



# **Selenium Framework**

— Design in —

## **Keyword-Driven Testing**

Automate Your Test Using Selenium and Appium



PINAKIN ASHOK CHAUBAL



**Selenium Framework**  
**Design in**  
**Keyword-Driven**  
**Testing**

---

*Automate Your Test Using Selenium and Appium*

---

*by*

**Pinakin Ashok Chaubal**



**FIRST EDITION 2020**

**Copyright © BPB Publications, India**

**ISBN: 978-93-89328-20-2**

All Rights Reserved. No part of this publication may be reproduced or distributed in any form or by any means or stored in a database or retrieval system, without the prior written permission of the publisher with the exception to the program listings which may be entered, stored and executed in a computer system, but they can not be reproduced by the means of publication.

#### **LIMITS OF LIABILITY AND DISCLAIMER OF WARRANTY**

The information contained in this book is true to correct and the best of author's & publisher's knowledge. The author has made every effort to ensure the accuracy of these publications, but 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.

#### **Distributors:**

**BPB PUBLICATIONS**

20, Ansari Road, Darya Ganj

New Delhi-110002

Ph: 23254990/23254991

### **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

### **DECCAN AGENCIES**

4-3-329, Bank Street,

Hyderabad-500195

Ph: 24756967/24756400

### **BPB BOOK CENTRE**

376 Old Lajpat Rai Market,

Delhi-110006

Ph: 23861747

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**

*Arnav*

*My son, who brought sunshine to my life.*

---

### ***About the Author***

**Pinakin Ashok Chaubal** is an Automation specialist working in Intellect Design Arena Ltd. He has 19+ years of experience in the IT industry and has worked with IT giants like Accenture and L&T Infotech Ltd. He has worked in the US for six years for Travelers and in Hong-Kong China for one year for HSBC. He has been involved in designing and maintaining several test automation frameworks like Page Object Model, Keyword Driven frameworks, Cucumber BDD frameworks, and TestNG based frameworks. He guides and mentors project teams on adopting Test Automation for their projects.

Pinakin has written two books on Selenium WebDriver and has a Youtube channel that teaches various concepts related to Test Automation to people. Pinakin also has a 3-in-1 video course on Jmeter on one commercial website. Currently, he is creating a blogsite for various concepts related to Selenium WebDriver.

## ***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 my parents, who supported me throughout.

This book wouldn't have happened if I hadn't had the support from Nitin Chourey, who helped in reviewing this book and provide valuable feedback.

Finally, I would like to thank BPB Publications for giving me this opportunity to write my first book for them.

## *Preface*

Test Automation has emerged as a savior in terms of saving manual testing time. Test automation reduces the cost of the project by leveraging testing for Regression, Smoke, and Sanity. It also helps in less manual intervention and thus helps in test execution in off-hours.

Selenium WebDriver is an open-source test automation tool that can be used efficiently for test automation. It is the market leader in test automation and proves to be very powerful in terms of automating application testing quickly. With the emergence of Agile and Scrum, Selenium WebDriver proves to be very efficient in reducing the iteration or sprint time, thus making it faster for the application to be made live.

This book is focusing on quickly learning Selenium concepts and, at the same time, creating a framework parallelly. I have shown how to create a Keyword driven framework explaining every concept in detail. We start with the basics and get into more complicated stuff with each chapter. Each chapter has exercises that will help you in understanding the concepts clearly.

Over the 14 chapters in this book, you will learn the following:

### **Chapter 1: First look at Selenium WebDriver and WebElements**

introduces Test automation and states why it is required. It explores the different test automation tools available, differences

between Selenium WebDriver versions 2 and 3, the WebDriver architecture, understand the various concepts like WebDriver and WebElements. Eventually, we know the basic operations on WebElements.

[Chapter 2: Looking at the various WebDrivers](#) starts exploring WebDriver concepts like driver instances, instantiating Firefox browser sessions, what is headless firefox, Instantiating chrome sessions, what is headless chrome, instantiating Internet Explorer, Microsoft Edge, and Safari browsers. We understand what a WebDriverManager is used for. Finally, we see the setup of the WebDriverManager class.

[Chapter 3: A brief look at Java 8](#) looks at a brief overview of Java 8. Concepts such as functional programming, lambda expressions, Streams, Map function, method references, and filtering web elements from the array list are explored.

[Chapter 4: Deep dive into Selenium WebDriver](#) is a deep dive in Selenium WebDriver. It covers concepts like ThreadLocal, Singleton pattern, and creating a Singleton Driver for WebDriver instances, Handling pop-up windows, alerts, and frames, Waiting for elements to load using Explicit wait, understanding fluent delays, Handling page loads and Ajax call completion.

[Chapter 5: Actions class and the Javascript Executor](#) is a chapter on Advanced interactions with Selenium WebDriver. This chapter teaches advanced communications, actions class, creating a reusable class for Actions, understanding the JavascriptExecutor,

and building a generic Javascript utility class for the framework and integrating with the frame.

[\*\*Chapter 6: WebDriver Events\*\*](#) introduces the reader to WebDriver Events. This chapter covers what WebDriver events are, the process of handling events, understand how EventFiringWebDriver and WebDriverEventListener work, registering single and multiple listeners with the EventFiringWebDriver, Un-registering the listener, Understand Log4J API, different methods in the WebDriverEventListener interface and integration with the framework.

[\*\*Chapter 7: Database Operations\*\*](#) introduces the reader to database operations, and covers Learn the various CRUD operations, Create a properties file for database configurations, Create queries in MySql workbench to retrieve data from the four tables, Understand what a cached RowSet is Create a class for database operations, Integrating the new class with the framework

[\*\*Chapter 8: Get Introduced to TestNG\*\*](#) introduces the TestNG framework, and covers Learn what a testing framework means, Get added to TestNG, Understand the TestNG xml, Learn the different annotations that TestNG provides, Understand the concept of Test classes and Learn what Test Suites are.

[\*\*Chapter 9: Parallel Execution\*\*](#) explains the concept of parallel execution with the Selenium webdriver. This chapter covers Introduction to TestNG groups, Learn how to execute tests in a test suite in serial fashion, Get introduced to various options available for parallel execution in TestNG xml, to create three tests

which perform the same class but with different parameters and Fetching test cases from the database based on these parameters

[Chapter 10: Understanding Maven](#) introduces Maven as a build tool. This chapter covers Introduction to Maven, Setting-up Maven, and Maven build Lifecycle, Maven command line calls, Goals in Maven, Packaging, Plug-ins, Triggering tests from TestNG xml, Using Dataprovider instead of For loops Using Assertions, Incorporating Extent Reports and Introduction to Git and GitHub

[Chapter 11: Jenkins Introduction and Scheduling](#) This chapter introduces the reader to Jenkins as a Build Automation tool. This chapter covers Setting-up Jenkins, Executing the Maven build from the command line, create and execute a Jenkins job and Scheduling a Jenkins job

[Chapter 12: Selenium grid and executing in the cloud](#) This chapter talks about Selenium Grid and executing in the cloud. This chapter covers Introducing RemoteWebDriver, Learning about Selenium StandAlone Server, Learning about the RemoteWebDriver Client, Steps to convert a regular script to use RemoteWebDriver Server, Looking at the Hub, Knowing the Node, Hub Configuration Parameters, Node Configuration Parameters, Specifying configuration using JSON files. Changes to the SingletonDriver class, Introducing BrowserStack, Setting-up BrowserStack

[Chapter 13: Mobile test automation using Appium](#) This chapter talks about Appium as a tool for mobile automation. This chapter covers Types of Mobile applications, Introducing Appium, Learning

the Appium architecture, Setting up Appium, Changes to pom.xml for Appium and Changes to the framework

[Chapter 14: A look at Selenium-4](#) Introduces the reader to newly introduced features of Selenium 4

***Downloading the code***

***bundle and coloured images:***

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

<https://rebrand.ly/f242d>

***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, 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](mailto:errata@bpbonline.com)

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

## ***Table of Contents***

### **1. First Look at Selenium WebDriver and WebElements**

Structure

Objective

Tools available for test automation

What's new in Selenium 3.X

APIs that conform to W3C

Advanced functionalities supported

Introduction of Appium project

Selenium WebDriver architecture

Selenium Server

Selenium IDE

Main interfaces of Selenium WebDriver

Property or attribute of WebElements

Accessing WebElements

Accessing the various attributes of WebElements

Maven

Creating a Maven project in Eclipse

Manual configuration

Introduction to the testingautomation framework

Keyword -driven framework

Introduction to JSON

Database table structures

Basic insert operation on JSONArray

Conclusion

Questions

### **2. Looking at the Various WebDrivers**

## Structure

### Objectives

What is a driver instance?

Firefox Driver

What is a Firefox Profile?

Adding an extension

Storing and retrieving Profiles

Understanding Firefox preferences

Setting preferences

Firefox in headless mode

Chrome driver

Chrome options

Chrome extensions

Chrome in headless mode

Internet Explorer Driver

IEDriver capabilities

Microsoft Edge Driver

Safari Driver

A small code example

Introducing WebDriverManager library

How to use the WebDriverManager library

WebDriverManager as a Java dependency

WebDriverManager server

Conclusion

Questions

## 3. A Brief Look at Java 8

Structure

Objectives

What is a functional programming

Pure functions

Higher-order functions

Functions as parameters

Functions as return values

Functions as first class objects

Functional interface

Lambda expressions

Streams

Stream operations

Intermediate operations

Terminal operations

Filtering WebElements and extracting the count

Fetching elements based on attributes

Introducing method references

Referencing a static method

Referencing an instance method

Referencing a constructor

Performing actions on WebElements

How to use the map function to get the text value of elements

Conclusion

Questions

## 4. Deep Dive into Selenium WebDriver

Structure

Objectives

ThreadLocal class

Using ThreadLocal

Framework design

Framework architecture

ActionKeywords class

FrameworkDriver class

[Singleton pattern?](#)

[Creating a Singleton driver](#)

[Handling pop-up windows](#)

[Window handles](#)

[The Set interface](#)

[Handling Alerts](#)

[Steps to handle Alerts](#)

[Handling frames/iFrames](#)

[Steps to handle frames/iFrames](#)

[Steps to handle nested frames/iFrames](#)

[What is synchronization?](#)

[Synchronization at WebDriver instance level](#)

[Implicit wait](#)

[PageLoad Timeout](#)

[Synchronization at WebElement level](#)

[Explicit wait](#)

[Fluent wait](#)

[Handling AJAX calls](#)

[Differences between get\(\) and navigate\(\).](#)

[Conclusion](#)

[Questions](#)

## [5. Actions Class and the Javascript Executor](#)

[Structure](#)

[Objectives](#)

[What are advanced interactions?](#)

[Understanding the Actions class](#)

[A small example to demonstrate the Actions class](#)

[Drag and drop example to demonstrate the Actions class](#)

[Pressing a key combination example](#)

[Click and hold action](#)

[The JavascriptExecutor class](#)

[Creating customized classes for Actions and JavascriptExecutor](#)

[Integrating with the framework](#)

[Conclusion](#)

[Questions](#)

## **6. WebDriver Events**

[Structure](#)

[Objectives](#)

[What are the events in Selenium WebDriver?](#)

[Introducing the EventFiringWebDriver class](#)

[Introducing the WebDriverEventListener interface](#)

[Introducing the AbstractWebDriverEventListener class](#)

[Process of event handling](#)

[Extending the AbstractWebDriverEvent Listener abstract class](#)

[Registering more than one listener to the EventFiringWebDriver instance](#)

[Introducing the Log4J framework](#)

[Steps to include Log4J in code](#)

[Integrating with the framework](#)

[Different WebDriver event listeners](#)

[WebElementsearch](#)

[WebElement click](#)

[WebElement value changes](#)

[Navigating back](#)

[Navigate forward](#)

[Navigate to](#)

[Script execution](#)

[Listening for exception](#)

[Conclusion](#)

[Questions](#)

## [7. Database Operations](#)

[Structure](#)

[Objectives](#)

[CRUD operations?](#)

[Creating the database and tables](#)

[Create testconfig table](#)

[Create testcases table](#)

[Create testdata table](#)

[Create object repository table](#)

[Understanding JOINS](#)

[Inner Join](#)

[Outer Joins](#)

[Inserting new testcases](#)

[Extracting records from a database in a single query.](#)

[Creating a class for database operations](#)

[Introducing RowSet](#)

[Connected RowSetobjects](#)

[Disconnected RowSetobjects](#)

[CachedRowSet](#)

[Database extract in a method](#)

[Integration with the framework](#)

[Conclusion](#)

[Questions](#)

## [8. Get Introduced to TestNG](#)

[Structure](#)

[Objectives](#)

What is a testing framework?

What is TestNG

Configure TestNG

Understanding TestNG XML

What are the Test classes?

Learning TestNG annotations

Parameterization using testng.xml

Data Providers

TestNG listeners

ISuiteListenerinterface

ITestListenerinterface

Conclusion

Questions

## **9. Parallel Execution**

Structure

Objectives

What are TestNG groups?

TestNG XML for incorporating groups

Introduction to parallel execution

Changes to the database for parallel execution

Conclusion

Questions

## **10. Understanding Maven**

Structure

Objectives

Introducing Maven

Setting up Maven

Maven build life cycle

[Maven phases](#)

[Maven command line calls](#)

[Plugin goals](#)

[Setting up your project to use the build lifecycle](#)

[Packaging](#)

[Plugins](#)

[Maven Compiler plugin](#)

[Maven SureFireplugin](#)

[Using Suite XML files](#)

[Specifying test parameters](#)

[Using the DataProvider to pull in the included group](#)

[Assertions in TestNG](#)

[Hard assertion](#)

[Soft assertion](#)

[Class for assertions](#)

[Introducing extent reports](#)

[Advantages of using extent reports](#)

[Components of extent reports](#)

[Maven dependency for extent reports](#)

[Generating extent reports](#)

[Integration with GitHub](#)

[Conclusion](#)

[Questions](#)

## [11. Jenkins Introduction and Scheduling](#)

[Structure](#)

[Objectives](#)

[Introducing Jenkins](#)

[Setting-up Jenkins](#)

[Execute a Maven Build from the command line](#)

[Creating a Jenkins job](#)

[Configuring extent report in Jenkins](#)

[Scheduling a Jenkins job to build periodically](#)

[Conclusion](#)

[Questions](#)

## [12. Selenium Grid and Executing in the Cloud](#)

[Structure](#)

[Objectives](#)

[Introducing RemoteWebDriver](#)

[Learning about Selenium standalone server](#)

[Starting the server](#)

[Learning about the RemoteWebDriver client](#)

[Steps to convert a regular script to use RemoteWebDriverServer](#)

[Understanding the Selenium Grid](#)

[Looking at the hub](#)

[Knowing the node](#)

[Hub configuration parameters](#)

[WaitTimeout of a new session](#)

[Waiting for DesiredCapability match](#)

[Node configuration parameters](#)

[Changing the default number of browsers](#)

[Node timeouts](#)

[Browser timeouts](#)

[Registering a node automatically](#)

[Unregister an unavailable node](#)

[Specifying configuration using JSON files](#)

[Changes to the SingletonDriver class](#)

[Changes to the setDriver method](#)

[Changes to the TestNG XML](#)

[Class for default global variables](#)

[Fetching runtime parameters](#)

[Introducing BrowserStack](#)

[Setting-up BrowserStack](#)

[Conclusion](#)

[Questions](#)

## [13. Mobile Test Automation Using Appium](#)

[Structure](#)

[Objectives](#)

[Types of mobile applications](#)

[Introduction to Appium](#)

[Learning about Appium architecture](#)

[Understanding UIAutomator2](#)

[Understanding XCUITest](#)

[Setting up Appium](#)

[Using NodeJS to install Appium](#)

[Downloading Appium desktop client](#)

[Start the Appium Server](#)

[Connecting a real Android device on Windows](#)

[Creating a simple test](#)

[Changes for the framework](#)

[Changes to the SingletonDriver class](#)

[Conclusion](#)

[Questions](#)

## [14. A Look at Selenium-4](#)

[Structure](#)

[Objectives](#)

[Changes related to taking screenshots](#)

Taking a screenshot of a single web element or a section

Switching to the parent frame directly

Selenium Grid-4

Standalone mode

Hub and node

Fully distributed

New locators

Using artificial intelligence with Selenium

Visual validation

Exploring AppliTools Eyes

Self-correcting tests

Auto coders

Improved Docker support

Next steps

Conclusion

Questions

## CHAPTER 1

### *First Look at Selenium WebDriver and WebElements*

Welcome to the exciting world of Test Automation. In this book, we will learn about Selenium WebDriver. Chapter by chapter, you will get up to speed with Selenium WebDriver. We will start with the basics and then move on to complex concepts, eventually designing a Hybrid Framework which fetches keywords and data from the MySql database. The database can be any relational database.

## Structure

This chapter will predominantly focus on:

Test automation and why is it required

Different tools available

Difference between Selenium 2 and 3

Selenium WebDriver Architecture

WebDriver and WebElements

Basic operations on WebElements

## Objective

Understanding test automation

Learn different tools available for automation

Understand Selenium WebDriver Architecture

Know the difference between Selenium IDE and Selenium Server

Learn about WebElements and the methods available

Setting up a Maven project in Eclipse

Understanding JSONObject and JSONArray

We start by understanding what test automation is and the need for it.

Imagine you have a data creation task at hand for an insurance portal. You need 700 test policies created in three days. Assume one person takes 20 minutes to issue a policy, and in an hour, he would be able to issue 3 policies. Assuming 6 productive hours in a day, he would be able to issue 18 policies in a day, which sums to 54 policies in three days. We need to hire

additional resources if we have to accomplish this task. Moreover, the execution done manually may contain errors.

This is only one scenario. There is always the challenge of increasing test coverage. Here is where test automation becomes mandatory. Few of the advantages of test automation are:

Increased test coverage

No errors in test execution

Fast due to parallel execution

Unattended 24/7 execution

## Tools available for test automation

When it comes to automating a web application, an automation tool is required. There are a few names that have been leaders in the automation market, namely.

**Unified functional test:** Owned by MicroFocus. This tool works primarily on Windows.

**Silk test:** Owned by MicroFocus and used for Functional and Regression testing

**Selenium WebDriver:** An open-source tool used for functional and regression testing

**IBM rational functional test:** A data-driven testing platform for functional and regression testing developed by IBM.

Selenium, being open-sourced, proves to be the most popular option.

## What's new in Selenium 3.X

Looking back at Selenium 2.X, there have been quite a few changes in Selenium 3.X. Few of the prominent ones are listed below:

### APIs that confirm to W3C

Selenium is now a W3C standard. This being the case, now most Browser vendors support Selenium. Moreover, the APIs are more robust and match the implementation of object-oriented programming. This has made programmers happy.

### *Advanced functionalities supported*

Selenium now supports advanced functionalities like File IO and Mobile APIs. This has simplified testing complex applications.

### [\*Introduction of Appium project\*](#)

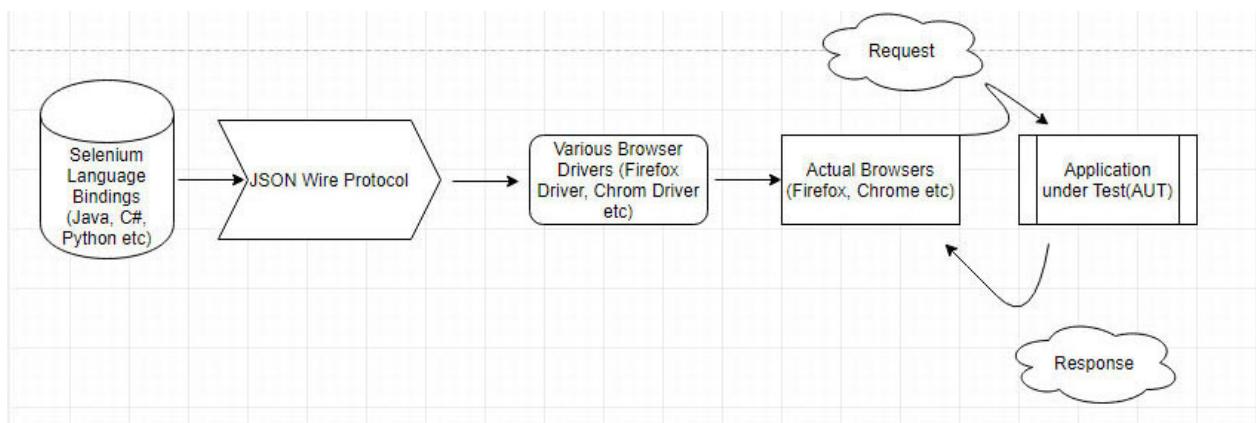
The mobile testing features which were included in Selenium 2 were moved to a separate project called Appium. Appium is an open-source framework for testing native, hybrid, and web applications on Android and IOS devices.

Appium replaces iPhoneDriver and AndroidDriver APIs that were part of Selenium 2.

Let's begin our journey of exploring Selenium WebDriver. We begin by studying the WebDriver architecture.

## Selenium WebDriver architecture

Shown below is the architecture of Selenium WebDriver.



**Figure 1.1**

In the diagram above, the selenium web driver scripts invoke the appropriate driver using the JSON wire protocol.

The driver converts the commands sent to it by WebDriver scripts into native commands, which are then passed to the appropriate browser.

Appropriate actions get performed on the browser by the native commands during which requests get sent to the application under test.

The application sends back a response that is eventually conveyed back to the Selenium WebDriver scripts.

## Selenium Server

Using Selenium Server, we can distribute out test cases to various nodes that are connected to a single hub. The hub is aware of the configuration of individual nodes. Whenever we configure our tests to connect to the hub, it selects the node with the requested configuration (for example, browser and version) and runs the test on that node. Thus it acts as a load balancer and distributes the tests. We can also run the tests parallelly on multiple machines using different combinations of OS, type of browser, and version.

## Selenium IDE

Using Selenium IDE, we can record, edit, playback, and debug tests which are in Selenese format. We can convert these tests to Selenium WebDriver format. The uses of Selenium IDE help in exploratory testing by creating quick tests through record and playback. Macros can also be created for automating repetitive tasks while testing.

## Main interfaces of Selenium WebDriver

**WebDriver** This is the main interface to use for testing and represents a web browser. The methods in this class fall into three categories, ie. Controlling the browser itself, Selection of WebElements, and aids for debugging. The frequently used key methods of this interface are: get used to load a new web page and findElement - for a single element as well as findElements - for a list of elements. The user needs to instantiate implementations of this interface directly. We will study findElement and findElements when we see WebElement next.

**WebElement:** Represents an HTML element on a web page. Generally, all operations for interacting with a page will be performed via this interface. All method calls will do a freshness check to ensure that the element reference is still valid. This determines whether or not the element is still attached to the DOM. If this test fails, then a StaleElementReferenceException is thrown, and all future calls to this particular instance fail.

## Property or attribute of WebElements

Property or attribute gives an identity to the WebElement.

Lets consider a simple example of the **Google Search** button. The HTML code snippet below shows the various properties or attributes for this button:

```
name="btnK" id="gbqfba" aria-label="Google Search" class="gbqfba">id="gbqfsa">Google Search
```

Here the various attributes of the button are name, id, aria-label and class.

## [Accessing WebElements](#)

Now that we know what WebElements are, we will see how to access webelements. Selenium WebDriver provides two methods for accessing WebElements: findElement and

### **FINDELEMENT**

findElement returns the first WebElement. It is invoked on the WebDriver instance since it is an instance method in the WebDriver interface. It accepts as an argument object of the By class. It is affected by the implicit wait time set in the script at the time of execution. Invoking the findElement method will return a matching row, or try again repeatedly until the configured timeout is reached.

The signature of the findElement method is:

```
WebElement findElement(By by);
```

### **FINDELEMENTS**

Finds all elements within the current page that match the criteria provided. This method is affected by the implicit wait times set in the script at the time of execution. When waiting, this method will

return as soon as there are more than 0 items in the collection, or an empty list is returned if the timeout is reached.

The signature of the findElements method is:

```
List findElements(By by);
```

If unsure whether the element would be found on the page, use findElements instead of

Let's understand By class.

#### **BY CLASS**

Abstract class, which provides a mechanism to locate elements within a document. To create your locating mechanisms, it is possible to subclass this class and override the protected methods as required, though it is expected that all subclasses rely on the basic finding mechanisms provided through static methods of this class.

Eight static methods of the By class:

id: Returns a By which locates elements based on the id attribute

linkText: Returns a By which locates elements by the exact text it displays

partialLinkText: returns a By which locates elements that contain the given link text

name: Returns a By which locates elements based on the name attribute

xpath: Returns a By which locates elements based on the xpath provided

className: Finds elements based on the value of the class attribute. If an element has many classes, then this will match each of them. For example, if the value is one two onone, then the following classNames will match: one and two

cssSelector: Finds elements via the driver's underlying W3 Selector engine. If the browser does not implement the Selector API, the best effort is made to emulate the API.

tagName: Returns a By, which locates elements by their tagname.

## Accessing the various attributes of WebElements

We can access the various attributes of a webelement using the `getAttribute` method, which is invoked on the `WebElement`. Let's take the example of the **Google Search** button above. To get the class attribute, we execute the code shown below:

```
WebElement googleButton = driver.findElement(By.Id("gbqfba"));
System.out.println("The class is:
"+googleButton.getAttribute("class"))
```

## Maven

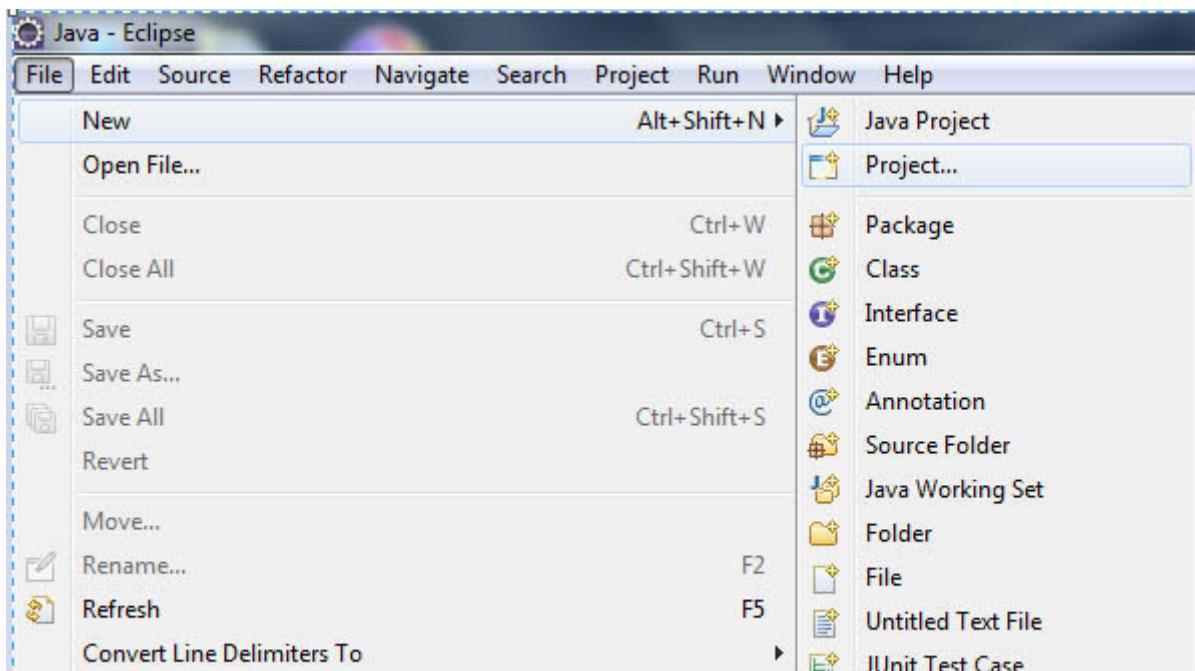
Maven is a build automation tool used mainly for Java projects. It describes how software is to be built and also its dependencies. Maven uses convention over configuration for the build process. An XML file called POM.xml describes the project being built, its dependencies on other modules and components, the build order, directories, and required plug-ins. It comes with pre-defined targets for executing well-defined tasks such as compiling code and packaging it for deployment.

Maven downloads Java libraries and Maven plug-ins dynamically from one or more repositories such as the Maven 2 Central Repository and stores them in a local cache or repository as one may call it.

