, and Excel. And we have seen the code that we will use to read these different data files. We have also understood that reading data or managing data is the job of the programming language and the person creating the program. By default, Selenium does not come with the capability to handle data files.

In the upcoming chapter, we will see a project management tool called as Maven, which is helpful in managing the project life cycle for projects which use open source software solutions.

Questions

1. Does Selenium support data handling?
2. Which package do we use to read data from the Excel file?
3. How many times does a test iterates? What defines it?

Chapter 11
Introducing Maven

In the previous chapter of the book, we have understood how we can use Selenium to automate the Web browser. We understood the various classes available in Selenium and the different methods in them, which allows us to handle the various types of HTML elements on the Web page. Using them, we can automate our actions to be done on a Web page. We have also understood that by default, Selenium does not come with any function set to manage data and provide validation to the actions. To create a proper test, we need to use a unit test framework like **`junit`** or **`testng`**. And to manage data, our approach depends on the type of data file and the programming language we are using. When we are creating a test automation solution, we have to understand we are creating a software artifact, and it needs a similar type of environment, practices and infrastructure that a software development project would require. To manage project-related documents and dependencies which are related to our test project, we will need to use a build management tool. There are various build management tools available, both free and paid. In this book, we will be looking at Maven as the build management solution.

Structure

The chapter is divided into the following sections:

- Need for build management
- What is Maven
- Installing Maven in Eclipse
- Creating a simple Maven project
- What is Maven repository
- What is project object model

Objectives

The process of creating a test automation solution using Selenium is similar to creating a complete software project. Being open-source and as a library to only automate the Web browser, we need other types of solutions available to manage our test-related activities. Each of these activities requires different types of executables to be made available both in the authoring test environment and execution test environment. A test automation solution, once created, can be used in various test environments to test the working of the application on different browsers, operating systems and other variable factors. For this reason, we require an effective build management solution. A solution that takes care of the project dependencies and is flexible enough to work with open-source software. The reason is that after every few years or maybe less, the open-source solution undergoes version change. And these version changes of the executables on which the working of our project is dependent can impact the implementation. Also, over a period of time, it becomes difficult to manage the various version of different types of solutions we have used to create our test automation project. For all this and more, we require Maven, an effective build management tool. Let us learn more about it in this chapter.

Need for build management

As we work with open source technologies like Selenium, there is a need to use other software as well as to support our test automation build. These open-source artifacts undergo version changes. Because most of the used software in building a test automation solution is open source, there is no liability on the Selenium creators or other open source solutions to provide any technical support to the end consumer. However, Selenium is a very popular and widely used open-source solution and has very wide community support and active group. But still, the management of your test automation build to always work in spite of version changes, and at the same

time ensuring that the version upgrade is seamless is on us. Also, a test automation solution needs to work on various environments; managing dependencies becomes a challenge if they have fixed locations. So in these situations, a build management tool helps. A build management solution has all the features to manage the life cycle of a project being built. And with continuous forms of tests involved, it becomes easier to find out if there is any break in the build because of integration issues.

Apache created Maven as an internal project, which they used it for the life cycle management of their solutions. Eventually, it was released as an open-source solution. We should note here that Maven is not the only build management solution; there are others, for example, ANT. But Maven is one of the most widely used solutions. To know more about Maven, we can visit the official page of it here: **https://maven.apache.org/**.

About Maven

Maven is a project management tool that can manage the different life cycles of a project, all based on the project object model, which is effectively an XML file. It can be used to manage projects for different programming languages. It is similar to a build management tool called as ANT, but it is more advanced than it. Maven allows us to create a project, manage dependencies, help with testing and final project deployment. Its seamless integration with CI/CD tools also helps in the DevOps process. Maven allows the creation of a one-stop project object model to manage and maintain project dependencies which it fetches from the Maven repository. It also comes with support to various integration with reporting tools, based on its SureFire Plugin, which allows execution of the unit tests. Let us now understand how to install Maven and create a simple Maven project.

Installing Maven in Eclipse

We can install Maven as a plugin in the eclipse IDE. To perform this action, we will be doing the following steps:

1. Open Eclipse IDE.

2. We need to click on **`Help`** and select **`Install New Software…`**, as shown in the following image:

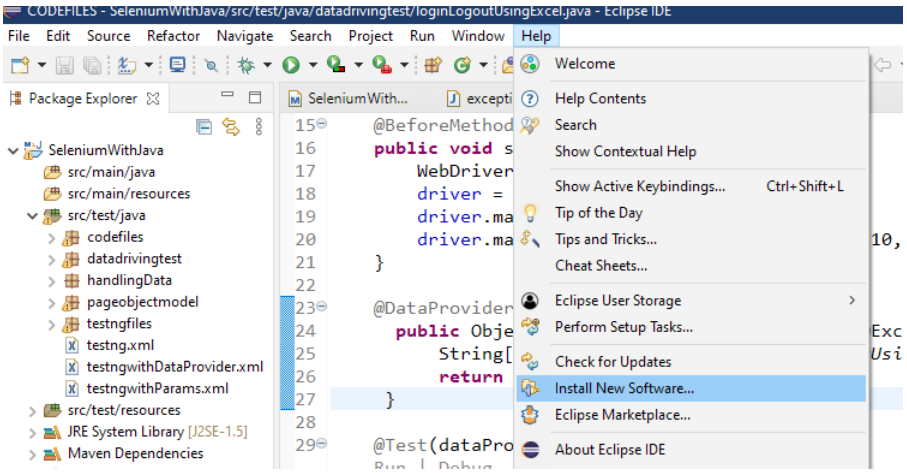


***Figure 11.1**: Installing Maven*

3. It will now open a new popup window, on which we will type the link to download Maven from. The link is as follows: **https://download.eclipse.org/technology/m2e/releases/latest/**

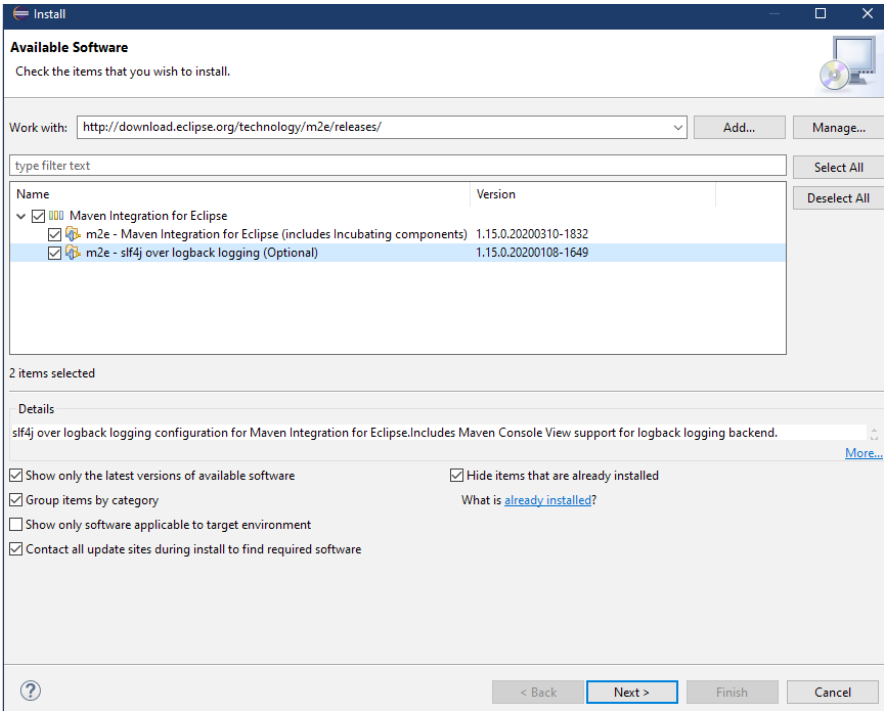


***Figure 11.2**: Maven integration for Eclipse*

4. We now have to click on the **`Next`** button to allow the installation to happen. It will show a page to accept the license agreement. Click on **`I accept`**.