## Creating a Maven project in Eclipse

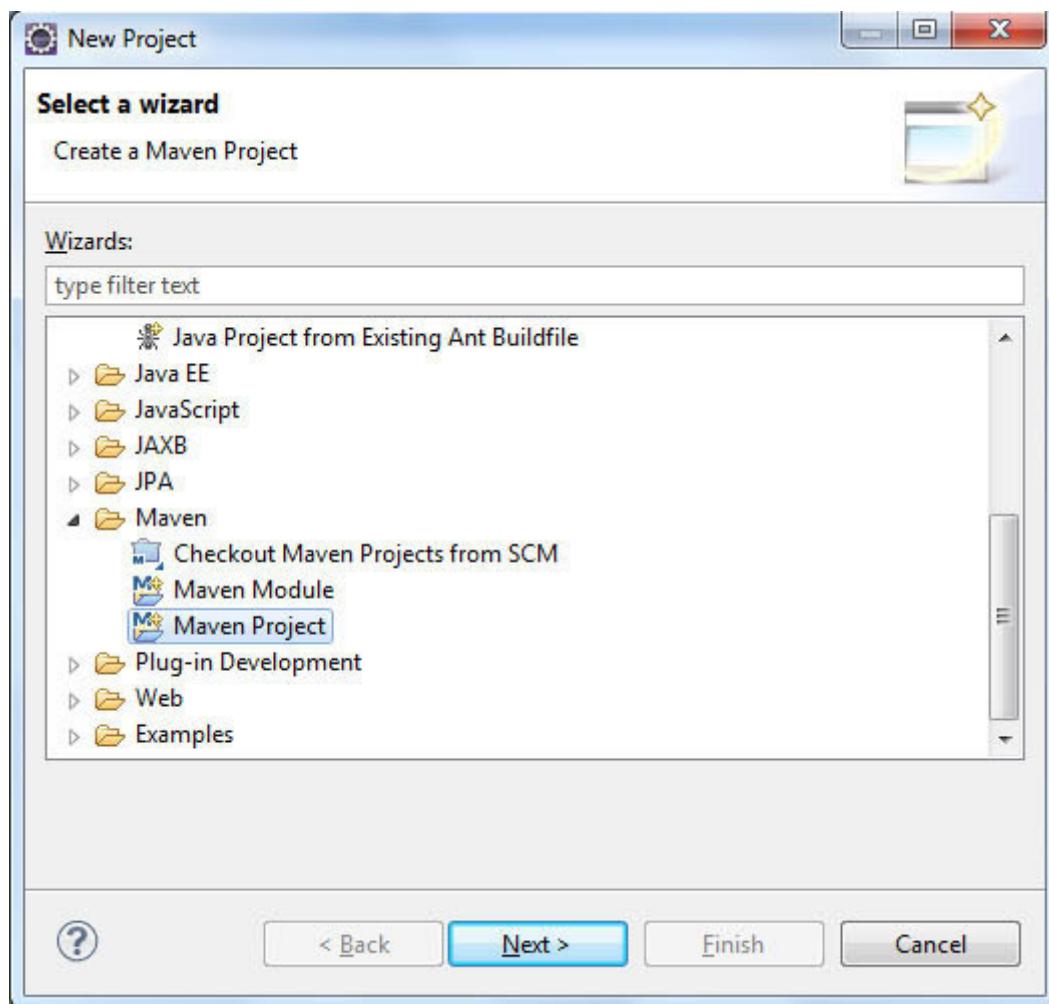
Now is the time to create a sample project using Maven. Follow the steps below to create a Maven project in Eclipse.

Click **File | New |**



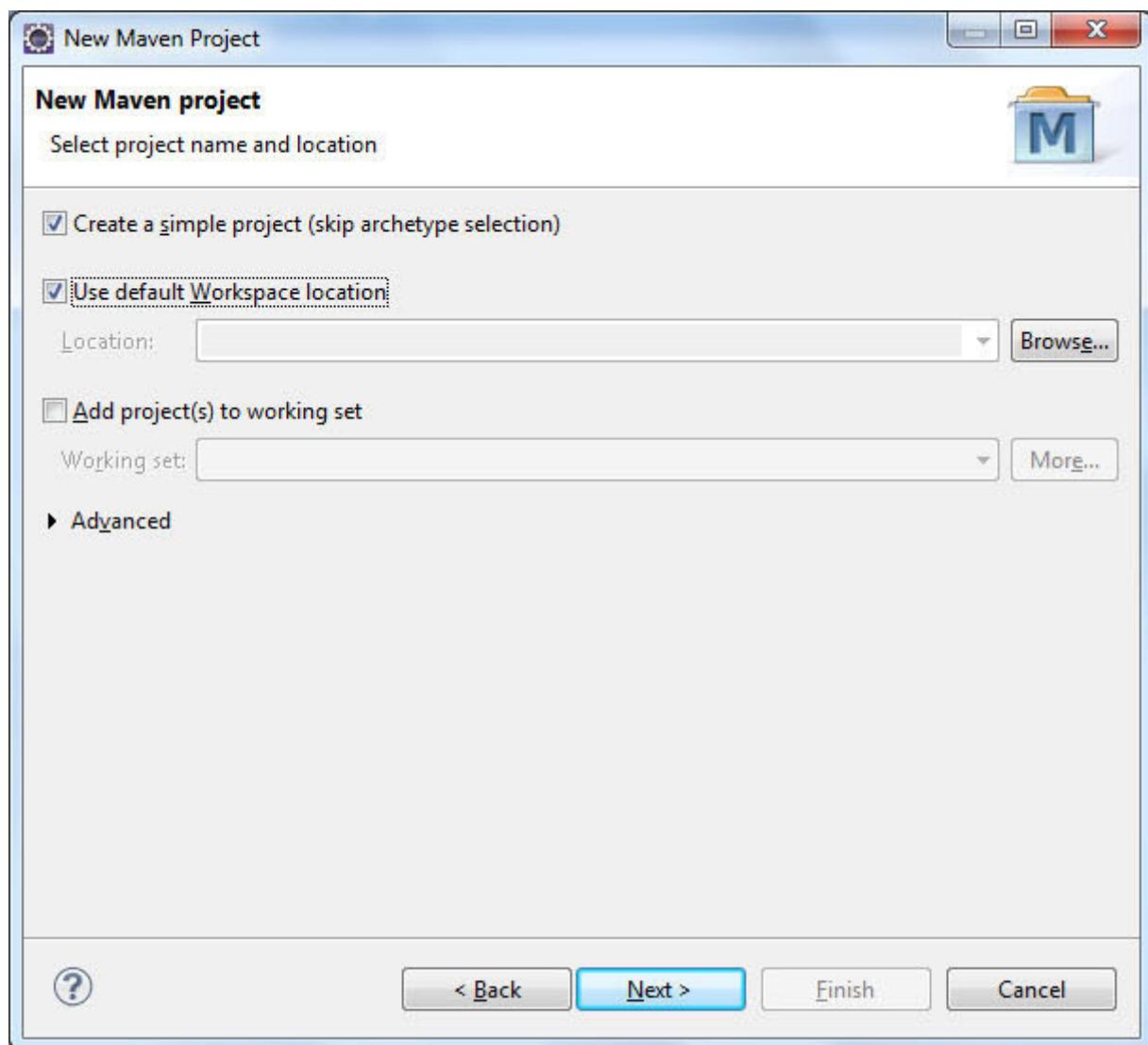
**Figure 1.2**

Select **Maven Project** as shown and click



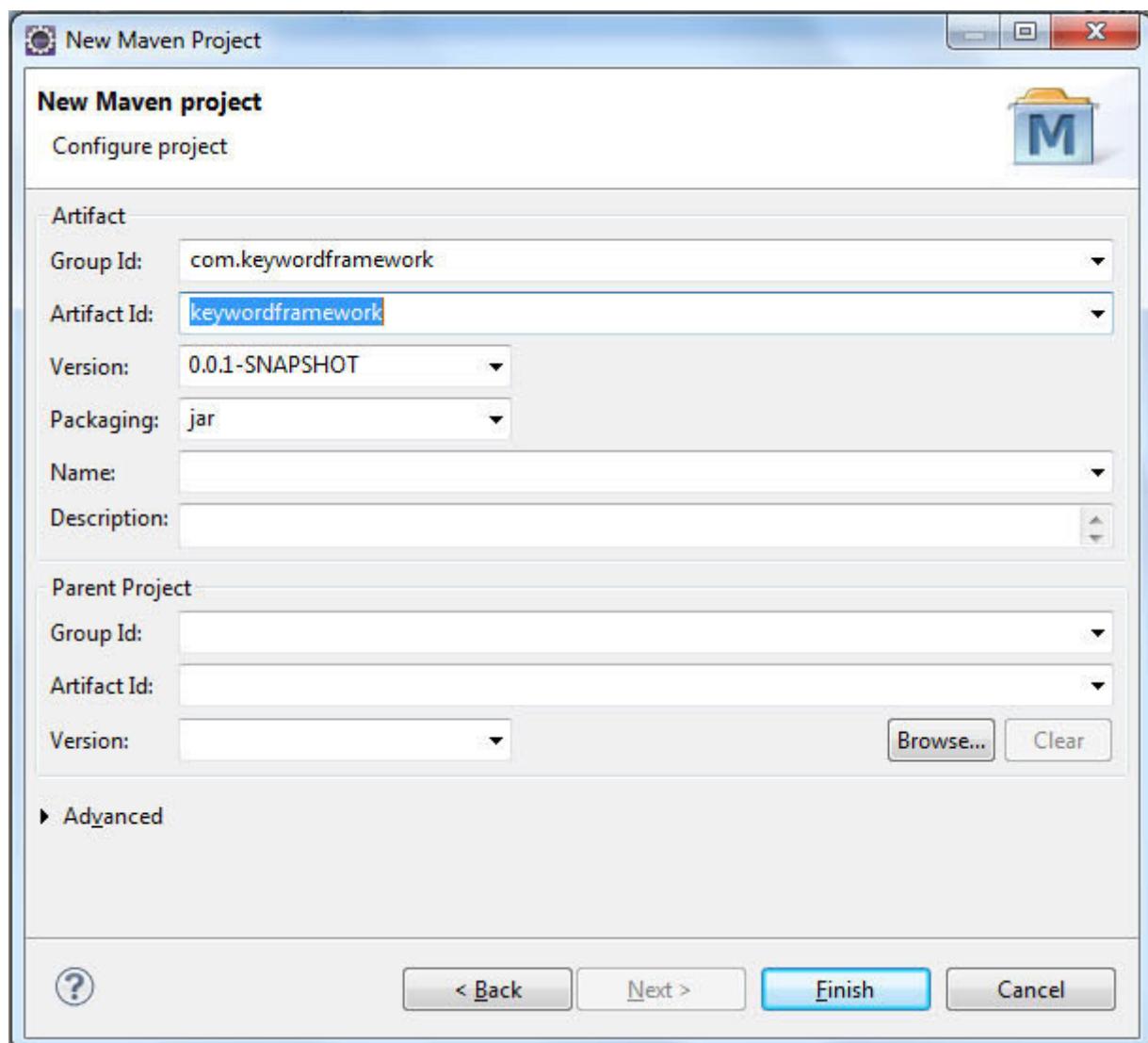
*Figure 1.3*

Click **Create a simple project (skip Archetype Selection)** on the next screen, as shown below. Click



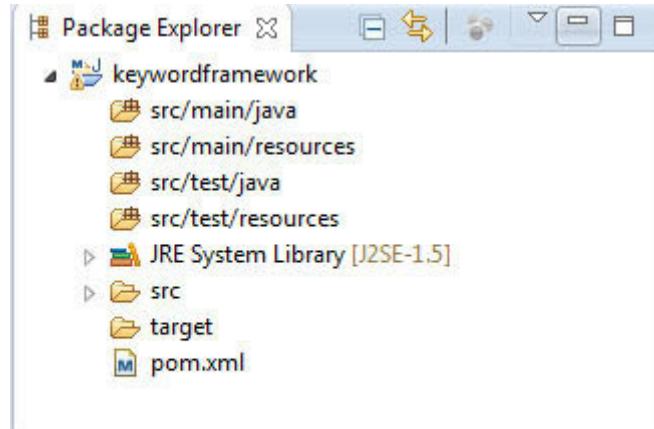
*Figure 1.4*

Input the **Group** Ideally, this is the package name of the project. The Artifact ID corresponds to the name of the JAR file, in case you wish to create one. Let the packaging be JAR. Note that the version is 0.0.1-SNAPSHOT. The SNAPSHOT at the end indicates that the project is under development and has not been released. Click



*Figure 1.5*

Eclipse takes some time to create the project and eventually creates a project structure, as shown below:



**Figure 1.6**

We will have all our source code in `src/main/java`. `src/main/resources` will contain any other files like property files that are required for our project. Similarly, `src/test/java` will contain test scripts, and `src/main/resources` will contain test files needed for testing. Apart from this, there is a Maven Dependencies folder, which is currently empty.

The heart of our project, though, is the `pom.xml` file. This will contain dependencies and build configurations for the project. This is what the `pom.xml` looks like:

```
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
http://maven.apache.org/xsd/maven-4.0.0.xsd">
4.0.0
com.keywordframework
```

keywordframework

0.0.1-SNAPSHOT

The groupId and artifactId that were added on the previous screens have appeared in the inside the project tag. In order to work with Selenium WebDriver, we will have to add Selenium dependencies within the project tag. We will be adding those from the Maven Repository.

Head over to Maven Repository and add the dependency shown below.

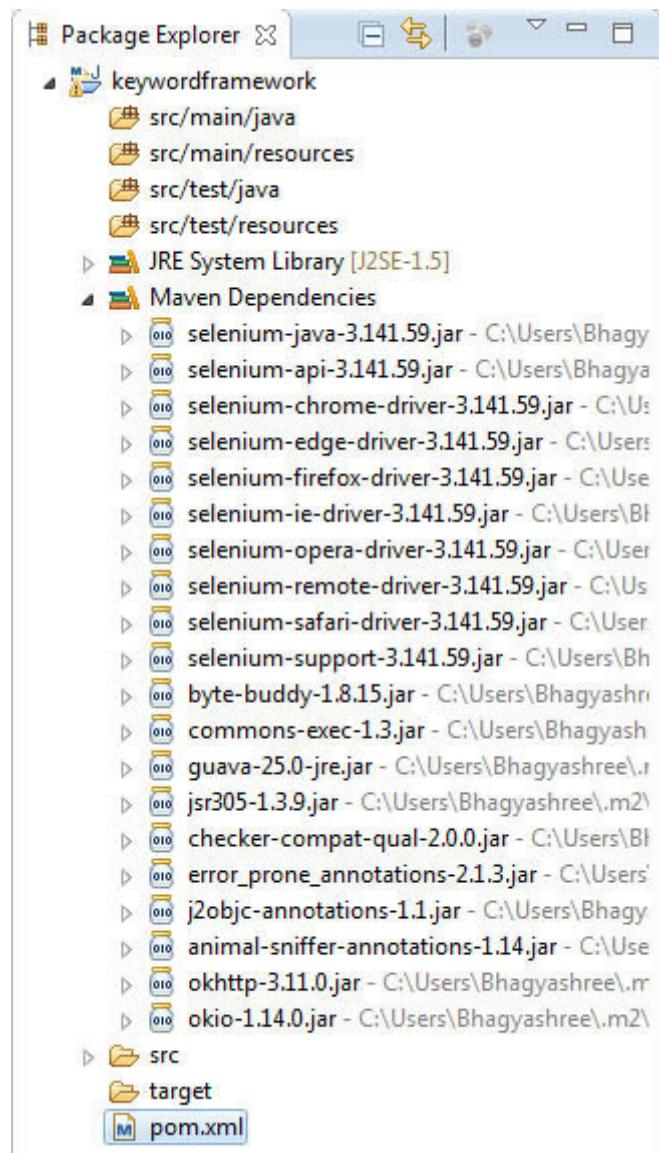
org.seleniumhq.selenium

selenium-java

3.141.59

Place this inside the tag in Since this tag does not exist currently, we will be adding it.

After adding, save the pom.xml file and notice the activity at the bottom right corner of Eclipse, which says **Building** A folder name **Maven Dependencies** will get created and will contain all the required JAR files for creating the project.



**Figure 1.7**

## [\*\*Manual configuration\*\*](#)

With this configuration, we are set to start coding. But wait, what if your company has a firewall that does not allow you to download files from the internet, For such situations, there is the option of manually adding the required JARs to the build path.

Follow the steps below to configure the **Build** path:

Create a Java project in Eclipse

Right-click on the project in **Project Explorer**

Select **Build Path | Configure Build Path**

Click on **Add External JARs** and add the required JARs

Next, let's move on to what a test automation framework is.

## *Introduction to the Test Automation Framework*

If there are a bunch of scripts and they each execute a particular functionality, those scripts are not of much use unless they are put into an organized structure. By doing this, the scripts can be connected if need be or run individually to produce an excel or graphical reports. Moreover, at the same time, logs can be generated to indicate what the status of test execution is. A folder structure can be created so that reusable methods can be kept in one folder, and the reports can be in another folder, a separate folder can contain Keyword classes, etc. The entire framework can be executed using the POM.xml file. The POM file can be invoked from Jenkins. We will have a detailed look at this when we slowly proceed towards the creation of a framework.

## Keyword -driven framework

The keyword-driven framework is a type of functional automation framework. The keyword-driven framework has four main components test case id, object xpath, page name, object the action to be performed on that object, and the data for that object.

The above setup can be achieved with the help of an Excel spreadsheet, CSV file, or database tables. Let's understand the different components of this framework.

**Test Config table:** This is a table with the primary key of The second field in this table will be an Execute flag, which has a value of Y or N and indicates whether the test case should be executed.

**Test Cases table:** The test cases table will be having the columns TestStepId, and

**Object Repository table:** The object repository table will have ObjectRepositoryID, Page Id, Object and the XPath of the object.

**Test Data table:** Data for each test case will be stored here. The table will have fields like Test Case ID and Test Step ID, where Test Case ID will be the foreign keys linking to the Test Cases Apart from this, it will have a field called Data.

The object repository table and the Test Data table act as lookup tables from where we can fetch data

The concept behind the keyword-driven approach is to separate the code from the actual input, which in our case are the TestConfig, and the individual page wise data tables. This helps a non-technical person to create test cases with data. With this, a manual tester with no experience in coding can write automation scripts. This also indicates that a technical person is needed to set up the framework and to work on changes and updates to the background automation code.

Let us try to understand the flow with the help of the [www.freecrm.com](http://www.freecrm.com) application. Imagine that the following flow has to be automated where one needs to do the following:

Open a browser

Navigate to URL

Click on **Sign In** button

Enter username

Enter password

Click on **Log In** button

Click on **LogOut** button

Close the browser

Below is the list of common components that you would require to achieve the task:

**MySQL or any other relational database:** This is the database which keeps most of the data for keyword driven framework, and this book will have tables for TestConfig, object repository and table for each page

**Driver script:** This is the heart of our framework that will read the database config property from a java interface class, then extract the test cases to execute from TestConfig table, extract the corresponding teststeps. Based on the Object Repository ID, extract the locator from the Object Repository table and, using the TestCaseID, TestStepID, and extract the corresponding data from the TestData table.

**Classes for each object:** Once we get the xpath and data for that locator from the driver script, the corresponding class is invoked based on the testcaseid and xpath. The xpath will have the format The idtype will be textbox, dropdown, button, etc. and the idname will be a unique xpath for the locator. Taking the it will execute the class with the same name pushing data fetched from the TestData table.

## **ADVANTAGES**

**Less technical expertise:** Once Framework is set up, it is very easy for manual testers or non-technical testers to write scripts.

**Easily understandable:** The scripts are easily understood since they are in the form of keywords and data, which can be easily understood by testers.

**Independent of application:** You can start building Keyword Driven test cases before the application is built, as Object Repository can be easily set up using simple keywords that mimic the manual actions

**Script reusability:** Since we utilize the same keywords and sometimes data, the same script can be reused at multiple places

Next, we introduce **JavaScript Object Notation(JSON)**. We will be using JSON objects to store database query results.

## Introduction to JSON

JSON is the data interchange format. A simple JSON object looks like the one shown below.

```
{  
  "Name": "Lion",  
  "Category": "Wild",  
  "EatsMeat": true  
}
```

Assign this to a variable called Animal. Animal.Name will then contain

The structure above is called a JSON object. Several such objects can be packed together in what is called a JSON array.

Shown below is a JSON array consisting of three elements.

```
{  
  "Animals":  
    [ "Name": "Lion",  
      "Category": "Wild",  
      "EatsMeat": true],
```

```
  [ "Name": "Cat",
```

```
    "Category": "Domestic",  
    "EatsMeat": true],
```

```
[{"Name": "Horse",  
 "Category": "Domestic",  
 "EatsMeat": false},  
 ]
```

The code to extract the Category of Cat is  
Animals.get(1).getString("Category");

In Java, we have classes by the names JSONObject and JSONArray which will help us when we start coding.

In order to use JSONObject and we need to add the dependency shown below to

```
org.json  
json  
20180813
```

Take the latest version from Save the

Let's have a look at how our TestCases, TestData and OR tables will look like in JSON format:

```
{  
  "TestCases":
```

```
[
  {
    "TestCaseID": "TCoo1",
    "TestSteps": [
      {"TestStepID": "TSoo1", "ORID": "ORo1", "Action": ":"EnterText"}, {"TestStepID": "TSoo2", "ORID": "ORo2", "Action": ":"EnterText"}, {"TestStepID": "TSoo3", "ORID": "ORo3", "Action": ":"EnterText"}, {"TestStepID": "TSoo4", "ORID": "ORo4", "Action": ":"Click"}],
  {
    "TestCaseID": "TCoo2",
    "TestSteps": [
      {"TestStepID": "TSoo1", "ORID": "ORo1", "Action": ":"EnterText"}, {"TestStepID": "TSoo2", "ORID": "ORo2", "Action": ":"EnterText"}, {"TestStepID": "TSoo3", "ORID": "ORo3", "Action": ":"EnterText"}, {"TestStepID": "TSoo4", "ORID": "ORo4", "Action": ":"Click"}]
  }
]
```

We have two testcases above, where TestSteps is a JSONArray.

Next, we look at the TestData JSON:

```
{
  "TestData": [
    {
      "TestCaseID": "TCoo1",
      "TestData": [
        {"TestStepID": "TSoo1", "TestDataID": "TDoo1", "Data": "AUTODM1AUG1"},
        {"TestStepID": "TSoo2", "TestDataID": "TDoo2", "Data": "AUTOMAKER"},
        {"TestStepID": "TSoo3", "TestDataID": "TDoo3", "Data": "Welcomeo1"}
      ]
    }
  ]
}
```

The final JSON that we would construct is the Object Repository JSON

```
{
  "OR": [
    {
      "PageID": "Login",
      "Objects": [
        {"ObjectID": "Domain", "Xpath": "//*[@id='ABC']"},  

        {"ObjectID": "UserName", "Xpath": "//*[@id='DEF']"},  

        {"ObjectID": "Password", "Xpath": "//*[@id='PQR']"},  

        {"ObjectID": "Submit", "Xpath": "//*[@id='SUB']"}
      ]
    }
  ]
}
```

## Database table structures

### TestConfig Table

Field	Type	Null	Key
TestCaseID	varchar(10)	NO	PRI
Description	varchar(20)	YES	
Execute	varchar(1)	YES	

*Figure 1.8*

### TestCases Table

Field	Type	Null	Key
TestCaseID	varchar(10)	NO	MUL
TestStepID	varchar(10)	NO	PRI
ActionKey	varchar(10)	NO	
ORID	varchar(10)	NO	

*Figure 1.9*

### TestData Table

Field	Type	Null	Key
TestCaseID	varchar(10)	NO	MUL
TestStepID	varchar(10)	NO	MUL
TestDataID	varchar(10)	NO	PRI
DataKey	varchar(10)	NO	

***Figure 1.10***

## **Object Repository Table**

Field	Type	Null	Key
ORID	varchar(10)	NO	PRI
PageID	varchar(10)	YES	
ObjectID	varchar(10)	YES	
XPath	varchar(100)	YES	

***Figure 1.11***

### Basic insert operation on JSONArray

Now that we have seen the table structures, lets understand how a JSONObject and JSONArray works in Java. For understanding this, we will construct the Object Repository JSON Object which contains one element which is a JSON Array

```
package keywordframework;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.ResultSetMetaData;
import java.sql.SQLException;
import java.sql.Statement;

import org.json.JSONArray;
import org.json.JSONObject;

public class TestExtract {

    private static String pageID;
    private static String objectID;
    private static String xPath;

    public static void main(String[] args) {
```

```
ResultSet resultSet = null;
ResultSetMetaData rsmd = null;
JSONArray test = new JSONArray();
JSONArray OR = new JSONArray();

JSONObject j = null;
JSONObject Object_Repository = null;
JSONObject Full_Object_Repository = null;
/******Create Connection, Statement and Resultset objects and
execute SQL Query and fetch metadata*****/
try {

Connection conn = DriverManager.getConnection(
“jdbc:mysql://localhost:3306/selenium_framework”, “root”,
“Pc9121975!”);
Statement stmt = conn.createStatement();
resultSet = stmt
.executeQuery(“SELECT B.PageID as PageID,B.ObjectID as
ObjectID,B.XPath as XPath FROM
selenium_framework.object_repositoryB,selenium_framework.testcases
A where A.ORID=B.ORID;”);
rsmd = resultSet.getMetaData();

resultSet.next();
} catch (SQLException e) {
// TODO Auto-generated catch block
e.printStackTrace();
}
/***Iterate over Resultset and create our JSON Object***/
try {
```

```
while (!resultSet.isAfterLast()) {  
    pageID = resultSet.getString("PageID");  
    objectID = resultSet.getString("ObjectID");  
    xPath = resultSet.getString("XPath");  
    j = new JSONObject();  
    for (int j1 = 1; j1 <= rsmd.getColumnCount(); j1++) {  
        System.out.println("Column: " + rsmd.getColumnName(j1));  
        System.out.println("Value: "  
            + resultSet.getString(rsmd.getColumnName(j1)));  
        if (!rsmd.getColumnName(j1).equalsIgnoreCase("pageid")) {  
            j.put(rsmd.getColumnName(j1),  
                resultSet.getString(rsmd.getColumnName(j1)));  
  
        }  
    }  
    test.put(j);  
    resultSet.next();  
}  
} catch (SQLException e) {  
    e.printStackTrace();  
}  
Object_Repository = new JSONObject();  
Object_Repository.put("pageID", pageID);  
Object_Repository.put("Objects", test);  
OR = new JSONArray();  
OR.put(Object_Repository);  
System.out.println(OR);  
Full_Object_Repository = new JSONObject();  
Full_Object_Repository.put("OR", OR);  
System.out.println(Full_Object_Repository);  
}
```

}

This is what Full\_Object\_Repository looks like when formatted:

```
{  
  "OR": [  
    {  
      "Objects": [  
        {  
          "XPath": "//*[@id='ABC']",  
          "ObjectID": "Domain"  
        },  
        {  
  
          "XPath": "//*[@id='DEF']",  
          "ObjectID": "UserName"  
        },  
        {  
          "XPath": "//*[@id='GHI']",  
          "ObjectID": "Password"  
        },  
        {  
          "XPath": "//*[@id='JKL']",  
          "ObjectID": "Submit"  
        }  
      ],  
      "pageID": "Login"  
    }  
  ]  
}
```

This extract is just an example to show how to create a JSON object from a resultset. By doing this, we are minimizing the hits to the database. Right now, this data is only for one page, which is login. When we start creating keywords and data for more pages, we will have to build a logic where if the page changes, then we create a new JSON array that contains a new object with a new pageID.

With this, we complete the introductory chapter on Selenium WebDriver. In the next chapter, we get introduced to the concept of instantiating a WebDriver object and the various WebDrivers available.

## Conclusion

This was the introductory chapter where we got introduced to what Test Automation is. We saw what Selenium WebDriver is and understood the architecture of Webdriver with a diagram. We went over the Selenium Server and IDE. We had a short introduction to the main interfaces of WebDriver and saw how we could access WebElements using the methods findElement() and A synopsis of the By class followed with an introduction about how to access the various attributes of We saw how to create a Maven project and configure it manually in case there is a firewall blocking internet downloads. We looked at a brief introduction to the test automation framework and understood what a keyword-driven framework is. We got introduced to JSONObject and JSONArray and saw a small code for extracting values from resultset and creating a JSON object from it.