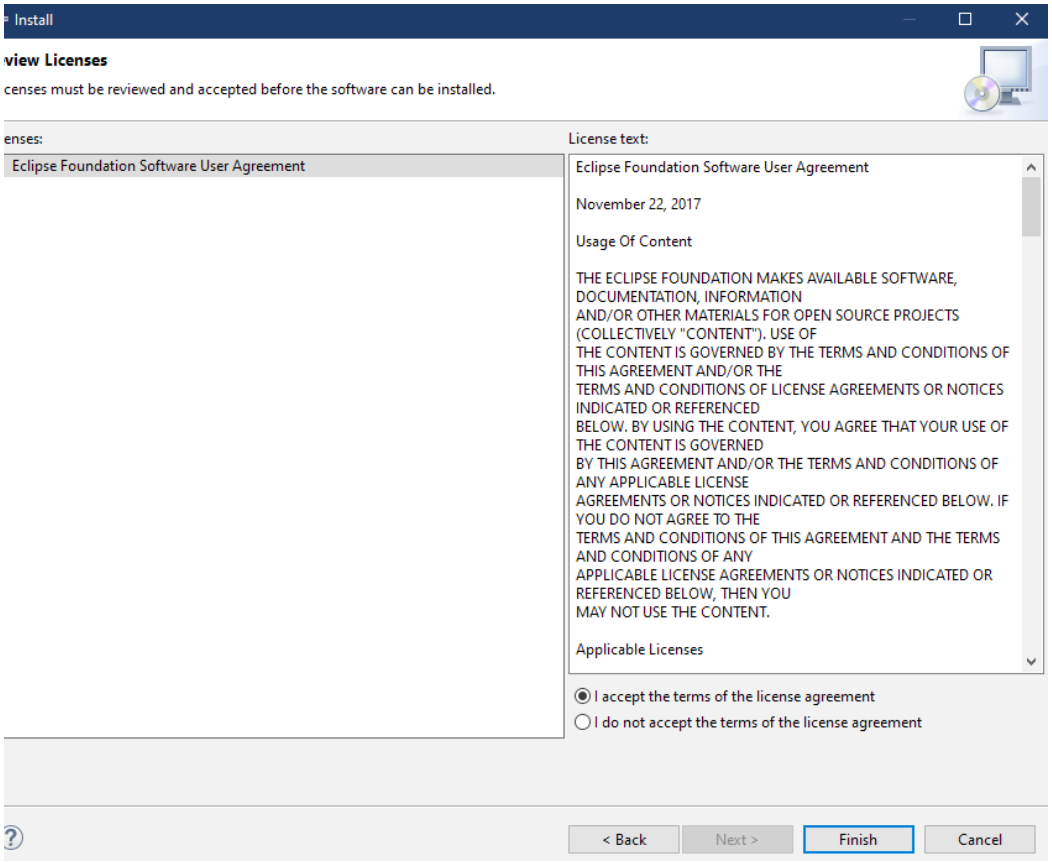


***Figure 11.3**: License agreement*

5. Click on the **`Finish`** button to complete the installation. It may take some time.

After the installation is done, we should be able to create a Maven project.

Creating a Maven project

To use Maven for project creation, we will have to first create a Maven project in Eclipse. Let us look at the steps to perform the action:

1. Go to **`File`**, and select **`New`**, and choose **`Project...`** from it:

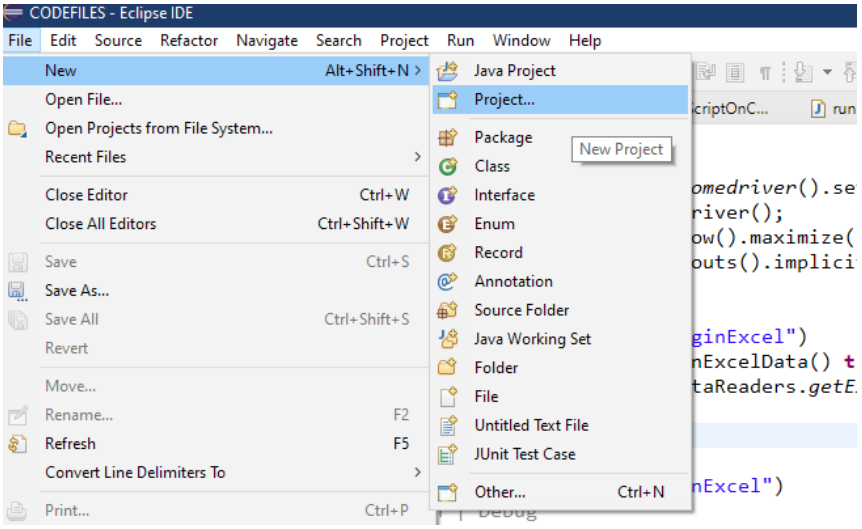


***Figure 11.4**: Creation of project*

2. This will open a **`New Project`** wizard. From there, select **`Maven`**, and click on **`Maven Project`**.

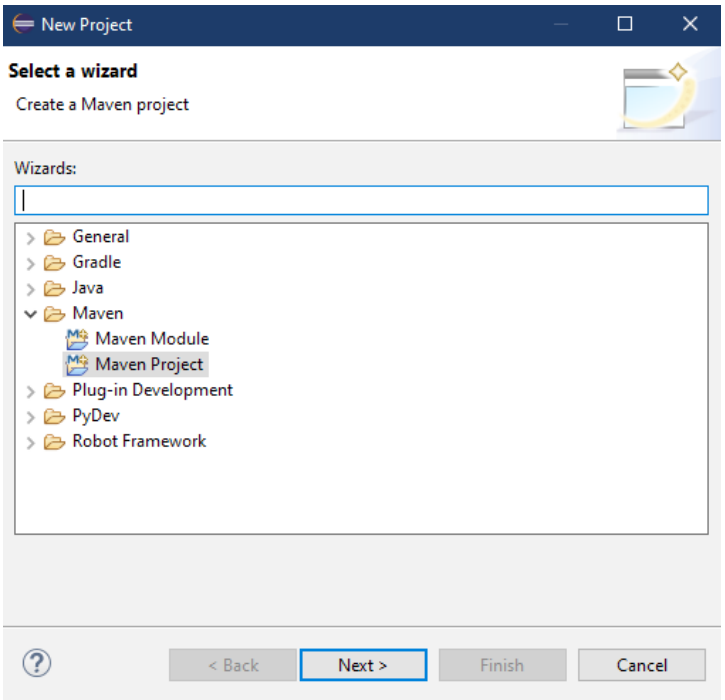


***Figure 11.5**: Maven project creation*

3. Click on the **`Next`** button, and select the checkbox to create a simple project and skip archetype selection.

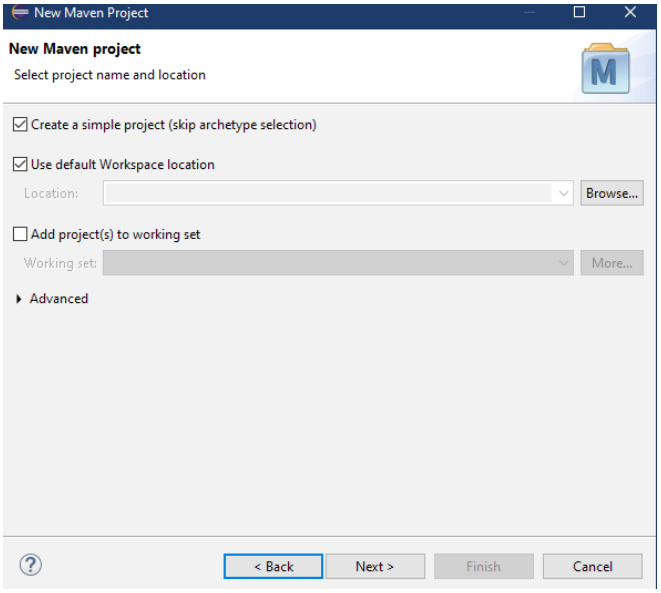


***Figure 11.6**: Simple project archetype*

4. When we click on the **`Next`** button, we will see a pop up to create a new Maven project:

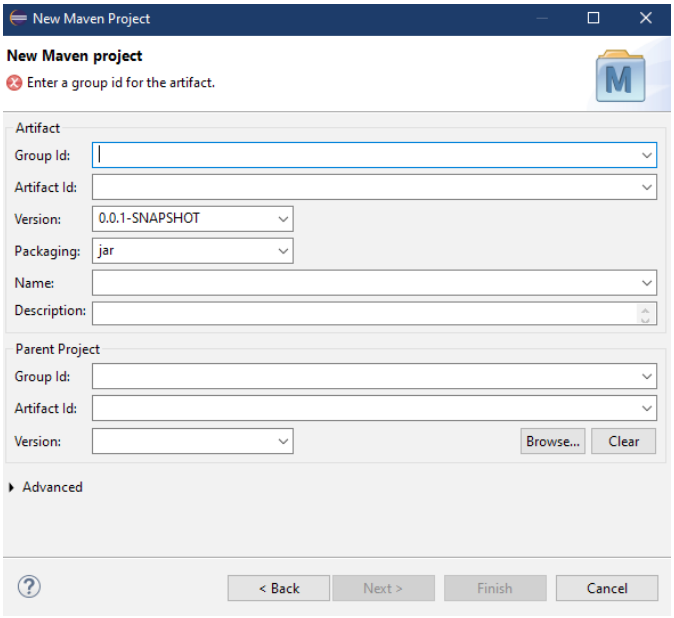


***Figure 11.7**: Maven project configuration*

In *figure 11.7*, we need to provide some important information which is explained as follows:

- **Group Id**: A name of the company/domain or group which creates the project. Similar to a Java package. We are providing **`org.bpb.com`**.
- **Artifact Id**: This is the name that will be given to the project. The name we are providing is **`demoMavenProject`**.

5. The rest of the fields can remain blank. And we click on the **`Finish`** button.

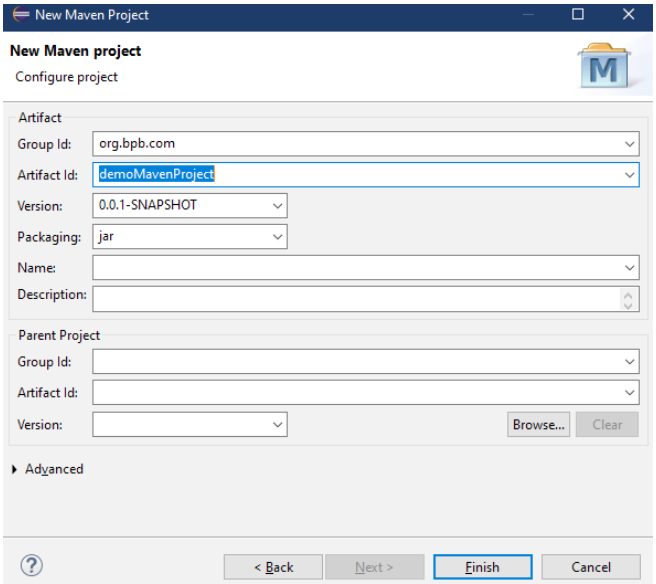


***Figure 11.8**: Group ID and Artifact ID*

6. As we click on **`Finish`**, we can see the Maven project created, to which, as required for our projects, we can add files.

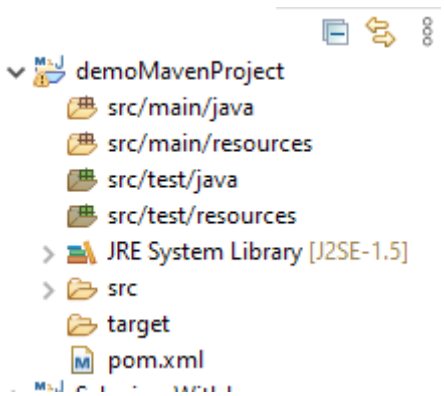


***Figure 11.9**: Maven project tree*

What is Maven repository?

Maven repository is a place that holds built artifacts and dependencies of various types. The location of the maven repository could be local or remote, where all dependencies are stored. In this chapter, we will see Maven central repository, which is a repository provided by the Maven community. It contains a large number of commonly used libraries. To visit it, we will use this link: **https://mvnrepository.com/**.

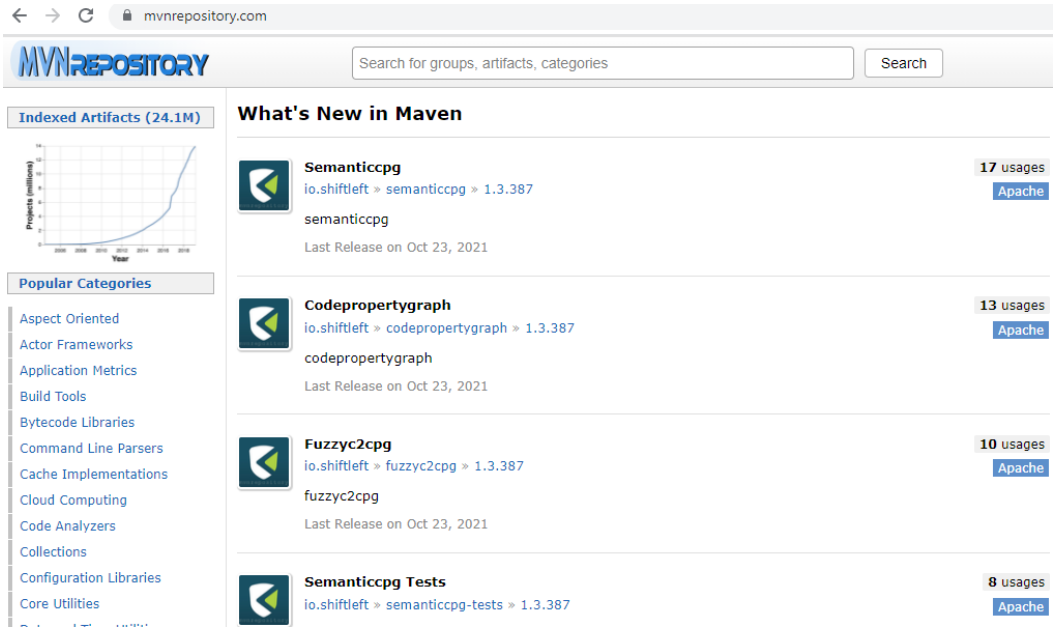


***Figure 11.10**: Maven repository*

For example, if we search for Selenium, this is what we will see:

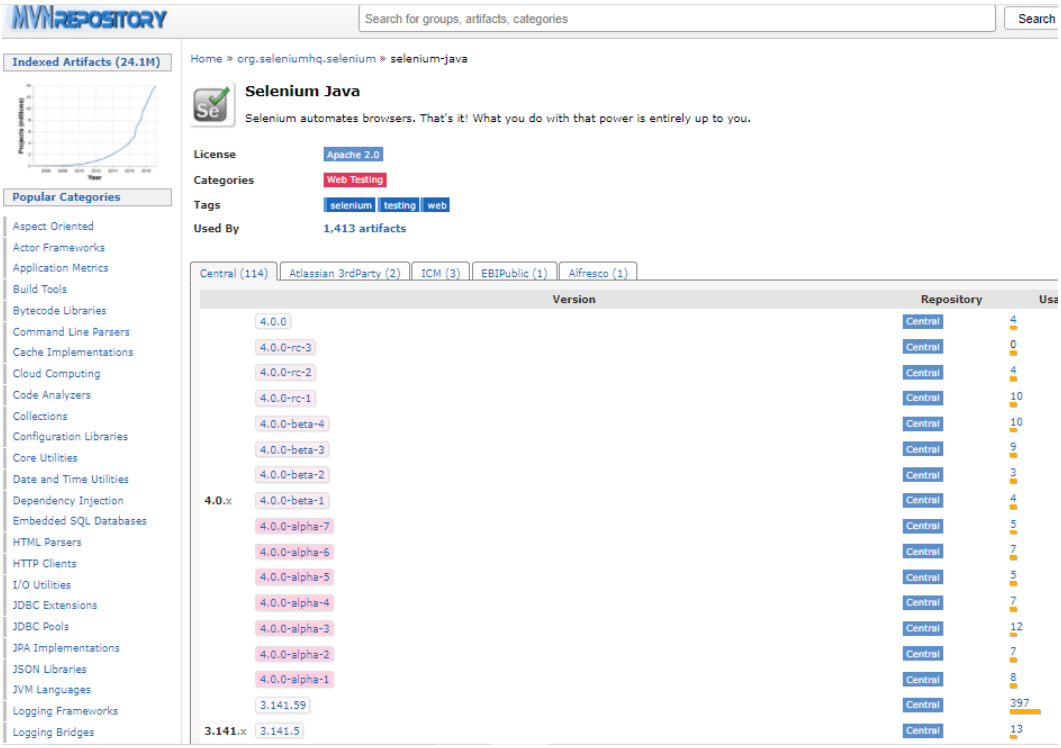


***Figure 11.11**: Selenium dependency JARs in Maven*

We can select any of the available builds in Selenium and copy the dependencies information to the **`pom.xml`**. The dependency information looks as follows for the version of Selenium used in this book which is 3.141.59:

```
<!-- https://mvnrepository.com/artifact/org.seleniumhq.selenium/selenium-java -->
<dependency>
    <groupId>org.seleniumhq.selenium</groupId>
    <artifactId>selenium-java</artifactId>
    <version>3.141.59</version>
</dependency>
```

***Figure 11.12**: Selenium dependency XML*

What is project object model?

A project object model is an XML file available with every Maven project. It is the basic unit of work in Maven. It contains all the information about the project, dependencies, and any configurations that would be required to build the maven project. The **`pom.xml`** is created as we create a Maven project.

The most basic information any pom file will contain is as follows:

```
<project>
  <modelVersion>4.0.0</modelVersion>

  <groupId>com.mycompany.app</groupId>
  <artifactId>my-app</artifactId>
  <version>1</version>
</project>
```

***Figure 11.13**: Default POM.xml*

Let us look at the **pom.xml** of our project, which we can see in the project tree.

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>org.bpb.com</groupId>
  <artifactId>demoMavenProject</artifactId>
  <version>0.0.1-SNAPSHOT</version>
</project>
```

***Figure 11.14**: Default POM.xml file*

A pom.xml contains many XML tags, which have their own purpose. One of the main XML tags, which we should know is **<dependencies>**. The **<dependencies>** is a parent tag, which can contain one or more than one child tags called as **<dependency>**. So if our project needs to use Selenium libraries, we can search in the central maven repository about Selenium and can add that information in the **pom.xml**. After that information is added, we can use the Selenium classes in our project.

In the same manner, whichever external JAR files our project will need it can be searched on the central repository and added to the pom.xml file.

Our main project, which we have been using throughout the book, is a Maven project, and its **pom.xml** carries the complete information that would be required to build, manage and configure the project. Let us have a look at that **pom.xml**.

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://
www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://maven.
apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>BPBPublication</groupId>
  <artifactId>SeleniumWithJava</artifactId>
  <version>0.0.1-SNAPSHOT</version>
   <properties>
    <project.build.sourceEncoding>UTF-8</project.build.
sourceEncoding>
  </properties>
```

```
 <dependencies>
    <!–https://mvnrepository.com/artifact/junit/junit -->
<dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.13.2</version>
    <scope>test</scope>
</dependency>

<!–https://mvnrepository.com/artifact/org.seleniumhq.selenium/
selenium-java -->
<dependency>
    <groupId>org.seleniumhq.selenium</groupId>
    <artifactId>selenium-java</artifactId>
    <version>3.141.59</version>
</dependency>
    <!–https://mvnrepository.com/artifact/commons-io/commons-io -->
<dependency>
    <groupId>commons-io</groupId>
    <artifactId>commons-io</artifactId>
    <version>2.10.0</version>
</dependency>
    <!–https://mvnrepository.com/artifact/io.github.bonigarcia/
webdrivermanager -->
<dependency>
    <groupId>io.github.bonigarcia</groupId>
    <artifactId>webdrivermanager</artifactId>
    <version>4.4.3</version>
</dependency>

<!–https://mvnrepository.com/artifact/com.opencsv/opencsv -->
<dependency>
    <groupId>com.opencsv</groupId>
    <artifactId>opencsv</artifactId>
    <version>5.5.2</version>
</dependency>
<!–https://mvnrepository.com/artifact/org.apache.poi/poi-ooxml -->
<dependency>
    <groupId>org.apache.poi</groupId>
    <artifactId>poi-ooxml</artifactId>
    <version>4.1.2</version>
</dependency>

  </dependencies>
</project>
```

Conclusion

Thus, in this chapter, we learned about a build management tool, Maven. We saw how to set it up in the eclipse environment. We saw how to create a simple maven project and work with the **`pom.xml`**. We also saw that to fetch the dependency information, we can search the central Maven repository and fetch the information that we can add to the **`pom.xml`** file.

In the upcoming chapter, we will talk about the Selenium component, called as Selenium grid.