In the next chapter, we understand the different drivers available and how to instantiate the different drivers. We will take a look at HeadLess browsers and executing the Selenium WebDriver script from the command prompt.

## Questions

Explain the difference between Selenium WebDriver, Server and Selenium IDE

Write a small code snippet to capture value from the text box

Write a small code snippet to capture the state of the text box

Explain Group ID and artifact ID in the Maven setup.

Write a program to extract test data in the form of JSON array

## CHAPTER 2

### *Looking at the Various WebDrivers*

The previous chapter gave us an introduction to what Selenium WebDriver is. Now it's time to understand how to use Selenium WebDriver to instantiate browser sessions. We will understand several concepts related to instantiating drivers along with the complete details of how to instantiate each of the drivers and the concepts linked to them.

## Structure

Here's what we will learn in this chapter:

What is a driver instance

Instantiating Firefox browser sessions

Headless Firefox

Instantiating Chrome sessions

Headless Chrome

Instantiating Internet Explorer sessions

Instantiating Microsoft Edge sessions

Instantiating Safari sessions

Introduction to WebDriverManager class

Setup of WebDriverManager

## Objectives

This chapter will act as an in-depth guide to driver instantiation in Selenium.

Understand concepts such as profiles for Firefox and Chrome and preferences for Firefox.

Understand what WebDriverManager is and how it can ease the creation of browser sessions without explicitly downloading the driver executable.

We begin by understanding what an instance means.

### **WHAT IS AN INSTANCE IN JAVA**

Java is object-oriented, and everything in Java is based on objects. Consider a class called Animal which has properties or attributes like legs, tail, ears etc. An Animal also displays certain behaviour like barking or meowing. An Animal class can be displayed as given under.

```
class Animal {  
    private Int legs;  
    private String tail;  
    private String ears;
```

```
public String createSound() {  
    return null;  
}  
}
```

An animal can be a cat, dog or horse. So how do you create cats, dogs or horses? Given below is the code to do so:

```
Animal cat = new Animal();  
Animal horse = new Animal();
```

Here cat and horse are called instances of the Animal class.

## **WHAT IS AN INTERFACE IN JAVA**

An interface can be considered as a blueprint of a class. It has static constants and abstract methods. If we want to define the Animal interface.

```
interface Animal {  
    int legs=0;  
    String tail=null;  
    String ears=null;  
    public String createSound();  
}
```

All variables in an interface are public static final by default, and all methods are public and abstract. Hence it is not required to explicitly define the variable as public static final and the methods as public and abstract. The way the compiler sees the above interface is:

```
interface Animal {  
    public static final int legs=0;  
    public static final String tail=null;  
    public static final String ears=null;  
    public abstract String createSound();  
}
```

To use this interface, a class has to be created that implements this interface as shown below:

```
public class Horse implements Animal{  
    @Override  
    public String createSound() {  
        return null;  
    }  
}
```

It is mandatory to provide an implementation of all unimplemented methods of the interface. Here we have one unimplemented method createSound for which we provided a sample implementation. The implementation can vary.

## What is a driver instance?

For performing automation, a browser session has to be opened. To open the browser session, we need to create something called driver instance. A driver instance is just a browser session which can be created in Selenium using the code below:

```
System.setProperty("webdriver.chrome.driver", "path to chromedriver executable")
WebDriver driver = new ChromeDriver();
```

System.setProperty indicates which driver needs to be used to open a particular browser along with the path to the executable. This executable file can be downloaded from

<http://www.seleniumhq.org> and stored at a convenient location. webdriver.chrome.driver is the key and the actual path is the value.

In the above code snippet, we saw how to instantiate a chrome driver instance. Let's have a look at how to instantiate the other drivers.

## [Firefox Driver](#)

With Selenium 3.X, implementation of the Firefox driver has changed. From Firefox version 47+, we need to use a separate driver that handles the Firefox browser. This driver, named Geckodriver, provides the HTTP API to communicate with the Firefox browser. To use the Geckodriver, we need to download it from

In the code, simply insert the two lines shown below:

```
System.setProperty("webdriver.gecko.driver",
"./src/test/resources/drivers/geckodriver.exe");
driver = new FirefoxDriver();
```

This will open a new Firefox browser. Notice that here we are instantiating the FirefoxDriver class which is an implementation of the webdriver interface.

## What is a Firefox Profile?

A Firefox profile is a folder that Firefox browser utilizes to store information like passwords, bookmarks, data specific to user and settings. Any number of profiles can be created with different settings, and the browser session can be customized accordingly.

As per Mozilla, the attributes shown below can be stored in profiles:

Passwords

Cookies

Download history

Autocomplete history

Download actions

Search engines

DOM storage

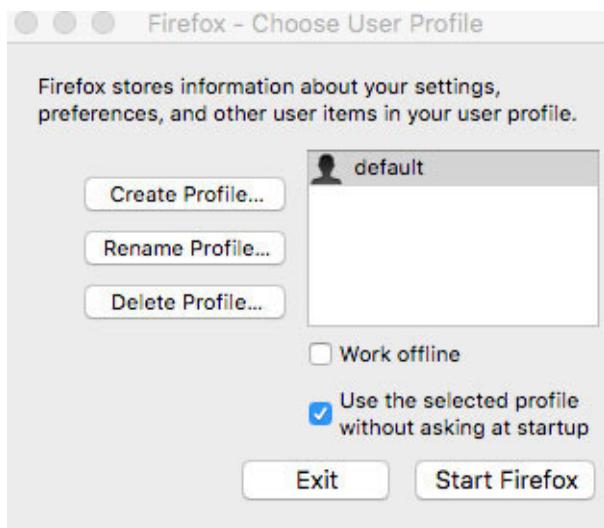
These are only some of the attributes. The complete list of attributes can be found on the Mozilla website.

One can create a Firefox profile by going to the run menu and type. The path shown below is the path to the Firefox executable;

"Path to firefox executable\firefox.exe" -P

**Please note: Before executing this command, all open instances of Firefox should be closed.**

This command will open the Profile Manager, which looks like the one shown below:



**Figure 2.1:** Choose User Profile dialog

The **Create Profile** button creates a new profile, **Rename Profile** renames an existing profile and **Delete Profile** deletes an existing profile.

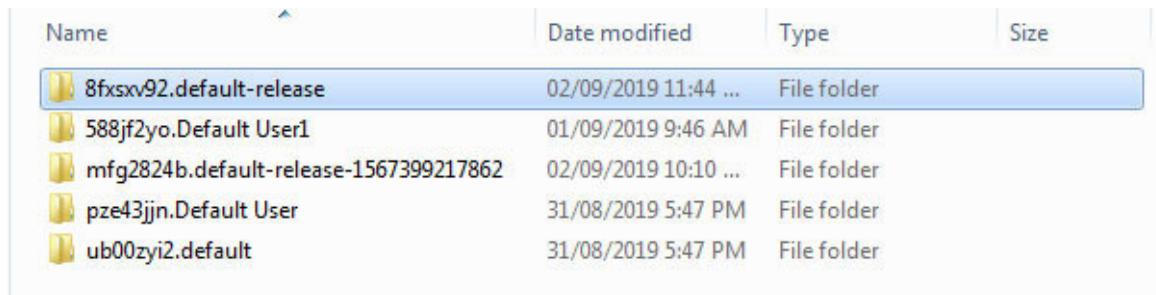
When we create an instance of FirefoxDriver, a temporary profile is created which the WebDriver uses. In order to see the profile that is currently being used by a Firefox instance, open a Firefox session if

not already open and type about:support. The screen shown below gets displayed:



**Figure 2.2:** Profile path

If you open this location and go one level up, you will see the profiles similar to the ones shown below:



**Figure 2.3:** Location of Firefox profile

These folders correspond to each Firefox instance launched by Firefox driver.

Next, we see how to create a custom profile. The code shown below creates a custom profile:

```
FirefoxProfile ffprofile = new FirefoxProfile();
FirefoxOptions ffOptions = new FirefoxOptions();
ffOptions.setProfile(ffprofile);
FirefoxDriver driver = new FirefoxDriver(ffOptions);
```

As shown above, an object of the FirefoxProfile class has been created for the Firefox Browser. Various options and preferences can be set using this object. There are two versions of this class. The first is the no-argument version shown above, which creates an empty profile which we can modify as needed. The other one uses one-argument version, and this argument is the profile directory which is the location of our desired profile.

## [Adding an extension](#)

We will add the Firebug extension to our profile so that whenever Firefox is opened using the appropriate profile, it opens up with the Firebug extension added. The code to make this happen is given below:

```
System.setProperty("webdriver.gecko.driver",
 "./src/test/resources/drivers/geckodriver.exe");
FirefoxProfile ffprofile = new FirefoxProfile();
ffprofile.addExtension(
 new File("./src/test/resources/extensions>xpath_finder.xpi"));

FirefoxOptions ffOptions = new FirefoxOptions();

ffOptions.setProfile(profile);
FirefoxDriver driver = new FirefoxDriver(ffOptions);
```

Let's look at how to store and retrieve profiles.

## [Storing and retrieving Profiles](#)

We can also store a Firefox profile information to a JSON file and later retrieve it. The class FirefoxProfile has a static method called `toJSON`. The return type is a string that has the JSON information.

Once we have the profile information in a string format, we can build a browser with the same profile using the static method `fromJSON` which takes a string argument. Let's see this in action in a small code snippet. Here we created a Firefox profile and added the firebug extension to the profile. We saved the profile in a JSON string format and later retrieved this profile for use.

```
FirefoxProfile ffProf = new FirefoxProfile();
ffProf.addExtension(
new File("./src/test/resources/extensions>xpath_finder.xpi"));
String jsonString = ffProf.toJson();
FirefoxOptions ffOpt = new FirefoxOptions();
ffOpt.setProfile(FirefoxProfile.fromJson(jsonString));
```

Now that we know what Firefox profiles are stored and retrieved, we will understand Firefox preferences next.

## [Understanding Firefox preferences](#)

A Firefox preference is a value that is defined or set by the user. The prefs.js file that is present in the Firefox profile is the file that contains user preferences (these preferences cannot be modified by the user). Shown below is the extract of this file:

```
user_pref("app.normandy.first_run", false);
user_pref("app.normandy.startupRolloutPrefs.network.cookie.cookieBehavior", 4);
user_pref("app.normandy.user_id", "2b82d10b-63cb-420c-8e23-6069cdc499b6");
user_pref("app.update.auto.migrated", true);
user_pref("app.update.download.attempts", 0);
user_pref("app.update.lastUpdateTime.addon-background-update-timer", 1568431440);
user_pref("app.update.lastUpdateTime.background-update-timer", 1568431200);
user_pref("app.update.lastUpdateTime.blocklist-background-update-timer", 1568431560);
user_pref("app.update.lastUpdateTime.browser-cleanup-thumbnails", 1568430840);
user_pref("app.update.lastUpdateTime.recipe-client-addon-run", 1568431080);
user_pref("app.update.lastUpdateTime.search-engine-update-timer", 1568430960);
user_pref("app.update.lastUpdateTime.services-settings-poll-changes",
```

If in case, we would like to modify these preferences, then another file called user.js should be created if not already present and the required preferences can be overridden in this file. Not all preferences can be modified (and hence named frozen preferences). Whenever a new preference is added, this preference gets added to the user.js file.

## Setting preferences

Next, we see how we can set our own preferences. We have a method in WebDriver for which the method signature is shown below:

```
public void setPreference(java.lang.String key, String value)
```

There are two overloaded versions of this method shown below:

```
public void setPreference(java.lang.String key, int value)  
public void setPreference(java.lang.String key, boolean value)
```

The key represents the preference that needs to be set, and the value represents the value that needs to be stored in it. Let us see a small example of setting the home page to [www.google.com](https://www.google.com) by using this method:

```
FirefoxProfile prof= new FirefoxProfile();  
prof.setPreference("browser.startup.homepage","https://www.google.co  
m");  
WebDriver driver = new FirefoxDriver(prof);
```

If the user tries to modify any of the frozen preferences, then an illegalArgumentException gets thrown, stating that the preference cannot be overridden.

## Firefox in headless mode

Headless mode is a very efficient way to run Firefox using Selenium WebDriver. It is extremely useful when we need to run the tests on Linux machines or to use Jenkins.

The FirefoxOptions class has to be configured in order to run Firefox in headless mode as shown below

```
@BeforeMethod  
public void setup() {
```

```
System.setProperty("webdriver.gecko.driver",  
"./src/test/resources/geckodriver.exe");
```

```
FirefoxOptions opt = new FirefoxOptions();  
opt.setHeadless(true);
```

```
driver = new FirefoxDriver(opt);
```

```
driver.get("http://www.google.com/");  
}
```

In the preceding code, we first created an instance of the FirefoxOptions class and then called the setHeadless() method, passing the value as true to launch the Firefox browser in

headless mode. You will see a message indicating the browser has been launched in headless mode, as shown in the following console output:

```
1532194389309 geckodriver INFO geckodriver 0.21.0
1532194389317 geckodriver INFO Listening on 127.0.0.1:21734
1532194390907 mozrunner::runner INFO Running command:
"/Applications/Firefox.app/Contents/MacOS/firefox-bin" "-marionette"
"-headless" "-foreground" "-no-remote" "-profile"
"/var/folders/zr/rdwhsjk54k5bj7yr34rfftrh0000gn/T/rust_mozprofile.DmJC
QRKVVRa6"
*** You are running in headless mode.
```

**Figure 2.4:** Headless execution

Now that we have seen how to execute Firefox in headless mode, we now understand how to instantiate a Chrome driver.

## Chrome driver

Chrome sessions can be invoked using the same procedure as Firefox, where one needs to set the path for the driver executable, as shown below:

```
System.setProperty(  
    "webdriver.chrome.driver",  
    "./src/test/resources/chromedriver.exe");  
WebDriver driver = new ChromeDriver();  
driver.get("http://www.google.com");
```

The code shown above opens a ChromeDriver session and navigates to www.google.com. Let us now understand the concept of Chrome options with the help of a small example.

## Chrome options

The Chrome options can be set using the ChromeOptions class as shown below. This is similar to a Firefox profile. In this example, we are disabling the remember password pop up that appears when you login to any website

```
ChromeOptions opt = new ChromeOptions();
MapObject> prefs = new HashMapObject>();
prefs.put("credentials_enable_service", false);
prefs.put("password_manager_enabled", false);
opt.setExperimentalOption("prefs", prefs);
WebDriver driver = new ChromeDriver(opt);
```

As can be seen above, we have used the ChromeOptions object and invoked the method setExperimentalOption to set two options credentials\_enable\_service and When the code snippet above is included, the remember password dialog no longer appears.

## Chrome extensions

Extensions can be added in Chrome that are similar to Firefox. There are two types of extensions that can be added, packed (in the form of crx file) and unpacked (in the form of a folder) using the ChromeOptions class.

The addExtensions method is utilized to add a packed extension as shown below:

```
ChromeOptions opt = new ChromeOptions();
opt.addExtensions(new File("src/main/extensions/XPath-Helper-
Chrome Web Store_v2.0.2.crx"));
ChromeDriver driver = new ChromeDriver(chromeOptions);
```

One can download a crx file by going to the page corresponding to the extension on Play Store, Google, copy the URL and download it from chrome-extension-downloader.com.

The code shown above will launch the Chrome browser with the XPath helper extension attached.

To add an unpacked extension, use the addArguments method as under

```
ChromeOptions opt = new ChromeOptions();
```

```
opt.addArguments("load-extension=/path/to/extension");
ChromeDriver driver = new ChromeDriver(opt);
```

Now that we have seen chrome extensions addition, let's look at dealing with Chrome in headless mode.

## Chrome in headless mode

Similar to Firefox, we can run tests in headless mode with This makes Chrome tests run faster and more efficiently, especially when working in a continuous integration environment. We can run Selenium tests in headless mode by setting the ChromeOptions class, as shown in the following code:

```
@BeforeMethod  
public void setup() {  
    System.setProperty("webdriver.chrome.driver",  
        "./src/test/resources/chromedriver");  
    ChromeOptions opt = new ChromeOptions();  
    opt.setHeadless(true);  
  
    driver = new ChromeDriver(opt);  
    driver.get("http://www.google.com/");  
}
```

We first created an instance of the ChromeOptions class and called the setHeadless() method on that instance that passes the value as true to launch the Chrome browser in headless mode. During the execution, no Chrome window is seen on the screen, and the test will be executed in headless mode.

## [Internet Explorer Driver](#)

Working with Internet Explorer is pretty much the same as with Chrome. Download the IEDriverServer.exe file from the Selenium HQ website and place in an appropriate folder.

Use the code shown below to open an IE browser and load a webpage in it.

```
System.setProperty("webdriver.ie.driver",
".src/test/resources/drivers/IEDriverServer.exe");
```

```
driver = new InternetExplorerDriver();
driver.get("http://www.google.com/");
```

Let's take a look at IEDriver capabilities next.

## IEDriver capabilities

Similar to options in Chrome and Firefox, we have capabilities for Internet Explorer. The code below ignores the security domains by using

INTRODUCE\_FLAKINESS\_BY\_IGNORING\_SECURITY\_DOMAINS capability. This is helpful to handle the protected mode settings of Internet Explorer.

```
DesiredCapabilities cap = DesiredCapabilities.internetExplorer();
cap.setCapability(InternetExplorerDriver.INTRODUCE_FLAKINESS_BY_IGNORING_SECURITY_DOMAINS,true);
```

There are other capabilities few of which are listed below:

below:

below: below: below: below: below: below: below: below:  
below: below: below: below: below:

below: below: below: below: below: below: below: below:  
below: below:

below: below: below: below: below: below: below: below:  
below: below:

below: below: below: below: below: below: below: below:  
below: below: below: below: below: below:

below: below: below: below: below: below: below: below:  
below: below:

below: below: below: below: below: below: below: below:  
below: below: below: below: below: below: below: below:  
below: below:

below: below: below: below: below: below: below:  
below: below: below: below: below: below: below: below:  
below: below: below: below: below: below: below: below:  
below: below: below: below: below: below: below: below:  
below: below: below: below: below: below: below: below:

**Table 2.1:** Internet Explorer capabilities

## Microsoft Edge Driver

Microsoft Edge is the latest web browser launched by Microsoft for Windows 10. It is the first browser to implement W3C WebDriver standard. Moreover, it provides built-in support for Selenium WebDriver. In order to execute scripts on the Edge browser, the EdgeDriver class and WebDriver Server executable have to be downloaded. The server executable can be downloaded from

Shown below is a small code sample for opening the Edge browser, navigating to a URL and verifying the title.

```
public class GoogleTest {  
    WebDriver driver;  
    @BeforeMethod  
    public void setup() {  
        System.setProperty("webdriver.edge.driver",  
            "./src/test/resources/drivers/MicrosoftWebDriver.exe");  
        EdgeOptions opt = new EdgeOptions();  
        opt.setPageLoadStrategy("eager");  
        driver = new EdgeDriver(opt);  
        driver.get("http://www.google.com/");  
    }  
    @Test  
    public void searchText() {  
        WebElement searchBox = driver.findElement(By.name("q"));  
        searchBox.sendKeys("selenium");  
    }  
}
```

```
WebElement srchButton =  
driver.findElement(By.name("btnK"));  
  
srchButton.click();  
  
Assert.assertEquals(driver.getTitle(),"Search results for: 'Phones'");  
}
```

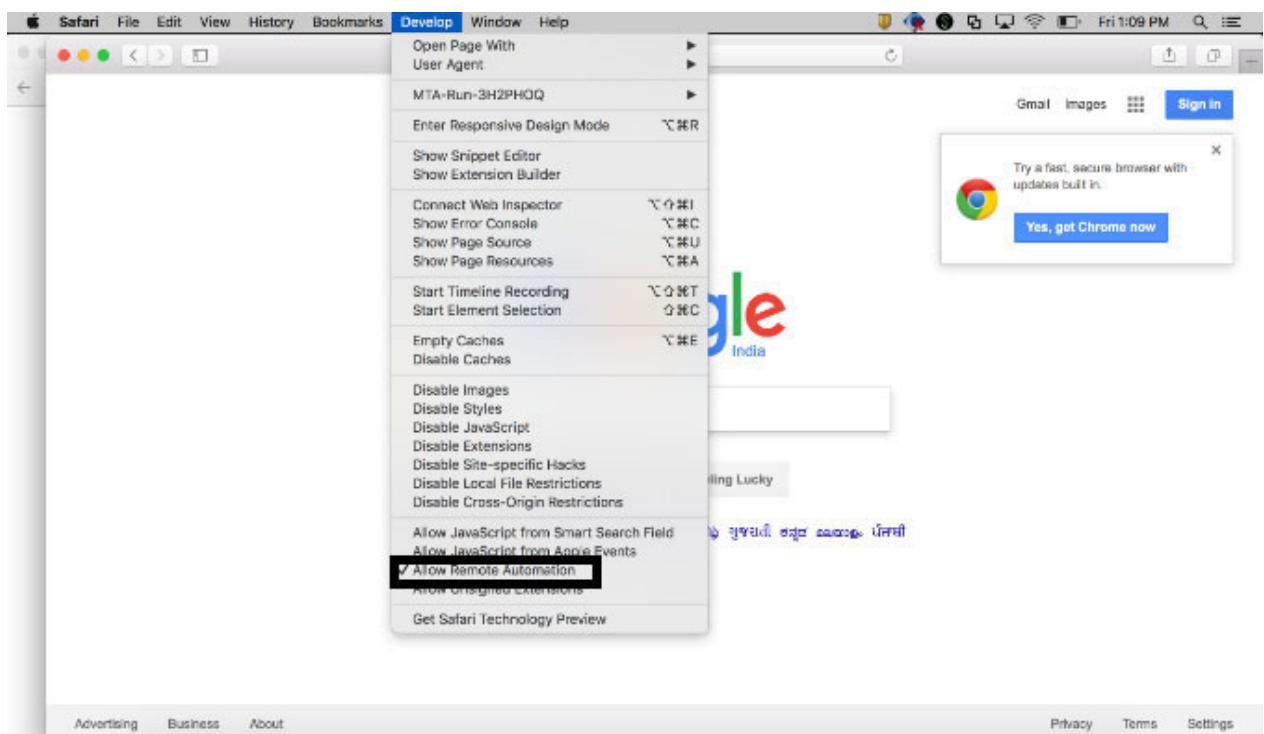
```
@AfterMethod  
public void tearDown() {  
driver.quit();  
}  
}
```

@BeforeMethod, @AfterMethod and @Test are annotations from the TestNG framework. Assert is a class from the TestNG framework. We will be learning the TestNG framework in a separate chapter.

When navigating to a new page URL, Selenium WebDriver, by default, waits until the page has fully loaded before control is sent to the next command. Most of the time, this works well but can cause long wait times on pages that have to load a large number of third-party resources. Using the eager page-load strategy can make a test execute faster.

## Safari Driver

Apple provides the browser with Safari Driver already built-in since WebDriver has become W3C standard. All that needs to be done is set the **Develop | Allow Remote Automation** in the Safari browser.



**Figure 2.5:** Safari settings for instantiation

We will take a look at a small example for Safari Driver next.

### A small code example

We will rewrite the code written for Microsoft Edge making changes specific to Safari Driver as shown below:

```
public class GoogleTest {  
  
    WebDriver driver;  
  
    @BeforeMethod  
    public void setup() {  
        driver = new SafariDriver(options);  
        driver.get("http://www.google.com/");  
    }  
  
    @Test  
    public void searchText() {  
        WebElement searchBox = driver.findElement(By.name("q"));  
        searchBox.sendKeys("selenium");  
        WebElement srchButton =  
            driver.findElement(By.name("btnK"));  
  
        srchButton.click();  
        Assert.assertEquals(driver.getTitle(),"Search results for: 'Phones'");  
    }  
    @AfterMethod  
    public void tearDown() {
```

```
    driver.quit();
}
}
```

It's now time to explore the WebDriverManager library for driver instantiation.

## [\*Introducing WebDriverManager library\*](#)

One constraint in Selenium automation is that every time, we need to download the driver binary executable for Chrome, Firefox, Internet Explorer and Edge. After downloading the executable files for Java, the absolute path of this executable file has to be set to JVM properties using This was overhead and has been removed with the introduction of a library called

**Please note: This works only with JDK 1.8 and above.**

With this library, there is no need to download the individual binaries for the different browsers. Earlier, we had to manage the versions of the binaries manually. With this library, this gets handled automatically.

## How to use the WebDriverManager library

The WebDriverManager library can be used in three ways, as shown below:

As a Java dependency

As a Command Line interface

As a server itself

We will understand how to use each.

## [WebDriverManager as a Java dependency](#)

One of the few ways to use the WebDriverManager library is to use it as a Java dependency. Add the dependency shown in the following code in the dependencies section of pom.xml file. This dependency can be found on GitHub:

```
io.github.bonigarcia  
webdrivermanager  
3.7.0
```

We have already seen how to open Chrome browser session by setting the property value for webdriver.chrome.driver using Now lets see how this can be achieved using the WebDriverManager library. To use the WebDriverManager in your code, place the statements shown below:

```
WebDriver driver = null;  
WebDriverManager.chromedriver().setup();  
driver = new ChromeDriver();
```

These two statements will open up a Chrome browser session for you. As we can see, we don't require to set the driver executable path using System.setProperty. WebDriverManager handles it for us.

The following code fragment displays the use of WebDriverManager for Firefox:

```
private WebDriver driver=null;  
WebDriverManager.firefoxdriver().setup();  
driver = new FirefoxDriver();
```

The following code fragment displays the use of WebDriverManager for Internet Explorer:

```
private WebDriver driver=null;  
WebDriverManager.iedriver().setup();  
driver = new InternetExplorerDriver();
```

Next let us see how to use WebDriverManager from the command line

The full code to accomplish this is shown below.

In the pom.xml insert the code shown below after the tag

```
org.codehaus.mojo  
exec-maven-plugin  
1.2.1  
java  
keywordframework.WebDriverMgr
```

The Java code snippet is shown below:

```
WebDriverManager.chromedriver().setup();
WebDriver driver = new ChromeDriver();
driver.get("https://google.com");
driver.close();
driver.quit();
```

Navigate the project home directory where pom.xml is placed and follow the procedure below.

The command that needs to be typed is mvn exec:java

The output shown below gets displayed in the console, and a new Chrome Session gets opened loading

```
C:\workspace\keywordframework>mvn exec:java -Dexec.args="chrom
[INFO] Scanning for projects...
[INFO]
[INFO] -----
[INFO] Building keywordframework 0.0.1-SNAPSHOT
[INFO] -----
[INFO]
[INFO] >>> exec-maven-plugin:1.2.1:java (default-cli) > validate @
keywordframework
[INFO] >>>
[INFO]
[INFO] <<< exec-maven-plugin:1.2.1:java (default-cli) < validate @
keywordframework
[INFO] <<<
[INFO]
[INFO] --- exec-maven-plugin:1.2.1:java (default-cli) @
keywordframework ---
SLF4J: Failed to load class "org.slf4j.impl.StaticLoggerBinder".
SLF4J: Defaulting to no-operation (NOP) logger implementation
SLF4J: See http://www.slf4j.org/codes.html#StaticLoggerBinder for
further detail|
[INFO] Starting ChromeDriver 76.0.3809.68
(420c9498db8ce8fcfd190a954d51297672c1515d5-ref
s/branch-heads/3809@{#864}) on port 25031
Only local connections are allowed.
Please protect ports used by ChromeDriver and related test frameworks
to prevent
access by malicious code.
Sep 15, 2019 9:21:01 AM org.openqa.selenium.remote.ProtocolHandshake
createSession
[INFO] Detected dialect: W3C
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 23.559 s
[INFO] Finished at: 2019-09-15T09:21:08+05:30
[INFO] Final Memory: 14M/142M
[INFO] -----
```

**Figure 2.6:** WebDriverManager Chrome execution

Let's see how we can use the WebDriverManager as a server next.