Questions

1. What is Maven?
2. Name any other tool similar to Maven?
3. What is the purpose of pom.xml file?

Chapter 12
Selenium Grid

The Selenium project consists of multiple projects. One of the projects is Selenium Grid. Selenium Grid allows us to execute the tests in parallel on different machines or instances of machines. The test automation activity can be broken down into multiple stages: authoring, execution, scheduling, reporting, and so on. The Selenium Grid is used in the execution stage. We can set up either our own lab locally or on the cloud. Also, there are multiple paid test automation on cloud solutions available, which are mainly built over Selenium Grid. One of the well-known examples of test automation execution on the cloud is Sauce Labs, which is built over Selenium Grid. The execution of a test automation suite is mainly done on Selenium Grid. Many large and small companies use it widely for running their tests in parallel. In this chapter, we will learn about it.

Structure

Selenium Grid is popularly used for the execution of a test automation suite where tests can be executed in parallel on multiple instances of machines. It is one of the projects of Selenium, like Selenium IDE and the Selenium WebDriver. Similarly, Selenium Grid is a separate project in Selenium. It became extremely popular when Google shared using Selenium grid internally for the execution of its test automation suites of their internal projects. In this chapter, we will be covering the following concepts:

- What is Selenium Grid
- Components of Selenium Grid
- Executing tests in parallel in local environment using Selenium Grid

Objectives

The objective of this chapter is to understand about Selenium Grid. As mentioned earlier, Selenium Grid is one of the projects of Selenium and is used to execute tests in parallel on multiple instances of machines. This basically means that we can run a test on various machine browser combinations in parallel at the same time. Let us look at the sections of this chapter to understand more.

What is Selenium Grid

Selenium Grid is one of the projects of Selenium, which allows execution of the tests in parallel on multiple instances of machine browser combinations. Selenium Grid consists of two components, a hub and a node, which are used to set up the grid. Both hub and node are set up using the Selenium server, a component of the Selenium project. By setting up the grid, we allow the tests to run in parallel on various machine browser instances. Let us understand the two main components of the grid and how they are used to allow the execution of tests in parallel. The following diagram provides us with an insight:

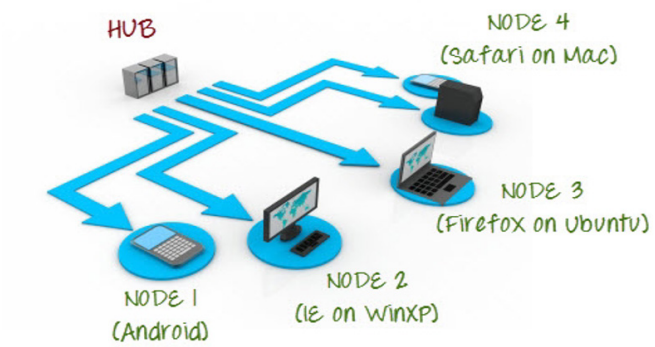


***Figure 12.1**: Grid representation*

Components of Selenium Grid

A Selenium grid consists of a node and a hub. And using them, we are able to execute tests in parallel. Let us define them:

- **Node**: a node is a machine browser instance on which the actual test execution will take place. A node could be a windows machine on which edge the browser is running. Or a node could be a mac machine on which the Safari browser is running. We need to understand here that a machine could also be a virtual machine on the cloud.
- **Hub**: a hub is a central component that is created by running the Selenium server using the flag—hub. A hub receives all the requests for the execution of the test. Based on the request received and the node registered with it, the hub sends the request to the node for the execution of the test. The actual execution of the test takes place on the node.

In this chapter, we will see the creation of a test script, and executing it on two different browsers at the same time, in a local machine.

Executing tests in local environment using Selenium Grid

To understand the working of a Selenium Grid, we will take an example in this chapter. We will create a test script and execute it on the local machine on two different browsers Firefox, and Chrome, at the same time. To achieve this, we will be performing the steps as follows:

1. Set up the hub on the local machine.
2. Create and register windows machine node with Firefox browser.
3. Create and run the test script using Selenium Grid on Firefox browser.
4. Create and register windows machine node with Chrome browser.
5. Create and run the test script using Selenium Grid on Chrome browser.
6. Execute test using **`testng`** suite file in parallel on both browsers.

Before we see the implementation of these steps, we should have an available Selenium server, which would be used for both hub and node. To download the Selenium server, we have to use this URL—**https://github.com/SeleniumHQ/selenium/releases/download/selenium-3.141.59/selenium-server-standalone-3.141.59.jar**/.

Please note that as of writing this book new Selenium release is available, which is Selenium 4.0. And the grid part of the release is entirely changed. The Selenium Grid version we are using in this book is 3.141.59. Save the downloaded JAR file in a location on your system. We will be using these JAR files to create the hub and the node.

Set up the hub on the local machine

The first step in setting the Selenium Grid is to start the hub. Let us do it by following these steps:

1. Open the command prompt, and go to the directory where the Selenium server JAR file is downloaded:

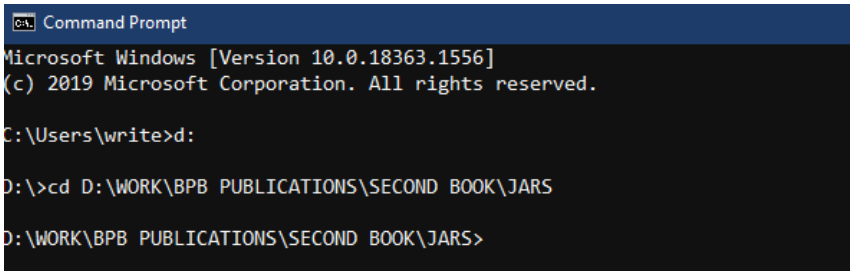


***Figure 12.2**: Change directory*

2. Start the hub by using the following command:

```
d:\> java -jar selenium-server-standalone-3.141.59.jar -role hub
```

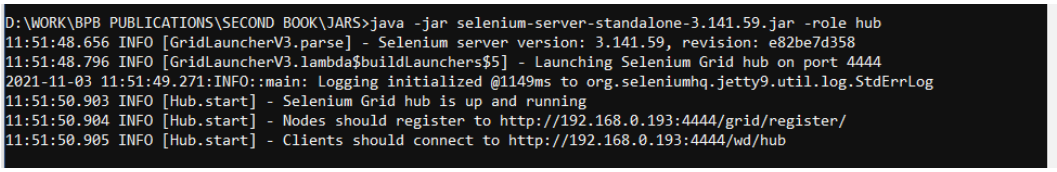


***Figure 12.3**: Starting hub*

3. The next step is to open the following URL on the browser to check if the hub is launched.

   URL—**http://localhost:4444/grid/console**

   If launched, you should be able to the following screen:

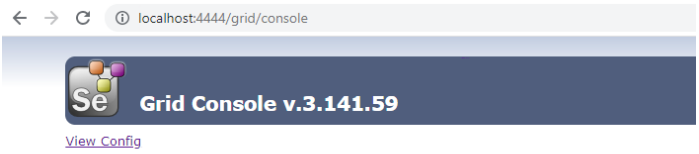


***Figure 12.4**: Hub page*

After launching the hub, we will create a node of the windows machine with Firefox browser. We will have to register it to the hub, which is also running locally. We need to note here that the default port the hub listens to is **-4444**, and if we want, we can

change the port by using the port flag and passing the port number.

Create and register windows machine node with Firefox browser

To register a windows machine with a Firefox browser node, we will do the following steps:

1. Download the correct version of the geckodriver on the system, based on the Firefox version you have on the system. To download geckodriver, use the link—**https://github.com/mozilla/geckodriver/releases**.
2. Open the command prompt and go to the directory where the Selenium server JAR file is downloaded:

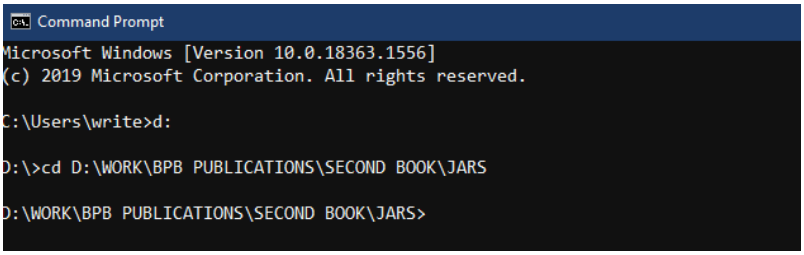


***Figure 12.5**: Change directory*

3. Start the node, using the following command:

```
java -Dwebdriver.gecko.driver="gecko\geckodriver.exe" -jar selenium-
server-standalone-3.141.59.jar -role node -hub http://localhost:4444/
grid/register -browser "browserName=firefox" -port 5556
```

```
D:\WORK\BPB PUBLICATIONS\SECOND BOOK\JARS>java -Dwebdriver.gecko.driver="gecko\geckodriver.exe" -jar selenium-server-standalone-3.14
1.59.jar -role node -hub http://localhost:4444/grid/register -browser "browserName=firefox" -port 5556
12:14:57.834 INFO [GridLauncherV3.parse] - Selenium server version: 3.141.59, revision: e82be7d358
12:14:57.915 INFO [GridLauncherV3.lambda$buildLaunchers$7] - Launching a Selenium Grid node on port 5556
2021-11-03 12:14:59.819:INFO::main: Logging initialized @2170ms to org.seleniumhq.jetty9.util.log.StdErrLog
12:15:00.025 INFO [WebDriverServlet.<init>] - Initialising WebDriverServlet
12:15:00.093 INFO [SeleniumServer.boot] - Selenium Server is up and running on port 5556
12:15:00.094 INFO [GridLauncherV3.lambda$buildLaunchers$7] - Selenium Grid node is up and ready to register to the hub
12:15:00.709 INFO [SelfRegisteringRemote$1.run] - Starting auto registration thread. Will try to register every 5000 ms.
12:15:01.908 INFO [SelfRegisteringRemote.registerToHub] - Registering the node to the hub: http://localhost:4444/grid/register
12:15:02.599 INFO [SelfRegisteringRemote.registerToHub] - The node is registered to the hub and ready to use
```

***Figure 12.6**: Start node Firefox*

Let us understand this command before we proceed further.

```
java -Dwebdriver.gecko.driver="gecko\geckodriver.exe" [this part
provides the path where geckdriver is available on the system]

-jar selenium-server-standalone-3.141.59.jar -role node [this starts the
selenium server in the role node]

 -hub http://localhost:4444/grid/register [this provides the point to
the hub location to which the node will get registered to. ]
```

```
 -browser "browserName=firefox" -port 5556 [this provides the information
about the browser, and the port on which the node will be listening to ]
```

4. To check if the node is registered to the hub, we will just refresh the page of the hub, which was opened using the URL—**http://localhost:4444/grid/console**

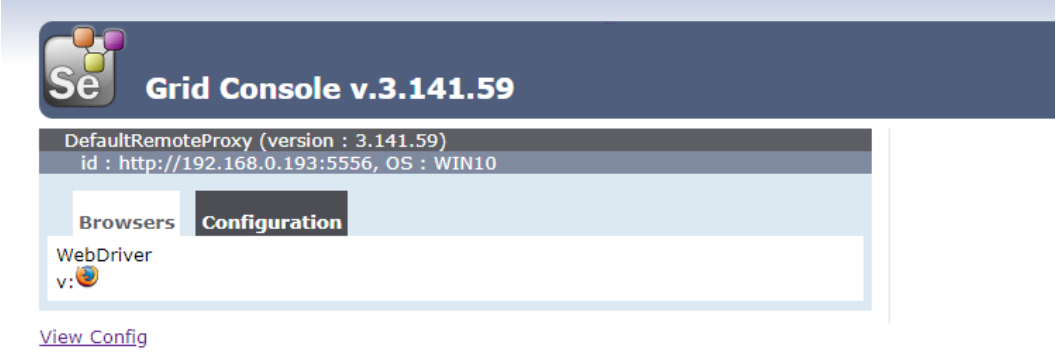


***Figure 12.7**: Hub with Firefox node registered.*

Create and run the test script using Selenium grid on Firefox browser

To create and run the test script on Selenium grid using the Firefox browser, we will have to create an instance of the RemoteWebDriver, instead of the WebDriver. And as we create the instance of the remote WebDriver, we will also be using the **`DesiredCapabilities`** class to pass the browser information as Firefox. Let us see the following code snippet for it:

```
@BeforeMethod
  public void setUp() throws Exception {
          DesiredCapabilities capability = DesiredCapabilities.
firefox();
          driver = new RemoteWebDriver(new URL("http://localhost:4444/
wd/hub"), capability);
          driver.manage().timeouts().implicitlyWait(30, TimeUnit.
SECONDS);
  }
```

In this code snippet, as we can see, we create the desired capabilities object for the Firefox browser. This is passed to the RemoteDriver instance created. The instance also takes the location of the hub. Please note that the location of the hub here is localhost:**4444**. This means that the hub is running locally on port **4444**. If our hub is running on a different machine instance, we will pass the information of that machine URL and the port address on which the hub is running. Once the driver

instance is created, the rest of the script remains the same. So, when we execute the script, the request will be sent to the hub. The hub will then pass on the request to the node, on which the actual execution as per the test script will take place.

The test script we will be using here is of the login logout scenario on the BPB practice application—**http://practice.bpbonline.com/**. The scenario is explained in the chapters earlier.

The complete test script is as follows when we run it using the **`testng.xml`** file, the request will be sent to the hub. The hub will pass the same to the node which is running the Firefox browser. It is there at the node where the actual execution takes place.

```
package seleniumgrid;

import java.net.URL;
import java.util.concurrent.TimeUnit;
import org.testng.annotations.*;
import org.openqa.selenium.*;
import org.openqa.selenium.remote.DesiredCapabilities;
import org.openqa.selenium.remote.RemoteWebDriver;

public class LoginLogoutWithFF {
  WebDriver driver;

  @BeforeMethod
  public void setUp() throws Exception {
          DesiredCapabilities capability = DesiredCapabilities.
firefox();
          driver = new RemoteWebDriver(new URL("http://
localhost:4444/wd/hub"), capability);
          driver.manage().timeouts().implicitlyWait(30, TimeUnit.
SECONDS);
  }
 @Test
  public void testRecordedLogin() throws Exception {
    driver.get("http://practice.bpbonline.com/");
    driver.findElement(By.linkText("My Account")).click();
    driver.findElement(By.name("email_address")).clear();
    driver.findElement(By.name("email_address")).sendKeys("bpb@bpb.
com");
    driver.findElement(By.name("password")).clear();
    driver.findElement(By.name("password")).sendKeys("bpb@123");
    driver.findElement(By.id("tdb1")).click();
     driver.findElement(By.linkText("Log Off")).click();
     driver.findElement(By.linkText("Continue")).click();
   }
```

```
  @AfterMethod
  public void tearDown() throws Exception {
    driver.quit();
  }
}
```