## WebDriverManager server

WebDriverManager can also be used as a server. There are two alternatives to use it as a server:

Directly from code using Maven

Using WebDriverManager as a fat JAR

Let us look at both the alternatives

Directly from code using Maven - The command to be used is:

`mvn exec:java -Dexec.args="server"` If the second argument is omitted, the default port (4041) will be used:

`mvn exec:java -Dexec.args="server"`: The console displays as shown below.

```
C:\workspace\keywordframework>mvn exec:java -Dexec.args="serve
"
[INFO] Scanning for projects...
[INFO]
[INFO] -----
[INFO] Building keywordframework 0.0.1-SNAPSHOT
[INFO] -----
[INFO]
[INFO] >>> exec-maven-plugin:1.2.1:java (default-cli) > validate @ keywordframewor
k >>>
[INFO]
[INFO] <<< exec-maven-plugin:1.2.1:java (default-cli) < validate @ keywordframewor
k <<<
[INFO]
[INFO] — exec-maven-plugin:1.2.1:java (default-cli) @ keywordframework —
SLF4J: Failed to load class "org.slf4j.impl.StaticLoggerBinder".
SLF4J: Defaulting to no-operation (NOP) logger implementation
SLF4J: See http://www.slf4j.org/codes.html#StaticLoggerBinder for further detail
5.
Starting ChromeDriver 76.0.3809.68 (420c9498db8ce8fd190a954d51297672c1515d5-ref
s/branch-heads/3809@(#864)) on port 27666
Only local connections are allowed.
Please protect ports used by ChromeDriver and related test frameworks to prevent
access by malicious code.
Sep 15, 2019 9:37:27 AM org.openqa.selenium.remote.ProtocolHandshake createSessi
90
INFO: Detected dialect: W3C
[INFO]
[INFO] BUILD SUCCESS
[INFO]
[INFO] -----
[INFO] Total time: 18.306 s
[INFO] Finished at: 2019-09-15T09:37:35+05:30
[INFO] Final Memory: 13M/142M
[INFO] -----
```

**Figure 2.7:** Chrome driver execution for WebDriverManager server

Making use of WebDriverManager as a fat JAR-We can also use WebDriverManager as a fat JAR for example: Download the webdrivermanager-3.6.2-fat.jar from

<https://github.com/bonigarcia/webdrivermanager/releases/tag/webdrivermanager-3.6.2> Go to the home directory and type the command shown below in the command prompt:

```
java -jar webdrivermanager-3.6.2-fat.jar chrome
```

The console shows the output shown below:

```
[INFO] Using WebDriverManager to resolve chrome  
[DEBUG] Latest version of chromedriver according to https://chromedriver.storage.googleapis.com/LATEST_RELEASE is 77.0.3865.40  
[INFO] Reading https://chromedriver.storage.googleapis.com/ to seek chromedriver  
  
[INFO] Downloading https://chromedriver.storage.googleapis.com/77.0.3865.40/chromedriver_win32.zip  
[INFO] Extracting binary from compressed file chromedriver_win32.zip  
[INFO] Resulting binary C:\workspace\keywordframework\chromedriver.exe
```

**Figure 2.8:** Downloading Chrome using WebDriverManager Fat JAR

Another command that can be given is:

```
java -jar webdrivermanager-3.6.2-fat.jar server
```

The following gets printed in the console:

```
[INFO] WebDriverManager server listening on port 4041
```

When the WebDriverManager is up and running, a REST-like API request using HTTP can be made to resolve driver binaries (chromedriver, geckodriver, etc.). For example, supposing that WebDriverManager is running the local host and in the default port:

Download the latest version of chromedriver

Download the latest version of geckodriver

Download the latest version of operadriver

Download the latest version of phantomjs driver

Download the latest version of msedgedriver

Download the latest version of IEDriverServer

These requests make use of HTTP GET and can be done using a normal browser. The driver binary is automatically downloaded by the browser and is sent as an attachment in the HTTP response.

## Conclusion

This chapter started off with what an instance means in Java and explored the various WebDrivers available and the technique of instantiating each driver. We had an in-depth look at Firefox driver and also understood what Firefox profiles and preferences are. We had a look at Chrome Options and extension and saw few examples. We explored IEDriver and its capabilities. Further Microsoft Edge driver and Safari drivers were explored. Eventually, we had a look at the WebDriverManager library where we looked at utilizing WebDriverManager as a Java dependency and as a server. Last but not least, we saw how to make use of WebDriverManager fat JAR to download the latest driver executables automatically.

In the next chapter, we take a look at Java 8 features like lambda expressions, streams, map function etc.

## Questions

Write a program to login to [newtours.demoaut.com](http://newtours.demoaut.com) using a traditional approach

Write a program to login to [www.freecrm.com](http://www.freecrm.com) in headless mode using Chrome

Write a program to login to [www.freecrm.com](http://www.freecrm.com) using Internet Explorer with few capabilities

Write a program to login to [newtours.demoaut.com](http://newtours.demoaut.com) using WebDriverManager

Download Gecko driver using Fat JAR

## CHAPTER 3

### A Brief Look at Java 8

This chapter will be an overview of the newly added features in Java 8. Java 8 came up with major enhancements compared to Java 1.7. We will be using Java 8 to build our framework and develop reusable components. Each feature will have an accompanying example with the output so that by the end of this chapter, one should be able to construct programs using the Java 8 features.

## Structure

Here's what we will learn in this chapter:

What is functional programming?

Introduction to lambda expressions

Introduction to streams

Introduction to various methods related to streams

Introduction to map function

Working with method references

Filtering WebElements from an ArrayList

## Objectives

This chapter will act as an overview of new features in Java 8

Understand how to utilize lambda expressions with examples

Get up to speed with stream intermediate and terminal operations

Usage of streams with WebDriver

We begin by learning about what functional programming is all about.

## What is a functional programming

Until Java 1.7, Java was predominantly considered as an object-oriented language which revolved around the creation and usage of objects. We were able to store objects like variables, pass objects as parameters to methods and get objects back as return values from a method. In functional programming, we are able to store functions into variables, pass functions as parameter values or get back functions as return values from other methods or functions.

Functional programming consists of three major concepts:

Pure functions

Higher-order functions

Using functions as first-class objects

Let's see each of these in detail

## Pure functions

A function is considered pure if it returns the same value when passed the same set of arguments. It also does not modify any variable that is not local to it. Furthermore, it does not perform any I/O operations.

Let's look at a simple example of a pure function:

```
Public string checknum (int num) {  
    return num%2==0? "Even":"Odd";  
}
```

### **OUTPUT**

**checknum(10) will give Even as output**

Everytime this method is called with the same argument, it returns the same value. For example, calling this method `checknum(10)` always returns the string `Even` irrespective of how many time this statement is executed.

## Higher-order functions

A function that accepts a function as an argument and/or returns a function is known as a higher-order function. We will see both the cases with an example for each.

Let's understand higher-order functions that accept another function as an argument.

## Functions as parameters

Suppose we have a list containing three fruits:

```
ListfruitList = new ArrayList<>();
fruitList.add("Apple");
fruitList.add("Banana");
fruitList.add("Carrot");
```

The Collections.sort function that is specified next, sorts the specified list according to the order that is specified by the associated comparator. All elements in the list must be comparable using the given comparator (that is, `comprtr.compare(elem1, elem2)` should not throw a `ClassCastException` for any elements `elem1` and `elem2` in the list). The sort function shown below accepts a lambda expression as its second argument. lambda expression is nothing but a function.

```
Collections.sort(fruitList, (String fruitA, String fruitB) -> {
    return fruitA.compareTo(fruitB);
});
```

This is what gets printed when we try to display the contents of `fruitList`

## OUTPUT

**[Apple, Banana, Carrot]**

Next, let's understand higher-order functions that return another function.

## Functions as return values

To understand functions as return values, we will consider the reversed function from the Comparator class. The reversed function returns a Comparator that forces the reverse ordering of this comparator. The signature of the reversed function is shown below:

```
default Comparator reversed
```

The practical use of the reversed function is shown below

```
Comparator comparator = (String a, String b) -> {  
    return a.compareTo(b);  
};  
Comparator comparatorReversed = comparator.reversed();  
Collections.sort(list, comparatorReversed);  
System.out.println(list);
```

Here we have created a lambda to compare two strings. By using the reversed function on the comparator object, we get another function which is then passed on as a second argument to the sort method. The output from the code above is shown below

## **OUTPUT**

**[Carrot, Banana, Apple]**

Next, we move on to see how to use functions as first class objects

## *Functions as first class objects*

In functional programming, functions are first class objects in the language. What this means is that one can create an instance of a function, and have a variable to point to that function instance, just like we create reference variables to point to a string, map or other objects. Functions can also be passed as parameters to other functions which we have already seen above.

We will now see what a functional interface is. We already saw some examples of lambda expressions above. In order to learn lambda expressions, it is essential to learn about functional interfaces. lambda expressions can be applied only to functional interfaces.

## Functional interface

A functional interface is an interface that will contain one and only one abstract method. From Java 8 onwards, lambda expressions can be used to represent the instance of a functional interface. A functional interface can have any number of static and default methods. Later, in this chapter, we will see what default and static methods are. Comparable, Runnable, ActionListener are few examples of functional interfaces.

Starting with Java 8, an interface can be marked as a functional interface by annotating it by `@FunctionalInterface`. Using this, one can be sure that nobody else will modify the interface by adding another abstract method declaration. The compiler complains when one tries to add another abstract method to an interface annotated with

This is what a functional interface looks like:

```
@FunctionalInterface  
interface Animal {  
    public static final int legs = 0;  
    public static final String tail = null;  
    public static final String ears = null;  
  
    public default void defaultTest1() {  
        System.out.println("Testing default method 1");  
    }  
}
```

```
public default void defaultTest2() {  
    System.out.println("Testing default method 2");  
}
```

```
public static void staticTest1() {  
    System.out.println("Testing static method");  
}
```

```
public static void staticTest2() {  
    System.out.println("Testing static method");  
}
```

```
public void createSound();  
}
```

As shown above, we have the `@FunctionalInterface` annotation at the top, which declares this interface as a functional interface. In this interface, we have only one abstract method, two static and two default methods. The static methods can be executed using the Interface name, but for executing the default method, a child class has to be created which implements this interface as shown below:

```
public class Lion implements Animal {
```

```
    Public void createSound() {  
        // TODO Auto-generated method stub  
    }
```

```
public static void main(String[] args) {  
  
    Animal lion = new Lion();  
    Animal.staticTest1();  
    Animal.staticTest2();  
    lion.defaultTest1();  
    lion.defaultTest2();  
}  
}
```

Notice how we have created an object of the Lion class using polymorphism where the variable is of type The output in the console is shown below.

## **OUTPUT**

**Testing static method**

**Testing static method**

**Testing default method 1**

**Testing default method 2**

Now is the right time to see what lambda expressions are.

## Lambda expressions

Lambda expression is an important addition to the Java family starting from Java 8. It provides a clear and concise way to represent a functional interface using an expression. It is useful while working with the collections library. It helps to extract, iterate and filter data from any collection.

The lambda expression provides the implementation of a functional interface. In the case of a lambda expression, we just write the implementation code for the abstract method.

Java lambda expressions are treated as functions, so .class file is not created by the compiler as in case of Java files

Let's take the example of the Lion class above, but this time the class does not implement the Animal interface.

```
public class Lion {  
    //Class logic here  
}
```

Using anonymous inner classes, we can provide an implementation for the abstract method, as shown below in the main method:

```
Animal zooLion = new Animal() {  
    @Override  
    public void createSound() {  
        System.out.println("Animal Create Sound anonymous");  
    }  
};  
zooLion.createSound();
```

Consider the abstract method `createSound` shown above, which we need to convert to a lambda expression. Shown below is the transformation into a lambda expression. For clarity, we have created another object here to show that lambda expression is another way of providing an implementation for an abstract method besides anonymous inner classes

```
Animal wildLion = () -> {  
    System.out.println("Animal Create Sound Lambda")  
};  
wildLion.createSound();
```

## OUTPUT

**Animal Create Sound anonymous**  
**Animal Create Sound Lambda**

We have already seen how a lambda with arguments looks like when we saw the example of higher order functions above. The `()` is where you specify arguments.

## Streams

We now touch upon the most important topic of this chapter which is streams. Streams are introduced in Java 8 and are used to process collections of objects. A stream is a sequence of objects that provide various methods which can be executed in sequential order or pipeline to produce the required result.

The most prominent features of Java streams are:

A stream is not a data structure but fetches input from the collections, arrays or I/O.

The original data structure is not changed by Streams, and they only provide the result according to the methods in the pipeline.

Each operation in between the stream is executed lazily and returns a stream as the output, hence various operations in between can be pipelined. Terminal operations mark the end of the stream, and the result is returned.

Let's look at stream operations next.

## Stream operations

There can be intermediate and terminal operations that can be performed with streams. Let's have a look at each.

## Intermediate operations

map: The map method is used to map the items in the collection to other objects according to the function passed as argument.

This works on every item in the collection:

```
List number = Arrays.asList(2,3,4,5,6);
ListsquaredNo = number.stream().map(x-
>x*x).collect(Collectors.toList());
```

### **OUTPUT**

**[4,9,16,25,36]**

The `toList()` static method from the `Collectors` class which returns a `Collector` that collects the input elements into a new `List`. A `Collector` specifies a mutable reduction operation that accumulates input elements into a result container that is mutable, transforming the accumulated result into a final representation, optionally after all input elements have been processed. Reduction operations can be performed either sequentially or in parallel.

filter: The filter method is used to select elements from a collection as per the predicate or condition that is passed as an argument.

```
ListmovieNames = Arrays.asList("Hard  
Rain","TheMatrix","CastAway","Home Alone");  
ListresultList = movieNames.stream().filter(s-  
>s.startsWith("H")).collect(Collectors.toList());
```

## OUTPUT

**[Hard Rain, Home Alone]**

Let's move to see what the sorted operation is.

**sorted:** The sorted method is used to sort the given stream.

```
List names = Arrays.asList("Hard Rain", "The Matrix Reloaded",  
"Cast Away", "Home Alone 2");  
List result = names.stream().sorted().collect(Collectors.toList());
```

If a System.out is included, a sorted list of movies gets printed as shown below.

## OUTPUT

**[Cast Away, Hard Rain, Home Alone 2, The Matrix Reloaded]**

Let's next see what terminal operations mean.

## Terminal operations

The collect method is used to return the result of the in-between operations executed on the stream. We have already seen examples of the collect method in the preceding section.

forEach: The forEach method is used to iterate through the individual elements of the stream as shown below

```
List number = Arrays.asList(2, 3, 4, 5);
number.stream().map(x -> x * x)
.forEach(y ->System.out.println("Individual values:" + y));
```

The code above performs a square on individual elements of the list and then prints out the following:

### **OUTPUT**

```
Individual values:4
Individual values:9
Individual values:16
Individual values:25
```

We now move on to the final operation in our list, which is

This is used to narrow down elements of a list to a single value as shown below

```
Listnumberplusone = Arrays.asList(2, 3, 4, 5);  
int sumtotal = numberplusone.stream().reduce(0, (ans, i) ->ans +  
i);
```

When sumtotal is printed using a `println` statement, the value 14 gets printed which is the total of 2,3,4,5.

## OUTPUT

14

Next we move on to utilizing these operations in Selenium for a list of WebElements

## Filtering WebElements and extracting the count

Let's start by extracting the count of input elements on a page

```
ListinputElem = driver.findElements(By.tagName("input"));
```

Now that inputElem has the list of input elements on the page, we can fetch the count using the statement below

```
long inputCount = inputElem.stream().filter(inputs -> inputs.isDisplayed()).count();
```

### **OUTPUT**

**5**

The code above will extract the number of all elements with the tagName input which are displayed on the screen.

## Fetching elements based on attributes

Let's extract the input elements where the type attribute is a text from the list **inputElem** above.

```
ListtextBoxes = inputElem.stream()  
.filter(item ->item.getAttribute("type") == "text")  
.collect(Collectors.toList());
```

### OUTPUT

**textBoxes will contain all textBoxes**

The code listed above will extract all text boxes in the list **textBoxes**. Let's move on to method references.

## [Introducing method references](#)

Java provides a new feature known as method reference in Java 8. Method reference is used to refer a method of a functional interface. It is an easy version of lambda expression. Each time when lambda expression is used only to refer a method, that lambda expression can be replaced with a method reference.

The method references listed below are available in Java 8:

Referencing a static method

Referencing an instance method

Referencing a constructor

We will see each of these one by one

## Referencing a static method

One can refer to a static method. A static method is one that can be referred to using a class name like Creation of an object is not required.

Below is the example for this scenario.

```
interface PrintMessage{  
    void print();  
}  
  
public class MethodReferenceExample1 {  
    public static void printSomeMessage(){  
        System.out.println("Hello from static method.");  
    }  
    public static void main(String[] args) {  
        PrintMessage message =  
            MethodReferenceExample1::printSomeMessage;  
        message.print();  
    }  
}
```

### **OUTPUT**

**Hello from static method**

In the above example, we have defined a functional interface and referred a static method printSomeMessage to it's functional method This prints out Hello from static method' to the console.

## Referencing an instance method

Similar to static methods, you can refer to instance methods also. An instance method is always referred to using the In the following example, the process of referring to the instance method has been described.

```
interface PrintMessage{  
    void print();  
}  
  
public class MethodReferenceExample2 {  
    public void printSomething(){  
        System.out.println("This is an instance method.");  
    }  
  
    public static void main(String[] args) {  
        MethodReferenceExample2 methodRef = new  
        MethodReferenceExample2();  
        PrintMessage message1 = methodRef::printSomething;  
        message1.print();  
        //Using Anonymous object  
        PrintMessage message2 = new  
        MethodReferenceExample2()::printSomething;  
        message2.print();  
    }  
}
```

## OUTPUT

**This is an instance method**

**This is an instance method**

Next, let's explore what it means by referencing a constructor along with a code example.

## Referencing a constructor

One can refer to a constructor using the new keyword. A constructor is called when a new object of a class is created. Here we are referring constructor with the help of functional interface

```
interface ConstructorMessage{  
    MessagePrintprintMessage(String msg);  
}  
  
class MessagePrint{  
    MessagePrint(String msg){  
        System.out.print(msg);  
    }  
}  
  
public class ConstructorReference {  
    public static void main(String[] args) {  
        ConstructorMessage hello = MessagePrint::new;  
        hello.printMessage("Hello");  
    }  
}
```

### **OUTPUT**

**Hello**

Let's perform some actions on web elements next.

## Performing actions on WebElements

Let's see a code snippet where we will filter the elements based on the visible text.

```
List items = driver
.findElements(By.tagName("a"));

WebElementproductSearch = items.stream()
.filter(item ->item.getText().equalsIgnoreCase("CORN FLOUR"))
.findFirst()
.get();

productSearch.click();
```

### **OUTPUT**

**First product having text corn as flour is clicked**

We click on the first product found. We retrieved the first product using the `findFirst()` method.

## How to use the map function to get the text value of elements

We will be using method references in this section to get the visible text of elements. The code snippet to accomplish this is shown below.

```
List items = driver
.findElements(By.cssSelector("h2.prod-name a"));
ListprodNames = items.stream()
.map(WebElement::getText)
.collect(Collectors.toList())
```

### **Output**

**The list prodNames should contain all WebElements whose cssSelector matches ‘h2.prod-name a’, an ‘h2’ element.**

Here we have used the getText method from the WebElement interface using method reference.

## Conclusion

This chapter started with what functional programming is and explained pure functions, higher-order functions and functions as first class objects. We explored a very important concept of Java 8, which is functional interface and the related concept of Lambda expressions. We saw what Streams are and what Intermediate and Terminal operations related to streams are. We then explored several ways of dealing with a list of WebElements using filter and map operations. Java 8 reduces most of the overhead of looping through a list of WebElements which was unavoidable till Java 1.7.

This chapter taught us how to work with various Java 8 features with code snippets for each.

In the next chapter, we take a look at the difference between navigate and get, different types of waits and polling, navigating to pop up windows and frames.

## Questions

Mention the version of Java when Streams API was introduced

Write a program to filter WebElements with visible text Header

Create a sample web page with 5 disabled text boxes. Write a program to enable all disabled textboxes.

Write a simple program to demonstrate filter operation

Write a simple program to demonstrate reduce operation

## CHAPTER 4

### Deep Dive into Selenium WebDriver

This chapter will dive deep into the various features of Selenium WebDriver. Here is where we will start creating the framework piece by piece. We will look at concepts involving pop up windows and frames, waiting for elements to display, waiting for page loads, and polling. We will have the skeleton of the framework ready by the end of this chapter. The way we will be going through this chapter is by understanding the concepts and creating a reusable method for each. These reusable methods will be added to a class called ActionKeywords. In the end, we will create the heart of our framework, which is the Singleton driver to run the framework.

## Structure

Here's what we will learn in this chapter.

Understanding ThreadLocal

Understanding the Singleton pattern and creating a Singleton driver

Handling pop-up windows, alerts and frames

Waiting for elements to load using explicit wait

Understanding fluent waits

Handling page loads

Waiting for Ajax calls to complete

## Objectives

Understand what the ThreadLocal class is

Understand how to create Singleton WebDriver

How to use ThreadLocal with Singleton WebDriver

Framework creation initial steps

Understand pop-up windows, alerts, and frames

Understand concepts such as polling

Understand what page loads are

Take a look at Ajax calls

We begin by first understanding the ThreadLocal class and move on to understanding components that we will be creating as a part of the framework. The first component is the ActionKeywords class.

### [ThreadLocal class](#)

ThreadLocal is a class in Java that enables us to create variables that can be read and written by the same thread. Thus, even if two threads are executing the same code, and the code has reference to a ThreadLocal variable, then the two threads cannot see each other's ThreadLocal variables.

In cases where we have a shared object, and that object has certain properties that get updated by different threads, and the update made in the property by one thread is not reflected in another thread and stays local to a particular thread, then the ThreadLocal class should be used.

## Using ThreadLocal

To use an object of the ThreadLocal class has to be created as shown below

```
Private ThreadLocalthreadLocal = new ThreadLocal();
```

Next, we set the value of the object using the set() method as shown below

```
threadLocal.set("ThreadLocal variable");
```

To get the value back from a ThreadLocal variable, we use ThreadLocal'sget() method as shown below

```
String value = threadLocal.get();
```

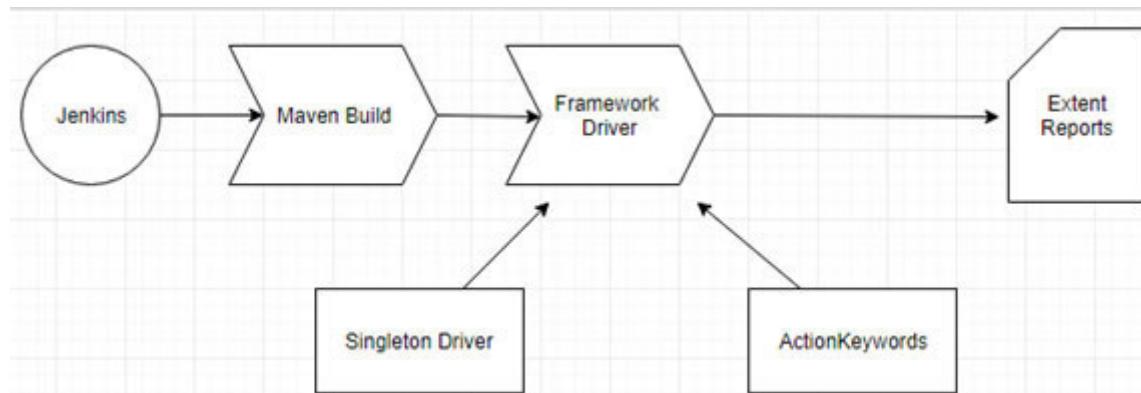
Here we have used generics to indicate that we are dealing with String values that are stored in the ThreadLocal variable, but ThreadLocal variables can be used to store any object as we will see next.

## Framework design

Let's now get our hands dirty and design a few blocks of our framework. To design a framework, it is very necessary to think about the architecture of the framework which we will understand best with the help of an architecture diagram

## Framework architecture

The first step in the creation of a framework is understanding the architecture diagram. As shown below, Jenkins triggers a Maven build which invokes the FrameworkDriver class logic which invokes the database logic to extract test cases which have Execute flag set to Y, the test steps corresponding to the test case, the required objects from the OR table and the data to be supplied for each test case. Finally, it generates Extent Reports with the results of the test execution.



**Figure 4.1**

Let's start understanding each component from this architecture diagram. We begin with the ActionKeywords class.

## ActionKeywords class

The main chunk of our code will lie in the ActionKeywords class. The code for individual keywords will reside in this class. The example keywords will be openbrowser, navigatetourl, clickElement, enterText, selectItem, waitforPageLoad, etc. Generic methods can be added to this class to build the framework. The structure of the ActionKeywords class is given below. Two keywords for opening the browser and navigating to URL have been coded. When both these keywords are invoked, a chrome browser will be launched, and the URL that is passed as a parameter will be navigated to.

```
package keywordframework;
public class ActionKeywords {
    //Step 1 Create an instance of SingletonDriver class
    SingletonDriver driver = SingletonDriver.getInstance();
    public void openBrowser(String browser) {
        try {
            driver.setDriver(browser, "windows", "windows");
            driver.getDriver();
        } catch (Exception e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
    }
    public void navigateURL(String URL) {
```

```
driver.getDriver().get(URL);
}
}
```

Let's see what the FrameworkDriver class is all about.

## FrameworkDriver class

This class can be considered as our main class. This will be running the entire framework. It will read the keyword from the database, call the corresponding method in the ActionKeywords class, and then move on to the next keyword.

The class shown below will open the chrome browser and navigate to

```
package keywordframework;
public class FrameworkDriver {
    public static void main(String[] args) {
        ActionKeywords act = new ActionKeywords();
        act.openBrowser("chrome");
        act.navigateURL("http://www.google.com");
    }
}
```

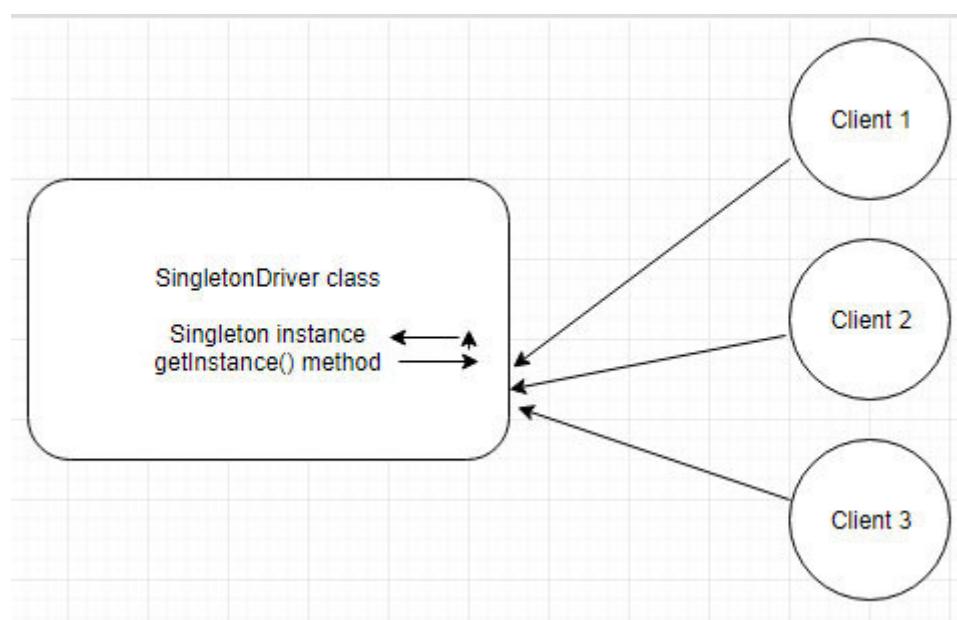
We will eventually be pulling the keywords from the database. The method shown above is a quick way to verify whether the framework works.

Let's have a look at the SingletonDriver class, which creates the driver instance for us, but first, we need to know about the Singleton pattern.

## Singleton pattern?

Sometimes it is necessary to have only one instance of a particular class; for example, in our case, we require only one browser session to execute one test case. In such a scenario, we need to utilize the Singleton pattern, which comprises of creating only one instance of a particular class.