If you note the set of activities at the command prompt of the hub and the node, we will see the information of the session creation at the node and hub passing the request. The following logs show us that:

Logs at hub

```
12:35:46.419 INFO [RequestHandler.process] - Got a request to create
a new session: Capabilities {acceptInsecureCerts: true, browserName:
firefox, version: }

12:35:46.422 INFO [TestSlot.getNewSession] - Trying to create a
new session on test slot {server:CONFIG_UUID=fa81d181-5c13-4ac2-
b39a-cbdbb7e57895, seleniumProtocol=WebDriver, browserName=firefox,
platformName=WIN10, platform=WIN10}

Logs at node

12:35:46.510 INFO [ActiveSessionFactory.apply] - Capabilities are: {
  "acceptInsecureCerts": true,
  "browserName": "firefox",
  "version": ""
}

12:35:46.512 INFO [ActiveSessionFactory.lambda$apply$11] - Matched
factory org.openqa.selenium.grid.session.remote.ServicedSession$Factory
(provider: org.openqa.selenium.firefox.GeckoDriverService)

1635923146670   geckodriver     INFO    Listening on 127.0.0.1:35267

1635923147076   mozrunner::runner       INFO    Running command: "C:\\
Program Files\\Mozilla Firefox\\firefox.exe" "--marionette" "-no-remote"
"-profile" "C:\\Users\\write\\AppData\\Local\\Temp\\rust_mozprofileoHoDlT"

1635923148049   Marionette      INFO    Marionette enabled

console.warn: SearchSettings: "get: No settings file exists, new profile?"
(new NotFoundError("Could not open the file at C:\\Users\\write\\
AppData\\Local\\Temp\\rust_mozprofileoHoDlT\\search.json.mozlz4", (void
0)))

1635923150514   Marionette      INFO    Listening on port 58568

1635923150722   Marionette      WARN    TLS certificate errors will be
ignored for this session
```

```
12:35:50.739 INFO [ProtocolHandshake.createSession] - Detected dialect:
W3C
12:35:50.760 INFO [RemoteSession$Factory.lambda$performHandshake$0] -
Started new session 5586d987-728a-4b01-bc92-99889b5e4ede (org.openqa.
selenium.firefox.GeckoDriverService)
1635923176066   Marionette      INFO    Stopped listening on port 58568
[Child 50352, IPC I/O Child] WARNING: pipe error: 232: file /builds/
worker/checkouts/gecko/ipc/chromium/src/chrome/common/ipc_channel_win.
cc:544
```

Create and register windows machine node with Chrome browser

To register the node for the Chrome browser, for the windows machine running locally we will be doing the following steps:

1. Download the Chrome driver, as per the Chrome browser version available on the system. Use the URL—**https://chromedriver.chromium.org/**.
2. The next step is to create a node, for which we will first have to open the command prompt, change the directory to the location of the Selenium server, and type the following command:

```
java -Dwebdriver.chrome.driver="chromedriver\chromedriver.exe" -jar
selenium-server-standalone-3.141.59.jar -role node -hub http://
localhost:4444/grid/register -browser "browserName=chrome" -port 5557
```

```
D:\WORK\BPB PUBLICATIONS\SECOND BOOK\JARS>java -Dwebdriver.chrome.driver="chromedriver\chromedriver.exe" -jar selenium-server-standalone-3.141.
59.jar -role node -hub http://localhost:4444/grid/register -browser "browserName=chrome" -port 5557
13:20:54.201 INFO [GridLauncherV3.parse] - Selenium server version: 3.141.59, revision: e82be7d358
13:20:54.286 INFO [GridLauncherV3.lambda$buildLaunchers$7] - Launching a Selenium Grid node on port 5557
2021-11-03 13:20:56.274:INFO::main: Logging initialized @2259ms to org.seleniumhq.jetty9.util.log.StdErrLog
13:20:56.451 INFO [WebDriverServlet.<init>] - Initialising WebDriverServlet
13:20:56.513 INFO [SeleniumServer.boot] - Selenium Server is up and running on port 5557
13:20:56.513 INFO [GridLauncherV3.lambda$buildLaunchers$7] - Selenium Grid node is up and ready to register to the hub
13:20:57.172 INFO [SelfRegisteringRemote$1.run] - Starting auto registration thread. Will try to register every 5000 ms.
13:20:58.037 INFO [SelfRegisteringRemote.registerToHub] - Registering the node to the hub: http://localhost:4444/grid/register
13:20:58.623 INFO [SelfRegisteringRemote.registerToHub] - The node is registered to the hub and ready to use
```

***Figure 12.8**: Start the node for Chrome browser*

As we did for the Firefox node, let us also understand the Chrome node command

```
java -Dwebdriver.chrome.driver="chromedriver\chromedriver.exe" [this
part provides the path for the chromedriver, which will help in the
chrome browser execution]

 -jar selenium-server-standalone-3.141.59.jar -role node  [this part
starts the SSelenium sever in the role node. ]

 -hub http://localhost:4444/grid/register [this part provides the path
of the hub to which the node for chrome will be registered. ]

 -browser "browserName=chrome" -port 5557 [this part provides the
browser information, and the port on which it will be executed]
```

To check if the Chrome browser node is registered to the hub, we will check the URL again—**http://localhost:4444/grid/console**.

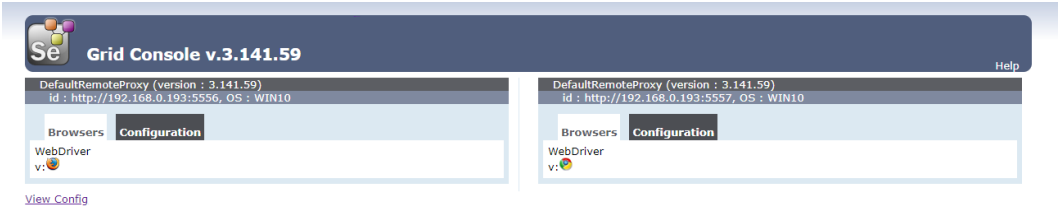


***Figure 12.9**: Hub showing nodes registered*

Create and run the test script using Selenium Grid on Chrome browser

To run the test on the Chrome browser, we will perform the steps similar to the execution for Firefox, which we saw before this. We will create the **`DesiredCapabilities`** object for the Chrome browser. This will be passed along with the hub URL for the RemoteDriver object instance. For execution on Selenium Grid, instead of WebDriver instance, we create the RemoteWebDriver instance.