The implementation of the Singleton pattern is done by using a private Constructor and a static getInstance method, which provides a single instance of the class. By using a private Constructor, we ensure that no other class creates a new instance of the class.



*Figure 4.2*

Now to implement this pattern, we create a `SingletonDriver` class with a private constructor and a `getInstance()` method, which is static and provides an instance of the `SingletonDriver` class. The instance stored in a private static variable of type

## [Creating a Singleton driver](#)

In this section, we will create a Singleton driver that implements the Singleton pattern.

Shown below is the structure of the SingletonDriver class:

```
@SuppressWarnings("varargs")
public class SingletonDriver {
    // local variables
    private static SingletonDriver instance = null;
    private String browserHandle = null;
    private static final int IMPLICIT_TIMEOUT = 0;
    private ThreadLocalwebDriver = new ThreadLocal();
    private ThreadLocalsessionId = new ThreadLocal();
    private ThreadLocalsessionBrowser = new ThreadLocal();
    private ThreadLocalsessionPlatform = new ThreadLocal();
    private ThreadLocalsessionVersion = new ThreadLocal();
    private String getEnv = null;
    public WebDriver getDriver() {
        return webDriver.get();
    }

    @SafeVarargs
    public final void setDriver(String browser, String environment,
        String platform, MapObject>... optPref) throws Exception {
```

```
FirefoxOptionsfirefoxopt = null;

ChromeOptionschromeopt = null;
InternetExplorerOptionsieopt = null;
SafariOptionssafariopt = null;
EdgeOptionsedgeopt = null;
String localHub = "http://127.0.0.1:4723/wd/hub";
String getPlatform = null;

switch (browser) {
    case "firefox":
        firefoxopt = new FirefoxOptions();
        webDriver.set(new FirefoxDriver(firefoxopt));
        break;
    case "chrome":
        chromeopt = new ChromeOptions();
        webDriver.set(new ChromeDriver(chromeopt));
        break;
    case "internet explorer":
        ieopt = new InternetExplorerOptions();
        webDriver.set(new InternetExplorerDriver(ieopt));
        break;
    case "safari":
        safariopt = new SafariOptions();
        webDriver.set(new SafariDriver(safariopt));
        break;
}
```

```

        case "microsoftedge":
            edgeopt = new EdgeOptions();
            webDriver.set(new EdgeDriver(edgeopt));

            break;
    }
}

// constructor
private SingletonDriver() {
}

/**
 * getInstance method to return active driver instance
 *
 * @return CreateDriver
 */
public static SingletonDriver getInstance() {
    if (instance == null) {
        instance = new SingletonDriver();
    }
    return instance;
}
}

```

We can see that if the instance variable is null, only then will the getInstance() method create a new instance of SingletonDriver class. The getInstance() static method is invoked in the ActionKeywords class, where we have logic for the openbrowser keyword. We invoke the setDriver() method next, which uses the

set method of the webDriverThreadLocal to pass the value of chrome while creating the appropriate driver instance. Finally, we get the current thread's copy of the ThreadLocal variable using the get method, which is invoked on the same webDriver variable.

When the FrameworkDriver class's main method gets executed, a Chrome browser is opened, and **http://www.google.com** is loaded in the browser.

Next, we understand a slightly advanced concept of handling pop up windows and frames.

## Handling pop-up windows

A pop-up window is a window that pops up as a child window when a button or link is clicked. We will be learning how to handle pop-up windows in this section.

A few concepts are revolving around handling pop-up windows. Let's have a look at those concepts:

Window handles

Understanding the Set interface

Learning the iterator method

Let's understand what window handles are.

## Window handles

A window handle is an alphanumeric id that gets assigned to each window that gets opened after the WebDriver object is instantiated. Selenium uses this id to identify open windows and switch between those. There are two methods that Selenium provides, namely:

`getWindowHandle()`: This method returns the handle of the current window having a focus, when invoked on a WebDriver object

`getWindowHandles()`: This method returns a set of handles for all open windows when invokes on a WebDriver object

We will set what the Set interface is next.

## The Set interface

Set is an interface that extends the collection interface. In a Set, duplicate values cannot be contained. The values stored in a set cannot be accessed using an index since these values are not stored in an ordered manner. Three different classes implement the Set interface, which is listed below:

HashSet: Contains unique elements and stores elements using a caching mechanism

LinkedHashSet: The insertion order is maintained in a linked HashSet

TreeSet: An ascending order is maintained in a TreeSet

Set interface contains methods such as add(), isEmpty(), contains(), iterator(), remove(), size(), and so on, out of which we will be looking at the use of iterator.

A look at the iterator() method

Each class in the collections framework provides an iterator() method to loop over the elements in a collection. Given below are the steps to use the iterator() method on a set containing window handles. Please add the code for the creation of a

WebDriver object, loading of a URL using get() method, and clicking a particular link that opens a pop-up.

Step 1: Get the window handle of the window which currently has the focus

```
String parentWindow = driver.getWindowHandle(); [Note that before using this line a WebDriver instance called driver has to be instantiated]
```

Step 2: Get the handles of all open windows. We put all the handles in a set of Strings

```
Set<String> openWindows = driver.getWindowHandles();
```

Step 3: Use iterator() to iterate the set of Window handles

```
Iterator<String> iterator1 = openWindows.iterator();
```

Step 4: Use a while loop and print title of each window

```
while(iterator1.hasNext()){
    String childWindow = iterator1.next();
    If (!parentWindow.equals(childWindow)){
        driver.switchTo.window(childWindow);
        System.out.println("Window title is:
"+driver.switchTo.window(childWindow).getTitle());
        driver.close();
    }
}
```

Step 5: Immediately after the closing brace of the while loop,  
switch back to the main window

```
driver.switchTo.window(parentWindow);
```

Let's move on to Alerts and how to handle those.

## Handling Alerts

Alerts are small boxes that appear on the screen to provide some information. We will be making use of the Alert interface, which has four methods listed below:

accept(): To accept the alert

dismiss(): To dismiss the alert

getText(): To capture text present on the alert

sendKeys(): To write some text to the alert

## Steps to handle Alerts

Handling alerts involves slightly different steps when compared to handling pop-ups. Follow the steps shown below. Note that a WebDriver object should be instantiated, a URL should be loaded and a button or link that opens an Alert should be clicked before these steps

Step 1: Switch to the Alert using the code below:

```
Alert alert = driver.switchTo.alert();
```

Step 2: Using the alert object, get the text on the alert, and accept the alert:

```
String alertText = alert.getText();
alert.accept();
```

Any text, if present, will be available in the alertText variable.

We will now see how to handle frames in Selenium WebDriver.

## [\*\*Handling frames/iFrames\*\*](#)

HTML frames divide the browser window into multiple sections where each section can load a separate HTML document. A collection of frames in a browser window is known as a frameset.

Frames are also handled using switchTo() method, but the frame() method that is chained to the switchTo() method has 4 overloaded versions, as shown below.

frame(int frameNumber): The frame index is passed as an argument so that selenium can switch to that frame

frame(String nameorid): The frame name or id is passed as an argument so that selenium can switch to that frame

frame(WebElement elemFrame): The frame webelement is passed as an argument so that selenium can switch to that frame

## Steps to handle frames/iFrames

Handling frames are mostly similar when compared to handling pop-ups. Follow the steps shown below. Note that a WebDriver object should be instantiated and a URL should be loaded before these steps

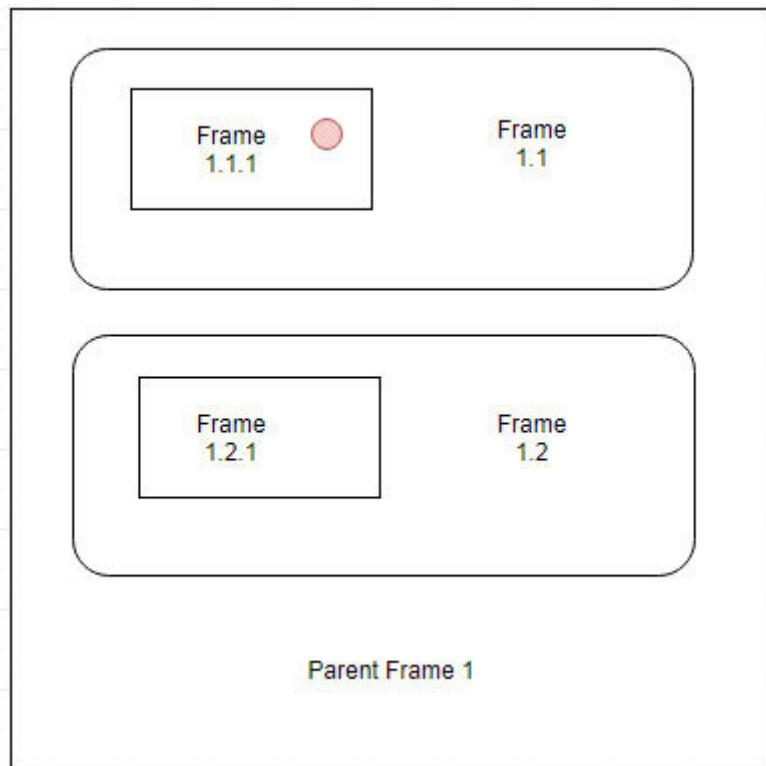
Step 1: Switch to the frame/iframe using the code below:

```
driver.switchTo.frame(frameName);
```

After this switch, we perform actions on webelements present in the frame

### Steps to handle nested frames/iFrames

One may be faced with a situation where there are frames inside frames, as shown below, where there is a total of 5 frames. Two frames are nested within the parent frame. Suppose the focus is in Frame 1.1.1, as indicated by the colored dot. Imagine we want to access elements in frame 1.2.1.



**Figure 4.3**

To navigate to a child frame that is inside another frame, switching to the parent window is critical. This is done using `driver.switchTo.defaultContent();`

Using this will switch to the main window. Let's list the code steps to navigate to frame 1.2.1 from 1.1.1

Step 1: Switch to the parent window using the code below:

```
driver.switchTo.defaultContent();
```

Step 2: Switch to the parent frame 1 using the code below. For this suppose, the name of the parent frame is parentFrame:

```
driver.switchTo.frame("parentFrame");
```

Step 3: Switch to the child frame 1.2 using the code below. For this suppose, the name of the child frame is Frame1\_2:

```
driver.switchTo.frame("Frame1_2");
```

Step 4: Switch to the child frame 1.2.1 using the code below. For this suppose, the name of the child frame is Frame1\_2\_1:

```
driver.switchTo.frame("Frame1_2_1");
```

Now you can access elements in frame 1.2.1.

This completes all concepts related to pop-up windows, alerts, and frames. Now we move on to another important topic, which is synchronization.

## What is synchronization?

The dictionary meaning of synchronization states that it is the operation or activity of two or more things happening at the same time. One can relate this to an automated car assembly where various events are synchronized with each other, and a dependent activity waits for the parent activity before it can perform some action.

In WebDriver terms, synchronization relates to matching the speed of **AUT(Application Under Test)** with the testing tool, which is Selenium, in our case. Selenium operates very rapidly and should be asked to slow down to match the speed of the AUT.

We will be looking at the following things in this section:

Synchronization at WebDriver instance level

Synchronization at WebElement level

Let's look at each of these.

## Synchronization at WebDriver instance level

There are three kinds of waits in this category:

**Implicit wait:** Applicable to all web elements on a web page

**Page load timeout:** Specify a page load time after which a PageLoadTimeout Exception will be thrown

**setScriptTimeout:** Used to set the timeout for asynchronous script execution

Let's start by understanding the implicit wait.

## Implicit wait

The way to use the implicit wait is by specifying it as  
driver.manage().timeouts().implicitlyWait(15,

The manage() method shown above, when invoked on the WebDriver instance, returns an instance of the options interface, on which we invoke the timeouts() method present in the options interface. The timeouts() method sends back an instance of the timeouts interface. We then invoke the implicitlyWait(long, TimeUnit) method present in the timeouts interface. The implicitlyWait method being a factory method returns an object of the same interface, which is timeouts.

When searching for a single element, the driver instance should perform an activity called polling until the element under consideration is found. In case the element is not found, or this timeout expires, a NoSuchElementException is thrown. When multiple elements are being scanned in the Document Object Model, the driver instance polls the page until it finds at least one element, or this timeout expires.

We will create a page using JavaScript, which will have a button. When the button is clicked, various fruit names will get displayed at a time interval of 5 seconds. Listed below is the code for the web page:

```
onclick="timerCheck()">Timer Check
```

```
id="demoTimer">Click on Timer Check
```

Let's write a small program that prints Orange is displayed when an element with the text Orange if found on screen. Note that, as per the preceding JavaScript program, Orange appears after 10 seconds.

The code to print details such as whether an Orange is displayed and the exact time required to find the element is shown below. The start time and end time are noted, and a difference in time is measured:

```
long startTime = oL;  
long endTime = oL;  
boolean status = false;  
  
System.setProperty("webdriver.chrome.driver",  
"C:\\\\SeleniumWD\\\\src\\\\main\\\\resources\\\\chromedriver.exe");  
WebDriver driver = new ChromeDriver();  
driver.manage().window().maximize();  
driver.navigate().to("C:\\\\Timer.html");  
driver.findElement(By.xpath("//*[text()='Timer Check']")).click();  
startTime = System.currentTimeMillis();  
driver.manage().timeouts().implicitlyWait(20, TimeUnit.SECONDS);  
try {  
status = driver.findElement(By.xpath("//*[text()='Orange']"))  
isDisplayed();
```

```
} catch (NoSuchElementException exception) {  
    System.out.println(exception.getMessage());  
}  
if (status) {  
    System.out.println("Orange is displayed");  
} else {  
    System.out.println("Orange is not displayed");  
  
}  
endTime = System.currentTimeMillis();  
long timeDiff = endTime - startTime;  
timeDiff = timeDiff / 1000;  
System.out.println("Start Time: " + startTime);  
System.out.println("End Time: " + endTime);  
System.out.println("Difference in Time: " + timeDiff);
```

#### **OUTPUT:**

**Element is displayed**  
**Start Time: 1570963797421**  
**End Time: 1570963807505**  
**Difference in Time: 10**

Let's move on to page load timeout.

## PageLoad Timeout

The page load timeout remains in action for the lifetime of the driver. It sets the amount of time for a page to load completely before throwing a Let's take the example of a website called The landing page takes some time to load before it can display the login fields fully.

The script shown below displays the use of where we deliberately set the PageLoadTimeout as 5 seconds:

```
System.setProperty("webdriver.chrome.driver",
"C:\\\\SeleniumWD\\\\src\\\\main\\\\resources\\\\chromedriver.exe");
WebDriver webDriver = new ChromeDriver();
webDriver.manage().timeouts().pageLoadTimeout(5,
TimeUnit.SECONDS);
webDriver.manage().window().maximize();
webDriver.navigate().to("http://www.freecrm.com");
```

This script will throw a Timeout exception, as shown in the following output:

The screenshot shows the Eclipse IDE interface with the title bar "Java - keywordframework/src/main/java/keywordframework/TestOrange.java - Eclipse". The menu bar includes File, Edit, Source, Refactor, Navigate, Search, Project, Run, Window, Help. The toolbar has various icons for file operations. The top status bar shows "File Edit Source Refactor Navigate Search Project Run Window Help". The bottom status bar shows "4:42 PM 13/10/2019". The central workspace displays the Java code for "TestOrange.java" and its execution output in the "Console" tab. The output shows the Selenium WebDriver starting up, detecting the dialect as OSS, and encountering a timeout exception during the handshake process. The stack trace points to several methods in the org.openqa.selenium.remote package, specifically related to Error Handler and JSON response decoding.

```
Starting ChromeDriver 2.41.578737 (49da6702b16031c40d63e5618de03a32ff6c197e) on port 17173
Only local connections are allowed.
Oct 13, 2019 4:41:26 PM org.openqa.selenium.remote.ProtocolHandshake createSession
INFO: Detected dialect: OSS
[1570965093.061][SEVERE]: Timed out receiving message from renderer: 2.859
[1570965093.065][SEVERE]: Timed out receiving message from renderer: -0.004
Exception in thread "main" org.openqa.selenium.TimeoutException: timeout
(Session info: chrome=77.0.3865.90)
(Driver info: chromedriver=2.41.578737 (49da6702b16031c40d63e5618de03a32ff6c197e),platform=Windows NT 6.1.7600 x86_64) (WARNING: The command duration or timeout: 0 milliseconds)
Build info: version: '3.141.59', revision: 'e82be7d358', time: '2018-11-14T08:17:03'
System info: host: 'BHAGYASHREE-PC', ip: '192.168.225.56', os.name: 'Windows 7', os.arch: 'amd64', os.version: '6.1', java.version: '1.8.0_201-b10'
Driver info: org.openqa.selenium.chrome.ChromeDriver
Capabilities {acceptInsecureCerts: false, acceptSslCerts: false, applicationCacheEnabled: false, browserConnectionEnabled: false, ...
Session ID: 87dfb5bc5385229566ffe281578de3b4
    at sun.reflect.NativeConstructorAccessorImpl.newInstance0(Native Method)
    at sun.reflect.NativeConstructorAccessorImpl.newInstance(Unknown Source)
    at sun.reflect.DelegatingConstructorAccessorImpl.newInstance(Unknown Source)
    at java.lang.reflect.Constructor.newInstance(Unknown Source)
    at org.openqa.selenium.remote.ErrorHandler.createThrowable(ErrorHandler.java:214)
    at org.openqa.selenium.remote.ErrorHandler.throwIfResponseFailed(ErrorHandler.java:166)
    at org.openqa.selenium.remote.http.JsonHttpResponseCodec.reconstructValue(JsonHttpResponseCodec.java:40)
    at org.openqa.selenium.remote.http.AbstractHttpResponseBodyCodec.decode(AbstractHttpResponseBodyCodec.java:80)
    at org.openqa.selenium.remote.http.AbstractHttpResponseBodyCodec.decode(AbstractHttpResponseBodyCodec.java:44)
```

*Figure 4.4*

Another timeout that can be applied at the driver-instance level is the `setScriptTimeout`. This timeout requires a good understanding of the `JavaScriptExecutor`, which will be covered in a later chapter; `setScriptTimeout` will be discussed then.

## Synchronization at WebElement level

We will now learn about synchronization at the WebElement level.  
Two waits happen to fall in this category:

Explicit wait

Fluent wait

We start by understanding the ExplicitWait.

## Explicit wait

To understand explicit waits, we will have to understand the WebDriverWait class. This class extends the FluentWait class, and the FluentWait class implements the wait interface.

The WebDriverWait constructor comprises of three overloaded versions. We will be mostly using the version with two arguments. The signature of the WebDriverWait constructor with two arguments is public. The way to instantiate a WebDriverWait object is by using the code as follows:

```
WebDriverWait explicitWait = new WebDriverWait(driver,12);
```

Here, we specify the maximum wait time as 12 seconds. To understand the explicit wait, we should first understand a class called WebDriverWait. This class has several methods for waiting for a WebElement to be displayed before throwing an exception. A few of these methods are as follows:

presenceOfElementLocated: Checks if the element is available on the DOM of a page.

visibilityOfElementLocated: Checks whether the element is available on the DOM of a page and is also visible.

frameToBeAvailableAndSwitchToIt: Checks whether a given frame is available. If it is, it then switches to the frame.

refreshed: when you expect DOM manipulation to occur, and you want to wait until the manipulation is complete.

Lets check the same program this time using explicit wait:

```
long startTime = oL;
long endTime = oL;
boolean status = false;
System.setProperty("webdriver.chrome.driver",
"C:\\\\SeleniumWD\\\\src\\\\main\\\\resources\\\\chromedriver.exe");
WebDriver driver = new ChromeDriver();
WebDriverWait wdWait = new WebDriverWait(driver, 12);
driver.manage().window().maximize();
driver.navigate().to("C:\\\\Users\\\\Bhagyashree\\\\Desktop\\\\Documents\\\\
Timer.html");
wdWait.until(ExpectedConditions.presenceOfElementLocated(By.xpath(
"//*[text()='Timer Check']"))).click();
startTime = System.currentTimeMillis();
try {
wdWait.until(ExpectedConditions.presenceOfElementLocated(By
.xpath("//*[text()='Orange']")));
status = driver.findElement(By.xpath("//*[text()='Orange']"))
.isDisplayed();
} catch (NoSuchElementException exception) {
System.out.println(exception.getMessage());
}
if (status) {
```

```
System.out.println("Orange is displayed");
} else {
System.out.println("Orange is not displayed");
}
endTime = System.currentTimeMillis();

timeDiff = endTime - startTime;
timeDiff = timeDiff / 1000;
System.out.println("Start Time: " + startTime);
System.out.println("End Time: " + endTime);
System.out.println("Difference in Time: " + timeDiff);
```

The output from the preceding program is as follows:

```
Orange is displayed
Start Time: 1570968212162
End Time: 1570968222419
Difference in Time: 10
```

Let's checkout fluent wait next.

## Fluent wait

A fluent wait is the implementation of the wait interface in which the timeout and the polling interval are configured dynamically. Apart from this, it enables us to add exception exclusions, such as the example that follows, where we ignore the NoSuchElementException exception. Each instance of the fluent wait defines the timeout and the polling time or interval. The polling interval specifies the frequency with which to check for the presence of the element under consideration.

Let's take the HTML we created earlier and try to find out whether we are able to locate the element under consideration using fluent wait:

```
long startTime = oL;
long endTime = oL;
WebElement elem = null;
boolean status = false;
System.setProperty("webdriver.chrome.driver",
"C:\\\\SeleniumWD\\\\src\\\\main\\\\resources\\\\chromedriver.exe");
WebDriver driver = new ChromeDriver();
WaitwdWait = new FluentWait(driver)
.withTimeout(30, TimeUnit.SECONDS)
.pollingEvery(1, TimeUnit.SECONDS)
.ignoring(NoSuchElementException.class);
driver.manage().window().maximize();
driver.navigate().to(
```

```
"C:\\\\Users\\\\Bhagyashree\\\\Desktop\\\\Documents\\\\Timer.html");  
wdWait.until(  
ExpectedConditions.presenceOfElementLocated(By  
  
.xpath("//*[text()='Timer Check']")).click();  
startTime = System.currentTimeMillis();  
try {  
elem = wdWait.until(new FunctionWebElement>() {  
public WebElement apply(WebDriver driver) {  
WebElement text1 = driver.findElement(By  
.xpath("//*[@id='demoTimer']"));  
String value = text1.getAttribute("innerHTML");  
if (value.equalsIgnoreCase("Orange")) {  
return text1;  
} else {  
System.out.println("Text on screen: " + value);  
return null;  
}  
}  
}  
});  
} catch (NoSuchElementException exception) {  
System.out.println(exception.getMessage());  
}  
if (elem.isDisplayed()) {  
System.out.println("Element is displayed");  
} else {  
System.out.println("Element is not displayed");  
}  
endTime = System.currentTimeMillis();  
long timeDiff = endTime - startTime;  
timeDiff = timeDiff / 1000;  
System.out.println("Start Time: " + startTime);
```

```
System.out.println("End Time: " + endTime);
System.out.println("Difference in Time: " + timeDiff);
}
```

The output from this program is as follows:

```
Text on screen: Click on Timer Check
Text on screen: Apple
Element is displayed
Start Time: 1570969252287
End Time: 1570969262815
Difference in Time: 10
```

This output clearly shows that the DOM is polled every second and each piece of text appears five times. The reason for this is that in the JavaScript, the various text items are displayed after a delay of five seconds.

The polling stops when the required text is found, and we get the exact time as 10 seconds.

Let's move on to the final topic in this chapter, which is waiting for AJAX calls to complete.

## Handling AJAX calls

Many websites nowadays have ajax calls triggering in the background. AJAX stands for Asynchronous JavaScript and XML. In order to perform automation on such websites, it is very essential to wait till all the AJAX calls in the background complete before we can interact with elements on the page. We take the example of classic CRMPRO website where we will wait till the AJAX calls complete and the username textbox is visible. ‘admin’ is then entered into the username textbox. Please note, we will learn JavascriptExecutor in a later chapter. The code shown below accomplishes this:

```
boolean status = false;
JavascriptExecutor js = null;
Boolean isJqueryCallDone = false;
System.setProperty("webdriver.chrome.driver",
"C:\\\\SeleniumWD\\\\src\\\\main\\\\resources\\\\chromedriver.exe");
WebDriver driver = new ChromeDriver();
driver.manage().window().maximize();
driver.navigate().to("https://classic.crmpro.com/index.html?e=2");
int counter = 0;
js = (JavascriptExecutor) driver;
isJqueryCallDone = (Boolean) js
.executeScript("return jQuery.active==0");
System.out.println("JQuery call done: " + isJqueryCallDone);
driver.findElement(By.xpath("//input[@name='username']")).sendKeys(
"admin");
```

**OUTPUT:**

**JQuery call done: true**

Once the AJAX calls on the page complete, the text admin is entered in the username textbox.

Lastly, we will understand the differences between navigate() and get() methods

## Differences between get() and navigate()

navigate()

navigate() navigate() navigate() navigate() navigate() navigate()  
navigate() navigate() navigate() navigate() navigate() navigate()  
navigate() navigate() navigate() navigate() navigate()

navigate() navigate() navigate() navigate() navigate() navigate()  
navigate() navigate() navigate() navigate() navigate()

navigate() navigate() navigate() navigate() navigate() navigate()  
navigate() navigate() navigate() navigate() navigate() navigate()  
navigate() navigate() navigate()

This concludes a very important chapter.

## Conclusion

This chapter was important from learning important concepts like ThreadLocal class, the architecture of the framework that we will be building, and the various patterns. We learned about the Singleton pattern and created the Singleton driver. We had an exhaustive look at handling pop-ups, alerts, and frames. Eventually, we learned about a very important concept, which is synchronization and explored different types of waits.

In the next chapter, we take a look at taking screenshots, the Actions and the JavascriptExecutor in depth. Simultaneously, we will enrich the ActionKeywords class with logic for more keywords.

## Questions

Write a program to enter values in a textbox with id username that is located in a frame with the name

Write a program to search for Selenium on Google and move forward and backward in the browser's history.

Create a sample web page with a link which, when clicked, opens an alert. Using Selenium, capture text on that Alert.

Write a program to handle multiple pop-ups using Capture the text in each popup.

Write a simple program to wait for Gmail to display the username field.

## CHAPTER 5

### *Actions Class and the Javascript Executor*

It's time to do some advanced interactions with Selenium WebDriver. Certain actions cannot be performed using the basic Selenium commands. Such actions include double-click, right-click, mouse over, etc. We then explore the JavascriptExecutor class. This class is used when the standard WebDriver API fails to locate WebElements or perform actions on WebElements.

While working with examples of Actions class and JavaScript executor, we will create two new classes ActionsExecutor and which will provide keyword logic for each. The normal keywords related to Selenium WebDriver will be in the ActionKeywords class, which we have seen before.

## Structure

Here's what we will learn in this chapter:

Learning about the advanced interactions

Understanding the Actions class

Creating the ActionsExecutor class

Understanding the JavascriptExecutor class

Creating the JavascriptUtility class

Integrating the new classes with the framework

## Objectives

Understanding advanced interactions with Selenium WebDriver

Learning to code handling of right clicks, double clicks, drag, and drop, etc.

Learn to write generic keywords

Understanding the Actions class

Understanding the javascriptExecutor class

Creating two new classes for implementing the logic for advanced interactions

Integration with the framework

We begin by first understanding the advanced interactions that can be performed on Selenium WebDriver.

## What are advanced interactions?

There are few events that the user can perform apart from the normal click() or which fall into the category of advanced interactions. The regular Selenium API does not provide this functionality through which we can perform these interactions.

Advanced interactions with Selenium WebDriver include double-clicking, right-clicking, mouse over, dragging, and dropping, enabling a disabled WebElement, clicking on a web element that cannot be clicked using the regular click() method, setting the value of a text box using JavaScript.

Many websites trigger JavaScript functions on the click of a button. Instead of clicking on the button, the underlying JavaScript can be executed using the JavascriptExecutor class. We will write generic methods to execute JavaScript.

There are two candidates available for achieving advanced interactions: 1) Actions class 2) JavascriptExecutor interface. We are going to study these two in detail in this chapter, which will help the reader perform complex operations on a web page.

Let's begin by learning about the Actions class.

## [Understanding the Actions class](#)

Actions class is used to emulate all the complex user actions. The complex user actions can be combined into a composite user action by grouping or chaining them together. This makes use of a design pattern called the builder design pattern, where the composite action gets built using the individual actions.

Action is an interface that contains only one method which is This method is responsible for executing the composite action. The perform() method internally calls the build() method, which builds the composite action.

Actions is a class in org.openqa.selenium.interactions.Actions which extends the base class This class has one protected field called action, which is of the CompositeAction type. CompositeAction is a class that accumulates all actions and triggers them at the same time. CompositeAction implements the Action interface. The Action interface has just one method: which the Actions class implements.

Let's have a look at a few of the important methods in the Actions class.

**build():** Builds the sequence of actions to be performed

**click():** Clicks at the last known mouse coordinates

`click(IWebElement)`: Clicks the mouse on the specified element

`clickAndHold()`: Click and hold the mouse button at the last known mouse coordinates

`clickAndHold(IWebElement)`: Click and hold the mouse button down on the element specified

`contextClick()`: Right clicks the mouse at the last known mouse coordinates

`contextClick(IWebElement)`: Right clicks the mouse at the specified element

`doubleClick()`: Double clicks the mouse at the last known mouse coordinates

`doubleClick(IWebElement)`: Double clicks the mouse on the specified element

`dragAndDrop(IWebElement,IWebElement)`: Does drag and drop operation from one element to another

`dragAndDropToOffset(IWebElement,int,int)`: Does a drag and drop operation on one element to a specified offset.

Let's have a look at a small example of the Actions class, which will show the use of these methods.

### A small example to demonstrate the Actions class

We will demonstrate three user actions, which are double click, right click and mouseover.

For this, we will design a page with few elements which will enable us to do the advanced user interactions.

html>

```
id="contextMenu" oncontextmenu="rightClick()">>
```

Right-click in the box to see the message!

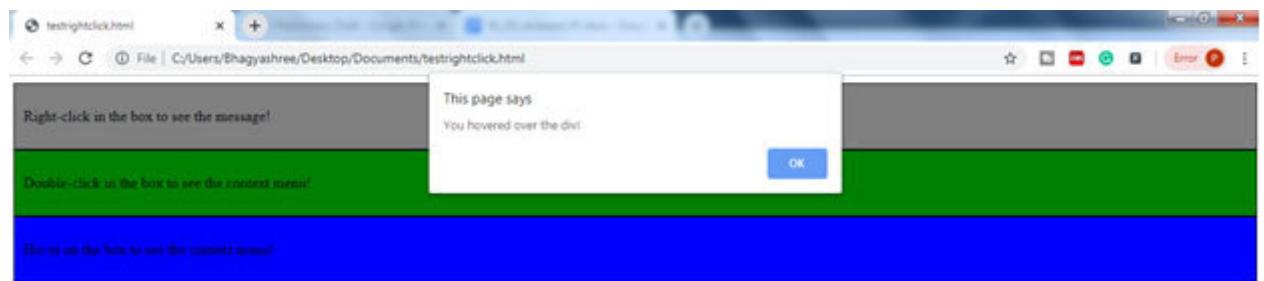
```
id="doubleclickMenu" ondblclick="doubleClick()">>
```

Double-click in the box to see the context menu!

```
id="hoverMenu" onmouseover="hover()">>
```

Hover on the box to see the context menu!

This is how our page looks like when the user hovers over the blue colored div:



**Figure 5.1**

Let's have a look at the prerequisites for running the code, after which we will have a look at the code snippet.

A driver object should be instantiated before adding the code below.

The code fragment given below right clicks on the div with id of

```
//instantiating an Actions object
Actions actions = new Actions(driver);
//instantiating a WebDriverWait object by providing a wait as an
integer number
wdWait = new WebDriverWait(driver, 10);
```

```
//Navigating to the desired page
driver.get("C:\\\\Users\\\\pinakin.chaubal\\\\Documents\\\\testrightclick.html");
//Finding the required webelement
WebElement contextMenu =
driver.findElement(By.id("contextMenu"));
//Performing the Context click
actions.contextClick(contextMenu).perform();
//Waiting till Alert appears
wdWait.until(ExpectedConditions.alertIsPresent());
//Fetching the text in the alert
String text = driver.switchTo().alert().getText();
//Printing the text
System.out.println("The text is: " + text);
```

#### **OUTPUT:**

**The text is: You right-clicked inside the div!**

The code fragment given below double clicks on the div with id of

```
//instantiating an Actions object
actions = new Actions(driver);
//instantiating a WebDriverWait object by providing a wait as an
integer number
wdWait = new WebDriverWait(driver, 10);
//Navigating to the desired page
driver.get("C:\\\\Users\\\\pinakin.chaubal\\\\Documents\\\\testrightclick.html");
//
```

```
//Finding the required webelement
WebElement doubleclickMenu = driver.findElement(By
.id("doubleclickMenu"));
//Performing the Double click
actions.doubleClick(doubleclickMenu).perform();
//Waiting till Alert appears
wdWait.until(ExpectedConditions.alertIsPresent());
//Fetching the text in the alert
String text = driver.switchTo().alert().getText();
//Printing the text
System.out.println("The text is: " + text);
```

#### **OUTPUT:**

**The text is: You double-clicked inside the div!**

The code fragment given below hovers on the div with id of

```
//Create the Actions object
actions = new Actions(driver);
//Instantiate the wait object
wdWait = new WebDriverWait(driver, 10);
//Navigate to URL

driver.get("C:\\\\Users\\\\pinakin.chaubal\\\\Documents\\\\testrightclick.html");
//Fetch WebElement
WebElement hoverMenu = driver.findElement(By.id("hoverMenu"));
//Perform Hover action
actions.moveToElement(hoverMenu).perform();
//Wait for Alert to be present
```

```
wdWait.until(ExpectedConditions.alertIsPresent());  
//Extract the text and print it  
String text = driver.switchTo().alert().getText();  
System.out.println("The text is: " + text);
```

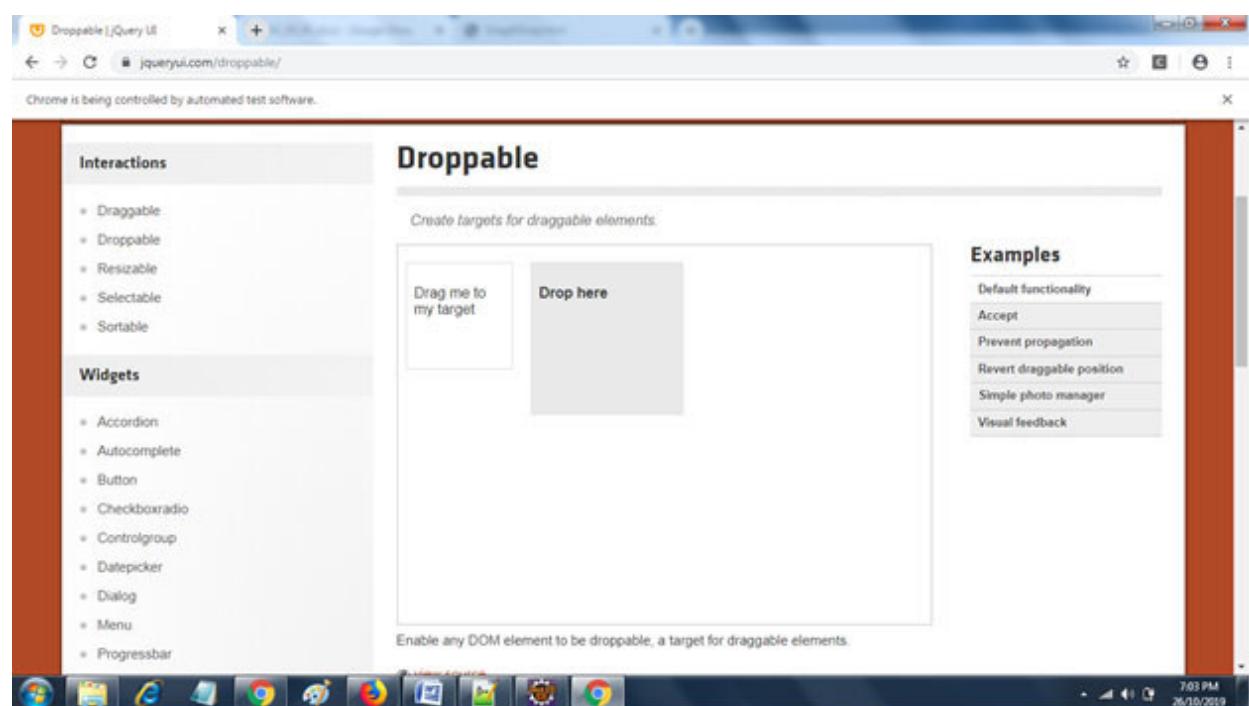
**OUTPUT:**

**The text is: You hovered over the div!**

## Drag and drop example to demonstrate the Actions class

Next, we illustrate the example of drag and drop using Actions class. For this, we use the URL <https://jqueryui.com/droppable/>

The page looks as shown below:



**Figure 5.2**

Here our code will drag the box labeled **Drag me to my target** to the box labeled **Drop**

Prerequisites for running code are as follows.

A driver object should be instantiated.

The code for drag and drop is shown below:

```
//instantiating an Actions object
Actions actions = new Actions(driver);
//instantiating a WebDriverWait object by providing a wait as an
integer number
WebDriverWait wdWait = new WebDriverWait(driver, 10);
//Navigating to the desired page
driver.get("https://jqueryui.com/droppable/");
//Switch to frame

driver.switchTo().frame(
    driver.findElement(By.xpath("//iframe[@class='demo-frame']")));
//Wait till element appears
wdWait.until(ExpectedConditions.elementToBeClickable(driver
.findElement(By.id("draggable"))));
//Search for webelements
WebElement source = driver.findElement(By.id("draggable"));
WebElement target = driver.findElement(By.id("droppable"));
WebElement source1 = target.findElement(By.tagName("p"));
//Perform drag and drop
actions.dragAndDrop(source, target).perform();
//Print the innerHTML
System.out.println("InnerHTML for the target is: " +
source1.getText());
```

**OUTPUT:**

**InnerHTML for the target is: Dropped!**

## Pressing a key combination example

Sometimes we might need to press a key combination, for example, click a link while holding down the Ctrl key to open the link in a new tab. The code shown below searches for Selenium on www.google .com and then opens the first link in a new tab:

```
//Instantiate an Actions object
Actions actions = new Actions(driver);
//Instantiating a WebDriverWait object
WebDriverWait wdWait = new WebDriverWait(driver, 10);
//Navigating to WebSite
driver.get("https://www.google.com");
//Waiting till element appears and provide search text
wdWait.until(
```

```
ExpectedConditions.elementToBeClickable(driver.findElement(By
.name("q")))).sendKeys("selenium");
//Simulate click of the ENTER key
driver.findElement(By.name("q")).sendKeys(Keys.ENTER);
//Key combination performed through Actions object
actions.keyDown(Keys.CONTROL)
.click(driver.findElement(By.className("LC20lb")))
.keyDown(Keys.CONTROL)
.perform();
```

Finally, we see the last code snippet for the Actions class:

## Click and hold action

It is sometimes required to move certain elements by clicking and holding down the left mouse button while the mouse is dragged. This can be achieved by the click and hold action combined with move by offset. The code shown below will move the draggable square by an offset of 100,100.

```
//Create the Actions object
Actions actions = new Actions(driver);
//Create the WebDriverWait object
WebDriverWait wdWait = new WebDriverWait(driver, 10);
//Navigate to WebSite
driver.get("https://jqueryui.com/draggable/");
//Wait till frame is available and switch to it
wdWait.until(ExpectedConditions.frameToBeAvailableAndSwitchToIt(driver
ver
.findElement(By.className("demo-frame"))));
//Wait for element to be clickable
wdWait.until(ExpectedConditions.elementToBeClickable(driver
.findElement(By.id("draggable"))));
//Drag the object by 100,100
actions.clickAndHold(driver.findElement(By.id("draggable")))
.moveByOffset(100, 100).perform();
```

This brings us to an end of Actions class. We will now move on to JavascriptExecutor class.

## The JavascriptExecutor class

We sometimes face situations where the normal Selenium WebDriver API does not work. In such situations, we should use the JavascriptExecutor interface. There are two methods in this interface listed below:

`executeScript(java.lang.String script, java.lang.Object... args):`  
Executes JavaScript in the context of the currently selected frame or window

`executeAsyncScript(java.lang.String script, java.lang.Object... args):`  
Executes an asynchronous piece of JavaScript in the context of the currently selected frame or window.

We will explore three concepts in

Scrolling

Enabling an element

Setting value in a WebElement

JavaScript code as an argument

Let's learn about each one:

**Scrolling:** It may happen that the element you need to work with is not visible until you scroll down to that particular element. We can accomplish this using JavaScript, as shown below. Here we navigate to the automationpractice website and scroll down till we find the link with the visible text of Selenium Framework. We then click the link:

```
//Create JavascriptExecutor object
JavascriptExecutor jExecutor = (JavascriptExecutor) driver;
//Navigate to website

driver.get("http://automationpractice.com/index.php?");
//Create wait object
WebDriverWait wdWait = new WebDriverWait(driver, 20);
//Find the element
WebElement scrollElem = driver.findElement(By
.xpath("//a[@href='http://www.seleniumframework.com']"));
//Perform the scrolling operation and then click on the element
jExecutor.executeScript("arguments[0].scrollIntoView(true)",scrollElem
);
scrollElem.click();
```

As shown above, we make use of the scrollIntoView method and use the requiredWebElement as the second argument.

**Enabling an element and entering value:** The snippet that we are going to see next will first enable the disabled textbox and then will enter hi in the text box. The small html that we will be working with is shown below:

html>

```
     id="myText" disabled=true>
```

The code to enable the text box and enter value in it is given below:

```
//Create the JavascriptExecutor object
JavascriptExecutor jExecutor = (JavascriptExecutor) driver;
//Fetch the URL
driver.get("file:///C:/Users/Bhagyashree/Desktop/Documents/disabledButton.html");
//Create the WebDriverWait object
WebDriverWait wdWait = new WebDriverWait(driver, 20);
//Enable the button by removing the disabled attribute
jExecutor.executeScript(
"arguments[0].removeAttribute('disabled','disabled')",
driver.findElement(By.id("myText")));
//Wait till element is clickable
wdWait.until(ExpectedConditions.elementToBeClickable(driver
.findElement(By.id("myText"))));
//Enter text
jExecutor
.executeScript("document.getElementById('myText').value='hi'");
```

**JavaScript code as an argument to show a textbox:** Now we will see how we can execute custom JavaScript passing it as an argument to The line shown below has to be added in the HTML

```
 id="myText1" hidden>
```

Code shown below will make this textbox visible. The JavaScript is passed as a string argument to the executeScript method:

```
String query =  
"document.getElementById('myText1').style.display='block'";  
jExecutor.executeScript(query);
```

Finally, we create customized classes for Actions and JavascriptExecutor which can be integrated with the framework.

## [Creating customized classes for Actions and JavascriptExecutor](#)

Now that we have an idea of how the Actions class and JavascriptExecutor works, we will create custom classes for both and integrating with the framework

Let's see the class for ActionsExecutor first:

```
package keywordframework;

import org.openqa.selenium.Keys;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.interactions.Actions;

public class ActionsExecutor {
    // constructor
    public ActionsExecutor() {
    }
    //Right click on an element
    public static void rightClickElement(WebElement elem) {
        Actions act = new Actions(SingletonDriver.getInstance().getDriver());
        act.contextClick(elem).perform();
    }
    //Double click on an element
    public static void doubleClickElement(WebElement elem) {
        Actions act = new Actions(SingletonDriver.getInstance().getDriver());
        act.doubleClick(elem).perform();
    }
}
```

```
}

//Drag and Drop on an element
public static void dragAndDrop(WebElement source, WebElement target) {

Actions act = new Actions(SingletonDriver.getInstance().getDriver());
act.dragAndDrop(source, target).perform();
}

//Hover on an element
public static void mouseOver(WebElement elem) {
Actions act = new Actions(SingletonDriver.getInstance().getDriver());
act.moveToElement(elem).perform();
}

//Keydown on an element
public static void keyDown(Keys key) {
Actions act = new Actions(SingletonDriver.getInstance().getDriver());
act.keyDown(key).perform();
}

//KeyUp on an element
public static void keyUp(Keys key) {
Actions act = new Actions(SingletonDriver.getInstance().getDriver());
act.keyUp(key).perform();
}

//Click and hold and move an element
public static void clickAndHoldAndMove(WebElement target, int xOff, int yOff) {
Actions act = new Actions(SingletonDriver.getInstance().getDriver());
act.clickAndHold(target).moveByOffset(xOff, yOff).perform();
}
```

```
}
```

Let's see the class for JavascriptExecutor next:

```
public class JavascriptUtility {  
    /**  
     * Selenium JavaScript Executor Utility Class  
     *  
     */  
    // constructor  
    public JavascriptUtility() {  
    }  
  
    /**  
     * execute - generic method to execute a non-parameterized JS  
     * command  
     *  
     * @param command  
     */  
    public static void execute(String command) {  
        WebDriver driver = SingletonDriver.getInstance().getDriver();  
  
        JavascriptExecutor js = (JavascriptExecutor) driver;  
        js.executeScript(command);  
    }  
  
    /**  
     * execute - overloaded method to execute a JavaScript command  
     * on WebElement  
     *  
     * @param command
```

```
* @param element
*/
public static void execute(String command, WebElement element) {

    WebDriver driver = SingletonDriver.getInstance().getDriver();

    JavascriptExecutorjs = (JavascriptExecutor) driver;
    js.executeScript(command, element);
}

/**
 * click - method to execute a JavaScript click event
 *
 * @param element
 */
public static void click(WebElement element) {
    WebDriver driver = SingletonDriver.getInstance().getDriver();

    JavascriptExecutorjs = (JavascriptExecutor) driver;
    js.executeScript("arguments[0].click();", element);
}

/**
 * click - overloaded method to execute a JavaScript click event
 * using By
 *
 * @param by
 */
public static void click(By by) {
```

```
WebDriver driver = SingletonDriver.getInstance().getDriver();
WebElement element = driver.findElement(by);

JavascriptExecutorjs = (JavascriptExecutor) driver;
js.executeScript("arguments[0].click();", element);
}

/**
 * sendKeys - method to execute a JavaScript value event

*
* @param keys
* @param element
*/
public static void sendKeys(String value, WebElement element) {

WebDriver driver = SingletonDriver.getInstance().getDriver();

JavascriptExecutorjs = (JavascriptExecutor) driver;
js.executeScript("arguments[0].value='" + value + "';", element);
}

/**
 * isPageLoaded - method to verify that a page has completely
rendered
*
* @param driver
* @return boolean
*/
public static booleanisPageLoaded(WebDriver driver) {
```

```

JavascriptExecutorjs = (JavascriptExecutor) driver;
return (Boolean) js.executeScript("return
document.readyState").equals(
"complete");
}

/**
* isAjaxComplete - method to verify that an ajax control has
rendered
*
* @param driver
* @return boolean

*/
public static boolean isAjaxComplete(WebDriver driver) {
JavascriptExecutorjs = (JavascriptExecutor) driver;
return (Boolean) js.executeScript("return jQuery.active == 0");
}

}

```

We now move on to the final part of this chapter, which is integrating the ActionsExecutor and JSEexecutor in our framework. In this section, the customized wait class has also been shown, which has methods for waiting for the frame to be available and element to be clickable.

## Integrating with the framework

We will start with the changes in FrameworkDriver. The code below resides in a main method. Here the jqueryui.com/draggable website is launched and the draggable object is moved by an offset of 100,100.

```
//Create the ActionKeywords object
ActionKeywords act = new ActionKeywords();
//invoke the browser
act.openBrowser("chrome");
//Create object of the Wait Class
WaitClass wdWait = new WaitClass();
//Navigate to URL
act.navigateURL("https://jqueryui.com/draggable/");
//Switch to frame
wdWait.switchToFrame("//*[@class='demo-frame']");
//Wait for element to be clickable
wdWait.waitForClickable("//*[@id='draggable']");
//Click hold and move element
ActionsExecutor.clickAndHoldAndMove(SingletonDriver.getInstance()
.getDriver().findElement(By.id("draggable")), 100, 100);
If you notice, we have created a class for wait logic called
WaitClass. Let's see the contents of the WaitClass next.
public class WaitClass {
    WebDriverWait explicitWait = null;
    // constructor
```

```
public WaitClass() {  
    explicitWait = new WebDriverWait(SingletonDriver.getInstance()  
.getDriver(), 30);
```

```
}
```

```
//Switching to frame
```

```
public void switchToFrame(String xpath) {  
    explicitWait.until(ExpectedConditions  
.frameToBeAvailableAndSwitchToIt(SingletonDriver.getInstance()  
.getDriver().findElement(By.xpath(xpath))));
```

```
}
```

```
//Wait for element to be clickable
```

```
public void waitForClickable(String xpath) {  
    explicitWait.until(ExpectedConditions  
.elementToBeClickable(SingletonDriver.getInstance().getDriver()  
.findElement(By.xpath(xpath))));  
}
```

```
}
```

Right now, there are only two methods, which are `switchToFrame()` and as shown in the code snippet above for We will be adding more methods as we move on.

This brings us to the end of this chapter.

## Conclusion

This chapter covered important aspects with regards to the Actions class and the We saw the important methods in both of the classes and also saw snippets of code that were used to demonstrate important aspects of both the classes. We also added a class for Wait Logic with two methods. As we move on, more methods will be added in the classes ActionsExecutor and All the principles covered in this chapter are very important from the standpoint of creating a framework from scratch, and we will expand on these concepts as we move on.

The next chapter will be about Event handling in Selenium WebDriver. Event handling is accomplished by using the In this chapter, we will also touch upon reporting using Extent Reports, logging using Log4J, and how this can be accomplished when a certain event occurs like button click. We will be creating a new class to handle Events in Selenium WebDriver.

## Questions

Write a program to open any one menu using moveToElement method of Actions class

List down five advantages of using the Actions class

Write a program to enable a disabled dropdown

List down five advantages of JavascriptExecutor

Try changing the Framework driver to invoke JavascriptUtils class methods.

## CHAPTER 6

### WebDriver Events

The Selenium WebDriver API provides us with the facility to know when the test scripts trigger a particular event like navigating to a URL or clicking a button. There are events that get triggered before and after a WebDriver internal event occurs. Capturing these events prove to be a great help while analyzing issues with the Application under test.

We will see how to trigger events and how to listen to the triggered events. We will understand in detail about the classes and interfaces used for triggering events and listening to those events. We will have a look at navigation events, element search, button click, and other related events. We will also study the logging process using the Log4J API.

This chapter will be an important one as far as learning the event handling process along with logging useful information to the console and a file. After completing this chapter, the reader will be well acquainted with the two ways of event handling and Logging, which will be very important in pinpointing errors in the application or debugging the test automation scripts.

## Structure

Here's what we will learn in this chapter:

What are Selenium WebDriver Events and their need

Learn the process of handling events

Understand how the EventFiringWebDriver and  
WebDriverEventListener work

Registering one and multiple listeners with the  
EventFiringWebDriver

Unregistering the listener

Understand the Log4J API

Looking at the different methods in the WebDriverEventListener  
interface

Integrating the new classes with the framework

## Objectives

Understanding events in Selenium WebDriver

Learning the process of event handling in Selenium WebDriver

Learn about the classes and interfaces involved in the event handling process

Learning about the Log4J API

Understand the advantages of using AbstractWebDriverEventListener class over the WebDriverEventListener interface

Know the functionality of each method in the WebDriverEventListener interface

Need for unregistering the listeners from the EventFiringWebDriver

Integration with the framework

We begin by first understanding what events mean in Selenium WebDriver.

## What are the events in Selenium WebDriver?

Test scripts perform actions on a web page by invoking internal WebDriver events like click or navigate to URL. These internal WebDriver events help us to achieve the process of automating any web page. Events help us in finding out if there are problems in the Application under test. Events assist the application developer in analyzing where a particular problem was encountered during the script execution

Selenium webdriver provides us the ability to track different events such as beforeNavigateTo, afterNavigateTo, beforeFindBy, afterFindBy, beforeClickOn, afterClickOn, and so on. Whenever test scripts are developed, custom implementation can be provided for these methods in order to analyze results.

Let's now have a glance at the classes and interfaces involved with event handling.

## [\*Introducing the EventFiringWebDriver class\*](#)

EventFiringWebDriver class acts as a wrapper around WebDriver that provides the driver instance with the ability to trigger events. The EventFiringWebDriver has many methods out of which we are right now interested in only two methods which are listed below:

public EventFiringWebDriver register(WebDriverEventListener): This method registers the event listener to the A single argument of type WebDriverEventListener is passed to the register method.

public EventFiringWebDriver unregister(WebDriverEventListener): This method unregisters the event listener from the A single argument of type WebDriverEventListener is passed to the register method.

We will see working examples of these methods when we get to the examples.

## [Introducing the WebDriverEventListener interface](#)

When events get triggered by the these events should be listened to, and appropriate logs or System out messages should be generated. This duty of listening to events is done by implementing the WebDriverEventListener interface. The implemented listener class waits and listens to the EventFiringWebDriver and handles all the events that get triggered by it. There can be more than one listeners that might wait for an event to trigger. To facilitate the listening process, all listeners should be registered with the EventFiringWebDriver in order to get notifications from time to time.

The WebDriverEventListener interface has 27 methods in all. We will be adding System out messages for each of these methods.

The interface is shown below:

```
public interface WebDriverEventListener {  
    void beforeAlertAccept(WebDriver driver);  
    void afterAlertAccept(WebDriver driver);  
    void afterAlertDismiss(WebDriver driver);  
    void beforeAlertDismiss(WebDriver driver);  
    void beforeNavigateTo(String url, WebDriver driver);  
    void afterNavigateTo(String url, WebDriver driver);  
    void beforeNavigateBack(WebDriver driver);  
    void afterNavigateBack(WebDriver driver);
```

```
void beforeNavigateForward(WebDriver driver);
void afterNavigateForward(WebDriver driver);
void beforeNavigateRefresh(WebDriver driver);
void afterNavigateRefresh(WebDriver driver);
void beforeFindBy(By by, WebElement element, WebDriver driver);

void afterFindBy(By by, WebElement element, WebDriver driver);
void beforeClickOn(WebElement element, WebDriver driver);
void afterClickOn(WebElement element, WebDriver driver);
void beforeChangeValueOf(WebElement element, WebDriver driver,
CharSequence[] keysToSend);
void afterChangeValueOf(WebElement element, WebDriver driver,
CharSequence[] keysToSend);
void beforeScript(String script, WebDriver driver);
void afterScript(String script, WebDriver driver);
void beforeSwitchToWindow(String windowName, WebDriver driver);
void afterSwitchToWindow(String windowName, WebDriver driver);
void onException(Throwable throwable, WebDriver driver);
void beforeGetScreenshotAs(OutputType target);
void afterGetScreenshotAs(OutputType target, X screenshot);
void beforeGetText(WebElement element, WebDriver driver);
void afterGetText(WebElement element, WebDriver driver, String
text);
}
```

Next let's get introduced to the `AbstractWebDriverEventListener` class.

### [\*Introducing the AbstractWebDriverEventListener class\*](#)

Notice the size of the WebDriverEventListener interface. If our custom EventListener class directly implements this interface, then we have to either provide implementations for each unimplemented method or provide dummy implementations for each method, which is an overhead. Selenium has simplified work for us and introduced the Abstract class which provides dummy implementations for all methods in the WebDriverEventListener interface. If the size of the code is a constraint, then one can extend the AbstractWebDriverEventListener while creating their custom Listener.

Now that we know about the basic classes and interfaces involved in event handling let's look at the process of event handling next.

## Process of event handling

Let's have a look at the steps that need to be followed, so that event listening happens for all the events triggered by the EventFiringWebDriver:

Create a custom EventListener class either by implementing the WebDriverEventListener interface or extending the

Fetch the instance of WebDriver using SingletonWebDriver class's getInstance() method chained with the getDriver() method. Prior to getting the driver, we set the driver using the setDriver() method.