The following script shows as follows:

```
@BeforeMethod
  public void setUp() throws Exception {
          DesiredCapabilities capability = DesiredCapabilities.
chrome();
          driver = new RemoteWebDriver(new URL("http://localhost:4444/
wd/hub"), capability);
          driver.manage().timeouts().implicitlyWait(30, TimeUnit.
SECONDS);
  }
```

The rest of the test script is of login logout on the BPB practice application, which does not changes. The complete script which will run on the Selenium Grid for Chrome browser is as follows:

```
package seleniumgrid;

import java.net.URL;
import java.util.concurrent.TimeUnit;
import org.testng.annotations.*;
import org.openqa.selenium.*;
import org.openqa.selenium.remote.DesiredCapabilities;
import org.openqa.selenium.remote.RemoteWebDriver;

public class LoginLogoutWithChrome {
  WebDriver driver;

  @BeforeMethod
  public void setUp() throws Exception {
          DesiredCapabilities capability = DesiredCapabilities.
chrome();
          driver = new RemoteWebDriver(new URL("http://localhost:4444/
wd/hub"), capability);
          driver.manage().timeouts().implicitlyWait(30, TimeUnit.
SECONDS);
  }

  @Test
  public void testRecordedLogin() throws Exception {
    driver.get("http://practice.bpbonline.com/");
    driver.findElement(By.linkText("My Account")).click();
    driver.findElement(By.name("email_address")).clear();
    driver.findElement(By.name("email_address")).sendKeys("bpb@bpb.
com");
    driver.findElement(By.name("password")).clear();
    driver.findElement(By.name("password")).sendKeys("bpb@123");
    driver.findElement(By.id("tdb1")).click();
     driver.findElement(By.linkText("Log Off")).click();
     driver.findElement(By.linkText("Continue")).click();
   }

  @AfterMethod
  public void tearDown() throws Exception {
    driver.quit();
  }
}
```

The preceding script will send the command to the hub, which is running on the local machine and listening to port **4444**. The hub will pass on the test information to the node, which has the Chrome browser running. The actual execution of the test

will take place on the node. The logs for the hub and the node shows the information of creation of the session and execution.

Logs of the hub

```
13:32:52.935 INFO [RequestHandler.process] - Got a request to create a
new session: Capabilities {browserName: chrome, version: }

13:32:52.936 INFO [TestSlot.getNewSession] - Trying to create a
new session on test slot {server:CONFIG_UUID=a580cce7-d91d-42ce-
9947-3d64cbcb1611, seleniumProtocol=WebDriver, browserName=chrome,
platformName=WIN10, platform=WIN10}

Logs of the node

13:32:52.977 INFO [ActiveSessionFactory.apply] - Capabilities are: {
  "browserName": "chrome",
  "goog:chromeOptions": {
  },
  "version": ""
}

13:32:52.978 INFO [ActiveSessionFactory.lambda$apply$11] - Matched
factory org.openqa.selenium.grid.session.remote.ServicedSession$Factory
(provider: org.openqa.selenium.chrome.ChromeDriverService)

Starting ChromeDriver 95.0.4638.54 (d31a821ec901f68d0d34ccdbaea45b4c86ce
543e-refs/branch-heads/4638@{#871}) on port 8174

Only local connections are allowed.

Please see https://chromedriver.chromium.org/security-considerations for
suggestions on keeping ChromeDriver safe.

ChromeDriver was started successfully.

13:32:54.173 INFO [ProtocolHandshake.createSession] - Detected dialect:
W3C

13:32:54.196 INFO [RemoteSession$Factory.lambda$performHandshake$0]
- Started new session ea912bfb2e1efa1cc4b488c5c48cc49d (org.openqa.
selenium.chrome.ChromeDriverService)

13:33:17.169 INFO [ActiveSessions$1.onStop] - Removing session
ea912bfb2e1efa1cc4b488c5c48cc49d (org.openqa.selenium.chrome.
ChromeDriverService)
```

So as the test executes, we will see the Chrome browser gets launched, and the test for login logout being executed.

In the next section, we will run the tests on Chrome and Firefox in parallel.

Execute test using TestNG suite file in parallel on both browsers

In the preceding section, we saw the creation of the Selenium Grid. We started the hub and registered two nodes on its Chrome and Firefox. We also individually executed tests on them of the login logout scenario on both Chrome and the Firefox browser. Looking at the logs, we verified that the hub received the request for session creation, and then it passed the test to the correct node, which provides the environment for execution. Now, we will run these two tests in parallel.

To execute the test on the Chrome browser and the Firefox browser, in parallel, we will be making changes in the **`testng.xml`** file. The **`testng.xml`** file used will be as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<suite parallel="classes" name="DemoSuite">

  <test name="LoginLogout">
    <classes>
    <class name="seleniumgrid.LoginLogoutWithFF"/>
      <class name="seleniumgrid.LoginLogoutWithChrome"/>
    </classes>
  </test> <!-- Test -->
</suite> <!-- Suite -->
```

As we run the suite file, we will see both Chrome and Firefox browsers being launched and the test execution on them:

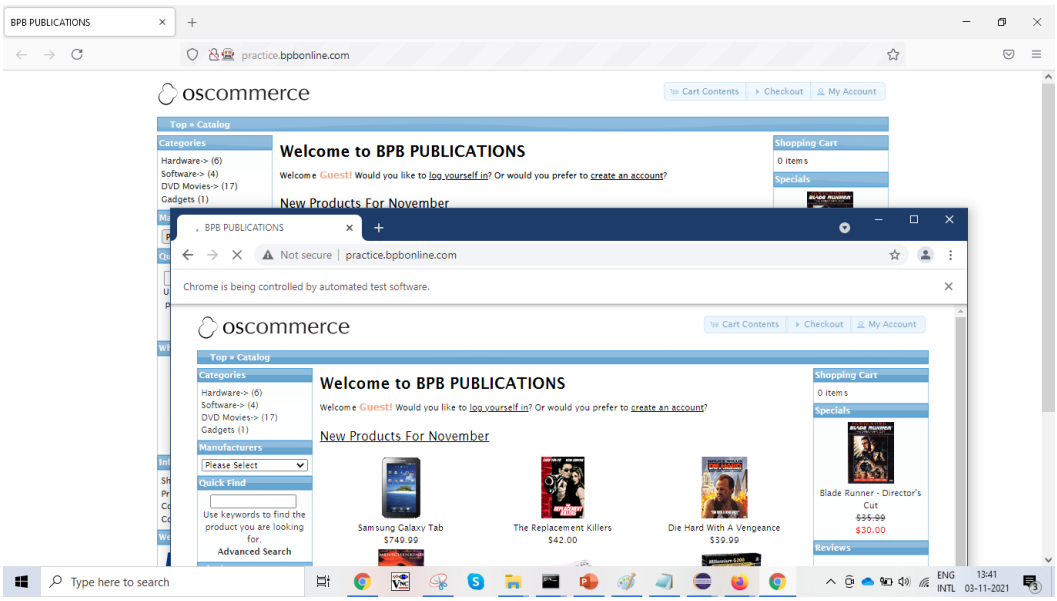


***Figure 12.10**: Execution on Chrome and Firefox*

Selenium Grid also provides the complete set-up on Docker. So we can directly use the Docker environment to set up the grid, which consists of node and hub. And then use it to run our tests. To learn more about it, visit the URL—**https://github.com/SeleniumHQ/docker-selenium**.

We can also use the various cloud solutions available such as Sauce Labs, BrowserStack, and LamdaTest, which all uses Selenium Grid files to set up a lab on the cloud, and thus, save us the hassle of maintaining a local test automation lab.

Conclusion

In this chapter, we learned about the Selenium Grid component of Selenium. We understood what is a hub and a node. We saw the creation of the hub and the nodes for Chrome and Firefox browsers. The hubs and nodes were running locally on the same machine, on which we executed individually and then, in parallel, the login logout scenario.

Questions

1. What is Selenium Grid used for?
2. Can you set up Selenium Grid using Docker?
3. On which port by default does the hub listens to?

Index