Create an instance of EventFiringWebDriver wrapping the WebDriver instance using the code shown below

```
EventFiringWebDriver eventDriver = new EventFiring  
WebDriver(SingletonDriver.getInstance().getDriver());
```

Create an instance of the EventListener class and register it to listen to the events fired by the EventFiringWebDriver object created in the prior step

```
MyEventListener eventListener = new EventListener();  
eventDriver.register(handler);
```

Finally, the instance of the MyEventListener should be unregistered from the listening to events using the code shown below

```
eventDriver.unRegister(eventListener);
```

We will now take a look at two ways of creating the custom event listener, The first way is to create the custom event listener by implementing the WebDriverEventListener interface, as shown below.

### Implementing the WebDriverEventListener interface

The first way to use event listeners is to implement the WebDriverEventListener interface, as shown below.

```
package bpb.events;

import org.openqa.selenium.By;
import org.openqa.selenium.OutputType;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.support.events.WebDriverEventListener;

public class MyEventListener implements WebDriverEventListener{

    @Override
    public void beforeAlertAccept(WebDriver driver) {
```

```
System.out.println("Before Alert Accept: "+driver.getCurrentUrl());  
}  
  
@Override  
public void afterAlertAccept(WebDriver driver) {  
System.out.println("After Alert Accept: "+driver.getCurrentUrl());  
}  
  
@Override  
public void afterAlertDismiss(WebDriver driver) {  
System.out.println("After Alert Dismiss: "+driver.getCurrentUrl());  
}  
  
@Override  
public void beforeAlertDismiss(WebDriver driver) {  
System.out.println("Before Alert Dismiss: "+driver.getCurrentUrl());  
}  
  
@Override  
public void beforeNavigateTo(String url, WebDriver driver) {  
System.out.println("Navigating to "+url+" for URL:  
"+driver.getCurrentUrl());  
}
```

```
@Override  
public void afterNavigateTo(String url, WebDriver driver) {  
    System.out.println("Navigated to "+url+" for URL:  
    "+driver.getCurrentUrl());  
  
}  
  
@Override  
public void beforeNavigateBack(WebDriver driver) {  
    System.out.println("Navigating back from "+driver.getCurrentUrl());  
  
}  
  
@Override  
public void afterNavigateBack(WebDriver driver) {  
    System.out.println("Navigated back from "+driver.getCurrentUrl());  
  
}  
  
@Override  
public void beforeNavigateForward(WebDriver driver) {  
    System.out.println("Navigating forward from  
    "+driver.getCurrentUrl());  
  
}  
  
@Override
```

```
public void afterNavigateForward(WebDriver driver) {  
    System.out.println("Navigated foward from "+driver.getCurrentUrl());  
}
```

```
@Override  
public void beforeNavigateRefresh(WebDriver driver) {  
    System.out.println("Before Navigate Refresh "+driver.getCurrentUrl());  
}
```

```
@Override  
public void afterNavigateRefresh(WebDriver driver) {  
    System.out.println("After Navigate Refresh "+driver.getCurrentUrl());  
}
```

```
@Override  
public void beforeFindBy(By by, WebElement element, WebDriver  
driver) {  
    System.out.println("Finding element by XPATH"+by.toString());  
}
```

```
@Override  
public void afterFindBy(By by, WebElement element, WebDriver  
driver) {  
    System.out.println("Found "+element.toString());
```

```
}
```

```
@Override  
public void beforeClickOn(WebElement element, WebDriver driver) {  
    System.out.println("Before clicking "+element.toString());
```

```
}
```

```
@Override  
public void afterClickOn(WebElement element, WebDriver driver) {  
    System.out.println("After clicking "+element.toString());
```

```
}
```

```
@Override  
public void beforeChangeValueOf(WebElement element, WebDriver  
    driver, CharSequence[] keysToSend) {  
    System.out.println("Before changing value of "+element.toString());
```

```
}
```

```
@Override  
public void afterChangeValueOf(WebElement element, WebDriver  
    driver, CharSequence[] keysToSend) {  
    System.out.println("After changing value of "+element.toString());
```

```
}
```

```
@Override  
public void beforeScript(String script, WebDriver driver) {  
    System.out.println("Before script "+script);  
  
}
```

```
@Override  
public void afterScript(String script, WebDriver driver) {  
    System.out.println("After script "+script);  
  
}
```

```
@Override  
public void beforeSwitchToWindow(String windowName, WebDriver  
driver) {  
    System.out.println("Before switching "+windowName);  
  
}
```

```
@Override  
public void afterSwitchToWindow(String windowName, WebDriver  
driver) {  
    System.out.println("After switching "+windowName);  
  
}
```

```
@Override  
public void onException(Throwable throwable, WebDriver driver) {
```

```
System.out.println("Exception thrown "+throwable.getMessage());  
}
```

```
@Override  
public void beforeGetScreenshotAs(OutputType target) {  
System.out.println("Before taking screenshot "+target.toString());  
}  
}
```

```
@Override  
public void afterGetScreenshotAs(OutputType target, X screenshot)  
{  
System.out.println("After taking screenshot "+target.toString());  
}  
}
```

```
@Override  
public void beforeGetText(WebElement element, WebDriver driver) {  
System.out.println("Before fetching text "+element.toString());  
}  
}
```

```
@Override  
public void afterGetText(WebElement element, WebDriver driver,  
String text) {  
System.out.println("After fetching text "+element.toString());
```

```
}
```

```
}
```

As can be seen, all methods in the WebDriverEventListener either need to be implemented or provided with dummy implementations. To run this code, we create a test class. In this class, instead of using the driver instance, we have used the EventFiringWebDriver instance wrapped over the WebDriver instance so that the events get printed in the console.

```
public class ListenerTest {  
  
    public static void main(String[] args) {  
  
        try {  
            SingletonDriver.getInstance().setDriver("chrome","Windows","Windows");  
        } catch (Exception e) {  
            // TODO Auto-generated catch block  
            e.printStackTrace();  
        }  
        WebDriver driver = SingletonDriver.getInstance().getDriver();  
        WaitClass wdWait = new WaitClass();  
        EventFiringWebDriver event = new EventFiringWebDriver(driver);  
        MyEventListener listen = new MyEventListener();  
        event.register(listen);  
        event.get("http://www.freecrm.com");  
        event.findElement(By.xpath("//*[  
[@href='https://ui.freecrm.com']]")).click();  
    }  
}
```

```
wdWait.waitForClickable("//div[text()='Login']");
event.findElement(By.xpath("//div[text()='Login']")).getText();
event.unregister(listen);
}
}
```

#### OUTPUT:

```
Navigating to http://www.freecrm.com for URL: data:,  
Navigated to http://www.freecrm.com for URL: https://freecrm.com/  
Finding element by XPATHBy.xpath: //*[  
[@href='https://ui.freecrm.com']]  
Found [[ChromeDriver: chrome on XP  
(b83a1oed5ec71cb1f81a3e8dd146da48)] ->xpath: //*[  
[@href='https://ui.freecrm.com']]  
Before clicking [[ChromeDriver: chrome on XP  
(b83a1oed5ec71cb1f81a3e8dd146da48)] ->xpath: //*[  
[@href='https://ui.freecrm.com']]  
After clicking [[ChromeDriver: chrome on XP  
(b83a1oed5ec71cb1f81a3e8dd146da48)] ->xpath: //*[  
[@href='https://ui.freecrm.com']]  
Finding element by XPATHBy.xpath: //div[text()='Login']  
Found [[ChromeDriver: chrome on XP  
(b83a1oed5ec71cb1f81a3e8dd146da48)] ->xpath: //div[text()='Login']]  
Before fetching text [[ChromeDriver: chrome on XP  
(b83a1oed5ec71cb1f81a3e8dd146da48)] ->xpath: //div[text()='Login']]  
After fetching text [[ChromeDriver: chrome on XP  
(b83a1oed5ec71cb1f81a3e8dd146da48)] ->xpath: //div[text()='Login']]
```

As can be seen all actions get logged into the console.

## Extending the AbstractWebDriverEvent Listener abstract class

It can be seen from the MyEventListener class that all methods in the WebDriverEventListener interface have to be implemented or provided with dummy implementations. This makes the code lengthy and redundant. A better alternative is to use the AbstractWebDriverEventListener class, which already implements the WebDriverEventListener interface by providing dummy implementations for all methods in the WebDriverEventListener interface.

Let's see how we can use the

Create a class called

```
public class MyEventListener2 extends  
AbstractWebDriverEventListener {  
    @Override  
    public void beforeNavigateTo(String url, WebDriver driver) {  
        System.out.println("Navigating to "+url+" for URL:  
        "+driver.getCurrentUrl());  
    }  
}
```

```
@Override  
public void afterNavigateTo(String url, WebDriver driver) {  
    System.out.println("Navigated to "+url+" for URL:  
    "+driver.getCurrentUrl());  
}
```

```
}
```

```
@Override  
public void beforeFindBy(By by, WebElement element, WebDriver  
driver) {  
System.out.println("Finding element by XPATH"+by.toString());  
}
```

```
@Override  
public void afterFindBy(By by, WebElement element, WebDriver  
driver) {  
System.out.println("Found "+element.toString());  
}
```

```
@Override  
public void beforeClickOn(WebElement element, WebDriver driver) {  
System.out.println("Before clicking "+element.toString());  
}
```

```
@Override  
public void afterClickOn(WebElement element, WebDriver driver) {  
System.out.println("After clicking "+element.toString());  
}
```

```
@Override  
public void beforeGetText(WebElement element, WebDriver driver) {  
System.out.println("Before fetching text "+element.toString());  
}
```

```
@Override  
public void afterGetText(WebElement element, WebDriver driver,  
String text) {  
    System.out.println("After fetching text "+text);  
  
}  
}
```

Now change only the line shown below in the test class. Notice that now we are creating object of

```
MyEventListener2 listen = new MyEventListener2();
```

#### OUTPUT:

```
Navigating to http://www.freecrm.com for URL: data:  
Navigated to http://www.freecrm.com for URL: https://freecrm.com/  
Finding element by XPATHBy.xpath: //*[  
[@href='https://ui.freecrm.com']]  
Found [[ChromeDriver: chrome on XP  
(7e3ce7dda64cf9a71065ff6cd1670453)] ->xpath: //*[  
[@href='https://ui.freecrm.com']]  
Before clicking [[ChromeDriver: chrome on XP  
(7e3ce7dda64cf9a71065ff6cd1670453)] ->xpath: //*[  
[@href='https://ui.freecrm.com']]  
After clicking [[ChromeDriver: chrome on XP  
(7e3ce7dda64cf9a71065ff6cd1670453)] ->xpath: //*[  
[@href='https://ui.freecrm.com']]  
Finding element by XPATHBy.xpath: //div[text()='Login']  
Found [[ChromeDriver: chrome on XP  
(7e3ce7dda64cf9a71065ff6cd1670453)] ->xpath: //div[text()='Login']]
```

Before fetching text [[ChromeDriver: chrome on XP  
(7e3ce7dda64cf9a71065ff6cd1670453)] ->xpath: //div[text()='Login']]  
After fetching text Login

As can be seen, we have extracted the text of the **Login** button.

## Registering more than one listener to the EventFiringWebDriver instance

In the earlier examples, we registered a single listener with the EventFiringWebDriver object using the register() method. Now we will register two listeners with the EventFiringWebDriver object.

The changes required in the test class are shown below. In the code below, we have registered objects of MyEventListener and MyEventListener2 with the EventListener object, and after the purpose, if fulfilled, both the listeners are unregistered.

```
try {
    SingletonDriver.getInstance().setDriver("chrome","Windows","Windows");
} catch (Exception e) {
    // TODO Auto-generated catch block
    e.printStackTrace();
}
WebDriver driver = SingletonDriver.getInstance().getDriver();
WaitClasswdWait = new WaitClass();
EventFiringWebDriver event = new EventFiringWebDriver(driver);
MyEventListener listen = new MyEventListener();
MyEventListener2 listen2 = new MyEventListener2();
event.register(listen);
event.register(listen2);
event.get("http://www.freecrm.com");
event.findElement(By.xpath("//*
[@href='https://ui.freecrm.com']")).click();
```

```
wdWait.waitForClickable("//div[text()='Login']");
event.findElement(By.xpath("//div[text()='Login']")).getText();
event.unregister(listen);
event.unregister(listen2);
```

**OUTPUT:**

```
Navigating to http://www.freecrm.com for URL: data:,  
MyEventListener2 Navigating to http://www.freecrm.com for URL:  
data:,  
Navigated to http://www.freecrm.com for URL: https://freecrm.com/  
MyEventListener2 Navigated to http://www.freecrm.com for URL:  
https://freecrm.com/  
Finding element by XPATHBy.xpath: //*[  
[@href='https://ui.freecrm.com']]  
MyEventListener2 Finding element by XPATHBy.xpath: //*[  
[@href='https://ui.freecrm.com']]  
Found [[ChromeDriver: chrome on XP  
(9d3e1fb027899faa756738c7c2d9c01)] ->xpath: //*[  
[@href='https://ui.freecrm.com']]  
MyEventListener2 Found [[ChromeDriver: chrome on XP  
(9d3e1fb027899faa756738c7c2d9c01)] ->xpath: //*[  
[@href='https://ui.freecrm.com']]  
Before clicking [[ChromeDriver: chrome on XP  
(9d3e1fb027899faa756738c7c2d9c01)] ->xpath: //*[  
[@href='https://ui.freecrm.com']]  
MyEventListener2 Before clicking [[ChromeDriver: chrome on XP  
(9d3e1fb027899faa756738c7c2d9c01)] ->xpath: //*[  
[@href='https://ui.freecrm.com']]
```

After clicking [[ChromeDriver: chrome on XP  
(9d3e1fb027899faa756738c7c2d9c01)] ->xpath: //\*[  
[@href='https://ui.freecrm.com']]

MyEventListener2 After clicking [[ChromeDriver: chrome on XP  
(9d3e1fb027899faa756738c7c2d9c01)] ->xpath: //\*[  
[@href='https://ui.freecrm.com']]

Finding element by XPATHBy.xpath: //div[text()='Login']

MyEventListener2 Finding element by XPATHBy.xpath:  
//div[text()='Login']

Found [[ChromeDriver: chrome on XP  
(9d3e1fb027899faa756738c7c2d9c01)] ->xpath: //div[text()='Login']]

MyEventListener2 Found [[ChromeDriver: chrome on XP  
(9d3e1fb027899faa756738c7c2d9c01)] ->xpath: //div[text()='Login']]

Before fetching text [[ChromeDriver: chrome on XP  
(9d3e1fb027899faa756738c7c2d9c01)] ->xpath: //div[text()='Login']]

MyEventListener2 Before fetching text [[ChromeDriver: chrome on  
XP (9d3e1fb027899faa756738c7c2d9c01)] ->xpath:  
//div[text()='Login']]

After fetching text [[ChromeDriver: chrome on XP  
(9d3e1fb027899faa756738c7c2d9c01)] ->xpath: //div[text()='Login']]

MyEventListener2 After fetching text Login

The **MyEventListener2** class was changed for the system.out statements in order to differentiate the outputs from both listeners.

## Introducing the Log4J framework

Log4J is a logging framework that is used for generating logs during test execution. Log4J is configurable through external configuration files. In order to use Log4J, the log4j dependency should be placed in the pom.xml, as shown below:

```
log4j  
log4j  
1.2.17
```

After putting this dependency, right click project **Maven | Update**. The required JAR files will be downloaded in the folder Maven

Log4J has three main components listed below:

**Loggers:** Used for capturing logging information

**Appenders:** Used for redirecting logging information to various preferred destinations

**Layouts:** Used for formatting logging information in different styles

Let's understand the steps involved in using Log4J to log user-defined output statements in code.

## Steps to include Log4J in code

The steps involved in using Log4J to log user-defined output statements are shown below:

Create a Log4J.properties file using the content shown below:

```
#Define the root logger. Level of Root logger is defined as
DEBUG
log4j.rootLogger=DEBUG, CONSOLE, LOGFILE

#Define the CONSOLE Appender, the threshold, the layout and
the conversion #pattern
# The logging levels are in the order ALL(Integer.MAX_VALUE) <
#DEBUG(500) < #INFO(400) < WARN(300) < ERROR(200) <
FATAL(100) < OFF(0). Since we #have kept the logging #level #
at INFO here, all INFO,WARN,ERROR,FATAL messages #will be
displayed. The DEBUG and #TRACE level messages are not
displayed
log4j.appender.CONSOLE=org.apache.log4j.ConsoleAppender
log4j.appender.CONSOLE.Threshold=INFO
log4j.appender.CONSOLE.layout=org.apache.log4j.PatternLayout
log4j.appender.CONSOLE.layout.ConversionPattern=%d{yyyy-MM-dd
HH:mm:ss} %-5p %C{1}:%L - %m%n
# Define the File Appender
log4j.appender.LOGFILE=org.apache.log4j.RollingFileAppender
log4j.appender.LOGFILE.File=D:\\logging.log
```

```
log4j.appender.LOGFILE.MaxFileSize=20MB  
log4j.appender.LOGFILE.MaxBackupIndex=5  
log4j.appender.LOGFILE.Append=true  
log4j.appender.LOGFILE.Threshold=INFO
```

The log messages will be pushed to the D:\logging.log file, which will be appended. On reaching 20 MB, a new file gets created. The new file will be an archived version, and at the place where it shows the type, the version number will be listed. For example, the first file will be of type 1, the second of type 2. When the version 2 file is created, the version 1 file data will be pushed to version 2, and the current version data will be pushed to version 1. The next thing worth mentioning is that there can be only five versions that can be created, as indicated by MaxBackupIndex. After the fifth version, once maxFileSize is reached, data will start flowing from the current version to version 1, from version 1 to version 2, version 2 to version 3, and so on. Data in version 5 that was originally there will be overwritten by data from version 4.

Perform the changes mentioned below for our listeners to work with Log4J.

Replace all statements in the MyEventListener2.java with log.info statements, as shown below:

```
@Override  
public void beforeNavigateTo(String url, WebDriver driver) {  
    log.info("MyEventListener2 Navigating to "+url);
```

```
}
```

```
@Override
```

```
public void afterNavigateTo(String url, WebDriver driver) {  
    log.info("MyEventListener2 Navigated to "+url+" for URL:  
    "+driver.getCurrentUrl());  
}
```

```
@Override
```

```
public void beforeFindBy(By by, WebElement element, WebDriver  
driver) {  
    log.info("MyEventListener2 Finding element by  
    XPATH"+by.toString());  
}
```

```
@Override
```

```
public void afterFindBy(By by, WebElement element, WebDriver  
driver) {  
    log.info("MyEventListener2 Found "+element.toString());  
}
```

```
@Override
```

```
public void beforeClickOn(WebElement element, WebDriver driver) {  
    log.info("MyEventListener2 Before clicking "+element.toString());  
}
```

```
@Override
```

```
public void afterClickOn(WebElement element, WebDriver driver) {  
  
    log.info("MyEventListener2 After clicking "+element.toString());  
}
```

```
}
```

```
@Override  
public void beforeGetText(WebElement element, WebDriver driver) {  
log.info("MyEventListener2 Before fetching text  
"+element.toString());  
}
```

```
@Override  
public void afterGetText(WebElement element, WebDriver driver,  
String text) {  
log.info("MyEventListener2 After fetching text "+text);  
}
```

Change the as shown below:

```
try {  
SingletonDriver.getInstance().setDriver("chrome","Windows","Windows  
");  
} catch (Exception e) {  
// TODO Auto-generated catch block  
e.printStackTrace();  
}  
WebDriver driver = SingletonDriver.getInstance().getDriver();  
WaitClass wdWait = new WaitClass();  
EventFiringWebDriver event = new EventFiringWebDriver(driver);  
MyEventListener2 listen2 = new MyEventListener2();  
event.register(listen2);  
  
event.get("http://www.freecrm.com");  
event.findElement(By.xpath("//*[@href='https://ui.freecrm.com']]"))
```

```
.click();
wdWait.waitForClickable("//div[text()='Login']");
event.findElement(By.xpath("//div[text()='Login']")).getText();
event.unregister(listen2);
```

Make the changes mentioned below in First of all instantiate a Logger object:

```
Logger log = Logger.getLogger(MyEventListener2.class);
```

Add a constructor and indicate the properties file to be used for Log4J:

```
public MyEventListener2() {
super();
PropertyConfigurator.configure("log4j.properties");
}
```

Run the ListenerTest.java as a Java application

#### OUTPUT:

```
2019-11-09 09:04:26 INFO MyEventListener2:21 - MyEventListener2
Navigating to http://www.freecrm.com
```

```
2019-11-09 09:04:34 INFO MyEventListener2:27 - MyEventListener2
Navigated to http://www.freecrm.com for URL: https://freecrm.com/
```

```
2019-11-09 09:04:34 INFO MyEventListener2:33 - MyEventListener2
Finding element by XPATHBy.xpath: //*
[@href='https://ui.freecrm.com']
```

```
2019-11-09 09:04:34 INFO MyEventListener2:39 - MyEventListener2
Found [[ChromeDriver: chrome on XP
(9781575735fdeaaaa1ae88fdfe25c5ad)] ->xpath: //*
[@href='https://ui.freecrm.com']]]

2019-11-09 09:04:34 INFO MyEventListener2:45 - MyEventListener2
Before clicking [[ChromeDriver: chrome on XP
(9781575735fdeaaaa1ae88fdfe25c5ad)] ->xpath: //*
[@href='https://ui.freecrm.com']]]

2019-11-09 09:04:47 INFO MyEventListener2:51 - MyEventListener2
After clicking [[ChromeDriver: chrome on XP
(9781575735fdeaaaa1ae88fdfe25c5ad)] ->xpath: //*
[@href='https://ui.freecrm.com']]]

2019-11-09 09:04:47 INFO MyEventListener2:33 - MyEventListener2
Finding element by XPATHBy.xpath: //div[text()='Login']

2019-11-09 09:04:47 INFO MyEventListener2:39 - MyEventListener2
Found [[ChromeDriver: chrome on XP
(9781575735fdeaaaa1ae88fdfe25c5ad)] ->xpath: //div[text()='Login']]]

2019-11-09 09:04:47 INFO MyEventListener2:56 - MyEventListener2
Before fetching text [[ChromeDriver: chrome on XP
(9781575735fdeaaaa1ae88fdfe25c5ad)] ->xpath: //div[text()='Login']]]

2019-11-09 09:04:47 INFO MyEventListener2:63 - MyEventListener2
After fetching text Login
```

Let's integrate event handling with the framework.

## Integrating with the framework

We come to the final part of this chapter where we integrate the event firing and handling into our framework. We start with

```
ActionKeywords act = new ActionKeywords();
act.navigateURL("http://www.freecrm.com");
WaitClass wdWait = new WaitClass();
wdWait.waitForClickable("//*[@href='https://ui.freecrm.com']");
WebElement elem = SingletonDriver.getInstance().getDriver()
act.clickElement("//*[@href='https://ui.freecrm.com']");
```

Here we create an object of ActionKeywords class. The corresponding methods for navigating to URL and clicking on an element are then invoked.

We look at the global declarations next:

```
EventFiringWebDriver event = null;
WebDriver driver = null;
MyEventListener2 list = new MyEventListener2();
```

Let's have a look at the ActionKeywords constructor next:

```
public ActionKeywords() {
try {
```

```
SingletonDriver.getInstance().setDriver("chrome", "windows",
"windows");
driver = SingletonDriver.getInstance().getDriver();
event = new EventFiringWebDriver(driver);
event.register(list);
event.manage().window().maximize();
} catch (Exception e) {
e.printStackTrace();

}
}
```

This method creates the EventFiringWebDriver from the chrome instance obtained from the SingletonDriver class.

Next comes the navigateToURL() method:

```
public void navigateURL(String URL) {
event.get(URL);
}
```

Next we take a look at the clickElement() method:

```
public void clickElement(String xpath) {
event.findElement(By.xpath(xpath)).click();
}
```

Let's have a look at the output generated.

## OUTPUT:

2019-11-09 10:50:40 INFO MyEventListener2:21 - MyEventListener2

Navigating to <http://www.freecrm.com>

2019-11-09 10:50:47 INFO MyEventListener2:27 - MyEventListener2

Navigated to <http://www.freecrm.com> for URL: <https://freecrm.com/>

2019-11-09 10:50:47 INFO MyEventListener2:33 - MyEventListener2

Finding element by XPATHBy.xpath: //\*

[@href='https://ui.freecrm.com']

2019-11-09 10:50:47 INFO MyEventListener2:39 - MyEventListener2

Found [[ChromeDriver: chrome on XP

(c67f7cc1d26cd766b68481236540b1ae)] ->xpath: //\*

[@href='https://ui.freecrm.com']]

2019-11-09 10:50:47 INFO MyEventListener2:45 - MyEventListener2

Before clicking [[ChromeDriver: chrome on XP

(c67f7cc1d26cd766b68481236540b1ae)] ->xpath: //\*

[@href='https://ui.freecrm.com']]

2019-11-09 10:50:56 INFO MyEventListener2:51 - MyEventListener2

After clicking [[ChromeDriver: chrome on XP

(c67f7cc1d26cd766b68481236540b1ae)] ->xpath: //\*

[@href='https://ui.freecrm.com']]

To conclude this chapter, let's list the methods in the WebDriverEventListener interface.

## Different WebDriver event listeners

As a conclusion to this chapter, let's list down the various methods available in the WebDriverEventListener interface.

## WebElement search

Methods shown below are triggered while finding an element:

```
public void beforeFindBy(By by, WebElement element, WebDriver driver): Triggered before finding an element.
```

```
public void afterFindBy(By by, WebElement element, WebDriver driver): Triggered after finding an element.
```

## [WebElement click](#)

Methods shown below are triggered while clicking an element:

public void beforeClickOn(WebElement element, WebDriver driver):

Triggered before clicking an element

public void afterClickOn(WebElement element, WebDriver driver):

Triggered after clicking an element

## WebElement value changes

Methods shown below are triggered when the value of a WebElement changes:

```
public void beforeChangeValueOf(WebElement element, WebDriver driver): Triggered before the WebElement value changes
```

```
public void afterChangeValueOf(WebElement element, WebDriver driver): Triggered after the WebElement value changes
```

## [Navigating back](#)

Methods shown below are triggered when the navigate.back() is invoked on the WebDriver object:

public void beforeNavigateBack(WebDriver driver): Triggered before navigate.back() is invoked

public void afterNavigateBack(WebDriver driver): Triggered after navigate.back() is invoked

## [Navigate forward](#)

Methods shown below are triggered when the `navigate.forward()` is invoked on the `WebDriver` object:

`public void beforeNavigateForward(WebDriver driver): Triggered before navigate.forward() is invoked`

`public void afterNavigateForward(WebDriver driver): Triggered after navigate.forward() is invoked`

## [Navigate to](#)

Methods shown below are triggered when the navigate.to() is invoked on the WebDriver object:

public void beforeNavigateTo(java.lang.String url, WebDriver driver):  
Triggered before navigate.to() is invoked

public void afterNavigateTo(java.lang.String url, WebDriver driver):  
Triggered after navigate.to() is invoked

## Script execution

Methods shown below are triggered when JavaScript code is executed:

public void beforeScript(java.lang.String script, WebDriver driver):  
Triggered before executing JavaScript code

public void afterScript(java.lang.String script, WebDriver driver):  
Triggered after executing JavaScript code

## [Listening for exception](#)

```
public void onException(java.lang.Throwable throwable, WebDriver  
driver): Triggered when an exception is thrown
```

With this, we come to an end of [Chapter](#) We had a detailed look at event handling in Selenium.

## Conclusion

This chapter covered a very important topic, which is event handling in Selenium. The purpose of event handling was explained, and the important classes and interfaces involved in event handling were covered. We covered the various methods of listening to WebDriver events. Finally, we integrated our code with the framework and, as a recap, saw all the methods.

The reader got an in-depth understanding of the event handling process, along with information on how to log useful information.

The next chapter will deal with the backend side of our framework. We will learn how to use the CachedRowSet interface so that we can disconnect from the database after querying, which saves the cost of database hits. We will have a look at all operations that need to be done in order to integrate MySQL database with our framework.

## Questions

Write a program to use the Firefox driver to create the

Write a program with Firefox driver to implement  
WebDriverEventListener interface.

Write a program with Firefox driver that extends  
AbstractWebDriverEventListener abstract class

List down the advantages of using AbstractWebDriverEventListener  
abstract class over WebDriverEventListener interface

Add a keyword for selecting items from a drop-down and check  
whether the event listener messages appear in the console

*Database Operations*

We move on to another important chapter which deals with Database operations. We already saw a glimpse of a database select query in [chapter 1](#). In [chapter 1](#), we had used a JSONObject and JSONArray to store query results. This somehow is a tedious process that requires the automation developer first to create nested JSONArray from the database extract and then extract the required fields from each of the JSONArrays . Wouldn't it be nice instead if we have a single query that extracts all the fields for us?

We will see how to extract records from 4 tables of the database, namely and using a single query. We will see a structured approach to building the database logic. We get introduced to extracting database configuration from a property file called

## **Structure**

Here's what we will learn in this chapter:

Learn the various CRUD operations

Create a properties file for database configurations

Create queries in MySQL workbench to retrieve data from the four tables

Understand what a cached RowSet is

Create a class for database operations

Integrating the new class with the framework

## Objectives

Understanding CRUD operations

Learning the process of extracting records from a database

Learn about the various joins available

Create a query to join four tables

Understand Cached Rowset

Create a class for database operations

Integration with the framework

We begin by first understanding what CRUD operations mean.

## CRUD operations?

**CRUD** stands for **Create, Read, Update**, and These are the four operations that can be performed on a database. Create operation is responsible for creating database and tables inside the database. Read selects data from the various tables by joining them on a common key or set of keys. Update changes the value of a field or a set of fields based on the key used to narrow down the query results. Delete will delete the entire row or a set of rows from a table based on the key. Let's have a look at the syntax of each:

**CREATE operation:** We will see the syntax of the two main CREATE statements that we will utilize

**CREATE DATABASE:** The query mentioned below creates a database testdb:

```
CREATE DATABASE testdb;
```

**CREATE TABLE:** The query mentioned below creates a database table test:

```
CREATE TABLE test;
```

Let's move on to the read operation

**Read operation:** We will see the syntax of the basic SELECT statement that we will utilize to extract records from the table of the database

SELECT: The query mentioned below selects rows from testcases table based on TestCaseID 'SmokeTest1'

```
SELECT * FROM testcases WHERE TestCaseID='SmokeTest1';
```

Let's move on to the UPDATE operation.

**Update operation:** We will see the syntax of the basic UPDATE statement that we will utilize to update records in the table of the database.

UPDATE: The query mentioned below updates the ActionKey column from testcases table based on TestCaseID 'SmokeTest1' and TestStepID 'TSoo6'

```
UPDATE testcases SET ActionKey='verify' WHERE  
TestCaseID='SmokeTest1' and TestStepID='TSoo6';
```

Let's move on to the Delete operation.

**Delete operation:** We will see the syntax of the basic DELETE statement that we will utilize to delete records from the table of the database.

**DELETE:** The query mentioned below updates the ActionKey column from testcases table based on TestCaseID ‘SmokeTest1’ and TestStepID ‘TSoo6’

```
DELETE * FROM testcases WHERE TestCaseID="SmokeTest1" and  
TestStepID="TSoo6";
```

This was a brief introduction to the four CRUD operations. We next implement a few of these CRUD operations for creating the database and tables of our framework.

### *Creating the database and tables*

We utilize the CREATE DATABASE and CREATE TABLE queries that we looked at above. The query listed below creates our database

```
CREATE DATABASE 'selenium_framework' /*!40100 DEFAULT  
CHARACTER SET utf8 */
```

The queries listed below creates the tables and

### Create testconfig table

The CREATE statement shown below creates the testconfig table:

```
CREATE TABLE 'testconfig' ('TestCaseID' varchar(10) NOT NULL,  
'Desrciption' varchar(20) DEFAULT NULL, 'Execute' varchar(1)  
DEFAULT NULL, PRIMARY KEY ('TestCaseID') ) ENGINE=InnoDB  
DEFAULT CHARSET=utf8
```

### Create testcases table

The CREATE statement shown below creates the testcases table:

```
CREATE TABLE 'testcases' ('TestCaseID' varchar(10) NOT NULL,  
'TestStepID' varchar(10) NOT NULL, 'ActionKey' varchar(10) NOT  
NULL, 'ORID' varchar(10) NOT NULL, PRIMARY KEY  
('TestStepID'), KEY 'testcases_ibfk_1' ('TestCaseID'), CONSTRAINT  
'testcases_ibfk_1' FOREIGN KEY ('TestCaseID') REFERENCES  
'testconfig' ('TestCaseID') ) ENGINE=InnoDB DEFAULT  
CHARSET=utf8
```

### Create testdata table

The CREATE statement shown below creates the testdata table:

```
CREATE TABLE 'testdata' ('TestCaseID' varchar(10) NOT NULL,  
'TestStepID' varchar(10) NOT NULL, 'TestDataID' varchar(10) NOT  
NULL, 'DataKey' varchar(10) NOT NULL, PRIMARY KEY  
('TestDataID'), KEY 'testdata_ibfk_1' ('TestCaseID'), KEY  
'testdata_ibfk_2' ('TestStepID'), CONSTRAINT 'testdata_ibfk_1'  
FOREIGN KEY ('TestCaseID') REFERENCES 'testconfig'  
('TestCaseID'), CONSTRAINT 'testdata_ibfk_2' FOREIGN KEY  
('TestStepID') REFERENCES 'testcases' ('TestStepID') )  
ENGINE=InnoDB DEFAULT CHARSET=utf8
```

### Create object repository table

The CREATE statement shown below creates the object\_repository table:

```
CREATE TABLE 'object_repository' ('ORID' varchar(10) NOT NULL,  
'PageID' varchar(10) DEFAULT NULL, 'ObjectID' varchar(10)  
DEFAULT NULL, 'XPath' varchar(100) DEFAULT NULL, PRIMARY  
KEY ('ORID') ) ENGINE=InnoDB DEFAULT CHARSET=utf8
```

This completes our database setup.

## [Understanding JOINS](#)

In order to derive data from tables of a database, we need to utilize joins, which will extract data from the tables of a database. We will take a brief look at Inner Join and Outer Join and the use of the Outer Join to extract records from the database.

Let's have a look at inner join first.

## Inner Join

The inner join clause joins two tables based on a condition known as a join predicate.

The inner join clause compares each row from the first table with every row from the second table. If values in both rows cause the join condition to evaluate to true, the inner join clause creates a new row containing all columns of the two rows from both tables and include this new row in the final result set. The inner join clause includes only rows whose values match.

We will use the TestCases and TestData tables to demonstrate inner join. The TestCases and TestData tables contain the following data shown below.

The TestCases table contains the data shown below:

TestCaseID	TestStepID	ActionKey	ORID
SmokeTest1	TS001	openbrw	NULL
SmokeTest1	TS002	navigate	NULL
SmokeTest1	TS003	enterText	OR01
SmokeTest1	TS004	enterText	OR02
SmokeTest1	TS005	enterText	OR03
SmokeTest1	TS006	verify	OR04
NULL	NULL	NULL	NULL

*Figure 7.1*

The TestData table contains the data shown below:

TestCaseID	TestStepID	TestDataID	DataKey
SmokeTest1	TS003	TD001	ABC
SmokeTest1	TS004	TD002	DEF
SmokeTest1	TS005	TD003	GHI
SmokeTest1	TS006	TD004	JKL
NULL	NULL	NULL	NULL

*Figure 7.2*

The example shown below extracts data from the TestCases and TestData tables using INNER join:

```
SELECT * FROM testcases A INNER JOIN testdata B ON  
A.TestCaseID=B.TestCaseID AND A.TestStepID=B.TestStepID
```

We get four rows, as shown below:

TestCaseID	TestStepID	TestDataID	DataKey
SmokeTest1	TS003	TD001	ABC
SmokeTest1	TS004	TD002	DEF
SmokeTest1	TS005	TD003	GHI
SmokeTest1	TS006	TD004	JKL
NULL	NULL	NULL	NULL

*Figure 7.3*

Next, we move on to Outer Joins.

## Outer Joins

Outer joins return all records matching from both the tables. It can detect records having no match in a joined table. It returns NULL values for records of a joined table if a match is not found.

There are two types of Outer Joins: Left Outer Join and Right Outer Join

**Left Outer Join:** The LEFT JOIN keyword returns all records from the left table and the matched records from the right table. The result is NULL from the right side if there is no match.

```
SELECT * FROM testcases A LEFT JOINtestdata B ON  
A.TestCaseID=B.TestCaseID AND A.TestStepID=B.TestStepID ORDER  
BY A.TestStepID;
```

The resultset from the above query is shown below:

TestCaseID	TestStepID	ActionKey	ORID	TestCaseID	TestStepID	TestDataID	DataKey
SmokeTest1	TS001	openbrw	NULL	NULL	NULL	NULL	NULL
SmokeTest1	TS002	navigate	NULL	NULL	NULL	NULL	NULL
SmokeTest1	TS003	enterText	OR01	SmokeTest1	TS003	TD001	ABC
SmokeTest1	TS004	enterText	OR02	SmokeTest1	TS004	TD002	DEF
SmokeTest1	TS005	enterText	OR03	SmokeTest1	TS005	TD003	GHI
SmokeTest1	TS006	verify	OR04	SmokeTest1	TS006	TD004	JKL

**Figure 7.4**

**Right Outer Join:** The RIGHT JOIN keyword returns all records from the right table and the matched records from the left table. The result is NULL from the left side if there is no match.

```
SELECT * FROM testcases A RIGHT JOINtestdata B ON  
A.TestCaseID=B.TestCaseID AND A.TestStepID=B.TestStepID ORDER  
BY A.TestStepID;
```

The resultset from the above query is shown below:

TestCaseID	TestStepID	ActionKey	ORID	TestCaseID	TestStepID	TestDataID	DataKey
SmokeTest1	TS003	enterText	OR01	SmokeTest1	TS003	TD001	ABC
SmokeTest1	TS004	enterText	OR02	SmokeTest1	TS004	TD002	DEF
SmokeTest1	TS005	enterText	OR03	SmokeTest1	TS005	TD003	GHI
SmokeTest1	TS006	verify	OR04	SmokeTest1	TS006	TD004	JKL

**Figure 7.5**

## Inserting new testcases

A second test case has to be added in the database either using an excel file, or an INSERT query or manually. A sample insert query to insert data in the testconfig table is shown below:

```
INSERT INTO 'selenium_framework'.'testconfig' ('TestCaseID',  
'Desrciption', 'Execute') VALUES ('SmokeTest3', 'Creating Invoice',  
'Y');
```

Insert data into the testcases table:

```
INSERT INTO 'selenium_framework'.'testcases' ('TestCaseID',  
'TestStepID', 'ActionKey', 'ORID') VALUES ('SmokeTest3', 'TS013',  
'openbrw', 'NULL');  
INSERT INTO 'selenium_framework'.'testcases' ('TestCaseID',  
'TestStepID', 'ActionKey', 'ORID') VALUES ('SmokeTest3', 'TS014',  
'navigate', 'NULL');  
INSERT INTO 'selenium_framework'.'testcases' ('TestCaseID',  
'TestStepID', 'ActionKey', 'ORID') VALUES ('SmokeTest3', 'TS015',  
'enterText', 'OR01');  
INSERT INTO 'selenium_framework'.'testcases' ('TestCaseID',  
'TestStepID', 'ActionKey', 'ORID') VALUES ('SmokeTest3', 'TS016',  
'enterText', 'OR02');  
INSERT INTO 'selenium_framework'.'testcases' ('TestCaseID',  
'TestStepID', 'ActionKey', 'ORID') VALUES ('SmokeTest3', 'TS017',  
'enterText', 'OR03');
```

```
INSERT INTO 'selenium_framework'.'testcases' ('TestCaseID',  
'TestStepID', 'ActionKey', 'ORID') VALUES ('SmokeTest3', 'TS018',  
'click', 'ORo4');
```

As per need data has to be modified in the testdata and object\_repository table.

### Extracting records from a database in a single query.

Let's extract the testcase steps for the testcases which have the Execute flag set to Y. We should take data from testconfig, testcases, and testdata tables. The query for accomplishing this is given below:

```
select A.TestCaseID,A.TestStepID,A.ActionKey,B.XPath,C.DataKey from
testcases A left outer join object_repository B on A.ORID=B.ORID
left outer join testdata C on A.TestCaseID=C.TestCaseID and
A.TestStepID=C.TestStepID left outer join object_repository D on
A.ORID=D.ORID where A.TestCaseID in (select TestCaseID from
testconfig TC where TC.Execute='Y') order by A.TestStepID asc
```

The output of the above query is given below:

TestCaseID	TestStepID	ActionKey	XPath	DataKey
SmokeTest1	TS001	openbrw	NULL	NULL
SmokeTest1	TS002	navigate	NULL	NULL
SmokeTest1	TS003	enterText	//*[@id='ABC']	ABC
SmokeTest1	TS004	enterText	//*[@id='DEF']	DEF
SmokeTest1	TS005	enterText	//*[@id='GHI']	GHI
SmokeTest1	TS006	click	//*[@id='JKL']	JKL
SmokeTest2	TS007	openbrw	NULL	NULL
SmokeTest2	TS008	navigate	NULL	NULL
SmokeTest2	TS009	enterText	//*[@id='ABC']	MNO
TestCaseID	TestStepID	ActionKey	XPath	DataKey
SmokeTest2	TS010	enterText	//*[@id='DEF']	KKK
SmokeTest2	TS011	enterText	//*[@id='GHI']	YYY
SmokeTest2	TS012	click	//*[@id='JKL']	ZZZ
SmokeTest3	TS013	openbrw	NULL	NULL
SmokeTest3	TS014	navigate	NULL	NULL
SmokeTest3	TS015	enterText	//*[@id='ABC']	AAA
SmokeTest3	TS016	enterText	//*[@id='DEF']	BBB
SmokeTest3	TS017	enterText	//*[@id='GHI']	CCC
SmokeTest3	TS018	click	//*[@id='JKL']	DDD

**Figure 7.6**

Now that we have seen the query that we need to execute let's execute it in a Java class.

## [Creating a class for database operations](#)

In this section, let's create a Java class for database operations. Before executing the class, make sure that the Executeflag in TestConfig is set to Y only for ResultSetMetaData interface is used for extracting column names so that the column names can be used for any purpose like reporting.

```
public class DBExtract2 {  
  
    private static String pageID;  
    private static String objectID;  
    private static String xPath;  
    private static Connection conn;  
    private static Statement stmt;  
  
    public static void main(String[] args) {  
        ResultSet resultSet = null;  
        ResultSetMetaData rsmd = null;  
        Properties prop = new Properties();  
        InputStream iStream = null;  
        String dbURL=null;  
        String dbUName=null;  
        String dbPwd=null;  
  
        /*****Create Connection, Statement and Resultset objects and  
        execute SQL Query and fetch metadata*****/
```

```
try {

iStream=DBExtract2.class.getClassLoader().getResourceAsStream("dbconfig.properties");
prop.load(iStream);
dbURL=prop.getProperty("DB_URL");
dbUName=prop.getProperty("DB_UNAME");
dbPwd=prop.getProperty("DB_PASSWORD");

conn = DriverManager.getConnection(
dbURL, dbUName,
dbPwd);
stmt = conn.createStatement();
resultSet = stmt
.executeQuery("select
A.TestCaseID,A.TestStepID,A.ActionKey,B.XPath,C.DataKey from
testcases A left outer join object_repository B on A.ORID=B.ORID
left outer join testdata C on A.TestCaseID=C.TestCaseID and
A.TestStepID=C.TestStepID where A.TestCaseID in (select
TestCaseID from testconfig TC where TC.Execute='Y') order by
A.TestStepID asc");
rsmd = resultSet.getMetaData();

resultSet.next();
} catch (Exception e) {
// TODO Auto-generated catch block
e.printStackTrace();
}
/**Iterate over Resultset and print individual values**/
try {
```

```
while (!resultSet.isAfterLast()) {  
    String testcaseid =  
        resultSet.getString("A.TestCaseID");  
    String teststepid =  
        resultSet.getString("A.TestStepID");  
  
    String actionKey =  
        resultSet.getString("A.ActionKey");  
    String xpath = resultSet.getString("B.XPath");  
    String datakey = resultSet.getString("C.DataKey");  
  
    for (int j1 = 1; j1 <= rsmd.getColumnCount(); j1++) {  
        System.out.println("Column: " +  
            rsmd.getColumnName(j1));  
        System.out.println("Value: "  
            +  
            resultSet.getString(rsmd.getColumnName(j1)));  
  
    }  
    resultSet.next();  
}  
} catch (SQLException e) {  
    e.printStackTrace();  
} finally {  
    try {  
        System.out.println("Closing objects");  
        stmt.close();  
        resultSet.close();  
        conn.close();  
    } catch (SQLException e) {  
        // TODO Auto-generated catch block  
    }
```

```
    e.printStackTrace();  
}  
}  
  
}
```

**OUTPUT:**

**Column: TestCaseID**

**Value: SmokeTest1**

**Column: TestStepID**

**Value: TSoo1**

**Column: ActionKey**

**Value: openbrw**

**Column: XPath**

**Value: null**

**Column: DataKey**

**Value: null**

**Column: TestCaseID**

**Value: SmokeTest1**

**Column: TestStepID**

**Value: TSoo2**

**Column: ActionKey**

**Value: navigate**

**Column: XPath**

**Value: null**

**Column: DataKey**

**Value: null**

**Column: TestCaseID**

**Value: SmokeTest1**

**Column:** TestStepID  
**Value:** TSoo3  
**Column:** ActionKey  
**Value:** entertext  
**Column:** XPath  
**Value:** //\*[@id='ABC']  
**Column:** DataKey  
**Value:** ABC  
**Column:** TestCaseID  
**Value:** SmokeTest1  
**Column:** TestStepID  
**Value:** TSoo4

**Column:** ActionKey  
**Value:** entertext  
**Column:** XPath  
**Value:** //\*[@id='DEF']  
**Column:** DataKey  
**Value:** DEF  
**Column:** TestCaseID  
**Value:** SmokeTest1  
**Column:** TestStepID  
**Value:** TSoo5  
**Column:** ActionKey  
**Value:** entertext  
**Column:** XPath  
**Value:** //\*[@id='GHI']  
**Column:** DataKey  
**Value:** GHI  
**Column:** TestCaseID  
**Value:** SmokeTest1  
**Column:** TestStepID

**Value:** TSoo6

**Column:** ActionKey

**Value:** click

**Column:** XPath

**Value:** //\*[@id='JKL']

**Column:** DataKey

**Value:** JKL

**Closing objects**

As can be seen from the output, all TestSteps along with the corresponding xpaths and test data are extracted. Data has been extracted from four tables which are testconfig, testcases, and

Notice the lines:

```
Properties prop = new Properties();
DBExtract2.class.getClassLoader().getResourceAsStream("dbconfig.properties");
prop.load(iStream);
dbURL=prop.getProperty("DB_URL");
dbUName=prop.getProperty("DB_UNAME");
dbPwd=prop.getProperty("DB_PASSWORD");
```

This loads the dbconfig.properties file and fetches the values from the file whose contents are shown below:

DB\_URL=jdbc:mysql://localhost:3306/selenium\_framework

DB\_UNAME=root

DB\_PASSWORD=Pc9121975!

Next, instead of putting the main() method in DBExtract2 class, let convert the main() method to a regular static method. Since we close the database objects of the final clause, using the recordset outside the DBExtract2 class is not possible unless we put the recordset contents into a JSONArray or a simple ArrayList of This is a tedious process since we need to create a JSONArray and create a JSON file containing this array, which can be accessed outside the class. A better alternative is to use something which is similar to a but at the same time, it should be disconnected from the database since we will return this from our method in the class in order to use it in some other class. For this, we need to learn a very important concept of disconnected Let's next move on to a very important concept called

## [Introducing RowSet](#)

The RowSet interface, in turn, extends the ResultSet interface and thus shares it's capabilities. The RowSet is scrollable and updatable. A RowSet object can either be connected or disconnected. A connected RowSet object makes use of a JDBC driver to make a connection to a relational database and maintains that connection throughout its life span. A disconnected RowSet object makes a connection to a data source only when it requires to read in data from a ResultSet object or to write data back to the data source. After reading data or writing data to its data source, the RowSet object disconnects from it and thus becomes disconnected. During much of its life span, a disconnected RowSet object is disconnected from its data source and operates independently. Let's have a look at the two types of

## Connected RowSet objects

Only one of the RowSet implementations is the interface. It is almost similar to a ResultSet and is most of the time used as a wrapper to make the non-scrollable and read-only ResultSet object scrollable and updatable. A scrollable ResultSet is one which makes it possible to retrieve the data in a forward direction as well as a backward direction, but no updates are allowed. RowSet being updatable takes care of the non-updateable feature of a scrollable

## *Disconnected RowSet objects*

There are four RowSet implementations, namely CachedRowSet, JoinRowSet, WebRowSet, and that fall into the category of disconnected RowSets. DisconnectedRowSet objects have all the capabilities of connected RowSet objects, and in addition, they have the capabilities that are available only to disconnectRowSet objects. Not having to maintain a connection to a data source makes disconnected RowSet objects much lightweight than a JdbcRowSet object or a ResultSet object. Disconnected RowSet objects are also serializable, and them being both serializable and lightweight makes them ideal candidates for sending data over a network.

## CachedRowSet

A CachedRowSet falls in the category of disconnected RowSet and hence is updatable and scrollable. The best way to see the power of CachedRowSet is through an example. Let's jump straight into the example. We create a new for this purpose.

## Database extract in a method

How can we utilize the power of the database extract in our framework? The first step towards this is to convert the main() method shown above to a regular method named which will be a static method returning an object of The steps to accomplish this are given below:

**Step 1:** Add the declaration shown below to the class:

```
private static CachedRowSet cachedRowset;
```

**Step 2:** Add the lines shown below to the method

```
cachedRowset = rsFactory.createCachedRowSet();
cachedRowset.populate(resultSet);
```

**Step 3:** Change the method declaration to the one shown below

```
public static CachedRowSet extractRecords()
```

**Step 3:** Add a return statement which returns the CachedRowSet object

```
return cachedRowset;
```

The entire class is shown below:

```
public class DBExtract3 {
    private static String pageID;
```

```
private static String objectID;
private static String xPath;
private static Connection conn;
private static CachedRowSet cachedRowset;
private static Statement stmt;

@SuppressWarnings("finally")
public static CachedRowSet extractRecords() {
    ResultSet resultSet = null;
    ResultSetMetaData rsmd = null;

    Properties prop = new Properties();
    InputStream iStream = null;
    String dbURL = null;
    String dbUName = null;
    String dbPwd = null;
    /*****Create Connection, Statement and Resultset objects and
execute SQL Query and fetch metadata*****/
    try {
        iStream =
            DBExtract2.class.getClassLoader().getResourceAsStream("dbconfig.pro
perties");
        prop.load(iStream);
        dbURL = prop.getProperty("DB_URL");
        dbUName = prop.getProperty("DB_UNAME");
        dbPwd = prop.getProperty("DB_PASSWORD");
        conn = DriverManager.getConnection(
            dbURL, dbUName,
            dbPwd);
        stmt = conn.createStatement();
        resultSet = stmt
```

```

.executeQuery("select
A.TestCaseID,A.TestStepID,A.ActionKey,B.XPath,C.DataKey
from testcases A left outer join object_repository B on
A.ORID=B.ORID left outer join testdata C on
A.TestCaseID=C.TestCaseID and A.TestStepID=C.TestStepID where
A.TestCaseID in (select TestCaseID from testconfig TC where
TC.Execute='Y') order by A.TestStepID asc");
rsmd = resultSet.getMetaData();
RowSetFactory rsFactory = RowSetProvider.newFactory();

cachedRowset = rsFactory.createCachedRowSet();
cachedRowset.populate(resultSet);
} catch (Exception e) {
e.printStackTrace();
} finally {
try {
System.out.println("Closing Connection");
stmt.close();
resultSet.close();

conn.close();
} catch (SQLException e) {
e.printStackTrace();
}
return cachedRowset;
}
}
}
}

```

Finally, let's change the FrameworkDriver class.

## Integration with the framework

Let's have a look at the steps that need to be followed in the FrameworkDriver class so that we can take in query results from

**Step 1:** Add the following line in the main() method.

```
CachedRowSet crs = DBExtract3.extractRecords();
```

**Step 2:** Replace all ResultSet objects in the code with the CachedRowSet object created above.

The entire code of the FrameworkDriver is given below

```
public class Framework_Driver {  
  
    public static void main(String[] args) {  
  
        ResultSet resultSet = null;  
  
        try {  
            CachedRowSet crs = DBExtract3.extractRecords();  
            ResultSetMetaData rsmd = crs.getMetaData();  
            System.out.println("Framework Driver Logic started");  
            crs.next();  
            while (!crs.isAfterLast()) {  
                String testcaseid = crs.getString("TestCaseID");  
            }  
        } catch (Exception e) {  
            e.printStackTrace();  
        }  
    }  
}
```

```

String teststepid = crs.getString("TestStepID");

String actionKey = crs.getString("ActionKey");
String xpath = crs.getString("XPath");
String datakey = crs.getString("DataKey");

for (int j1 = 1; j1 <= rsmd.getColumnCount(); j1++) {
    System.out.println("Column in FrameworkDriver: " +
        rsmd.getColumnName(j1));
    System.out.println("Value in FrameworkDriver: " +
        + crs.getString(rsmd.getColumnName(j1)));
}

crs.next();
}
} catch (SQLException e) {
e.printStackTrace();
}
System.out.println("Framework Driver Logic ended");
}
}

```

#### **OUTPUT:**

**Closing Connection**  
**Framework Driver Logic started**  
**Column in FrameworkDriver: TestCaseID**  
**Value in FrameworkDriver: SmokeTest1**  
**Column in FrameworkDriver: TestStepID**

Value in FrameworkDriver: TSoo1  
Column in FrameworkDriver: ActionKey  
Value in FrameworkDriver: openbrw  
Column in FrameworkDriver: XPath  
Value in FrameworkDriver: null  
Column in FrameworkDriver: DataKey  
Value in FrameworkDriver: null  
Column in FrameworkDriver: TestCaseID  
Value in FrameworkDriver: SmokeTest1  
Column in FrameworkDriver: TestStepID  
Value in FrameworkDriver: TSoo2  
Column in FrameworkDriver: ActionKey  
Value in FrameworkDriver: navigate  
Column in FrameworkDriver: XPath  
Value in FrameworkDriver: null

Column in FrameworkDriver: DataKey  
Value in FrameworkDriver: null  
Column in FrameworkDriver: TestCaseID  
Value in FrameworkDriver: SmokeTest1  
Column in FrameworkDriver: TestStepID  
Value in FrameworkDriver: TSoo3  
Column in FrameworkDriver: ActionKey  
Value in FrameworkDriver: entertext  
Column in FrameworkDriver: XPath  
Value in FrameworkDriver: //\*[@id='ABC']  
Column in FrameworkDriver: DataKey  
Value in FrameworkDriver: ABC  
Column in FrameworkDriver: TestCaseID  
Value in FrameworkDriver: SmokeTest1  
Column in FrameworkDriver: TestStepID  
Value in FrameworkDriver: TSoo4

**Column in FrameworkDriver: ActionKey**  
**Value in FrameworkDriver: entertext**  
**Column in FrameworkDriver: XPath**  
**Value in FrameworkDriver: //\*[@id='DEF']**  
**Column in FrameworkDriver: DataKey**  
**Value in FrameworkDriver: DEF**  
**Column in FrameworkDriver: TestCaseID**  
**Value in FrameworkDriver: SmokeTest1**  
**Column in FrameworkDriver: TestStepID**  
**Value in FrameworkDriver: TSoo5**  
**Column in FrameworkDriver: ActionKey**  
**Value in FrameworkDriver: entertext**  
**Column in FrameworkDriver: XPath**

**Value in FrameworkDriver: //\*[@id='GHI']**  
**Column in FrameworkDriver: DataKey**  
**Value in FrameworkDriver: GHI**  
**Column in FrameworkDriver: TestCaseID**  
**Value in FrameworkDriver: SmokeTest1**  
**Column in FrameworkDriver: TestStepID**  
**Value in FrameworkDriver: TSoo6**  
**Column in FrameworkDriver: ActionKey**  
**Value in FrameworkDriver: click**  
**Column in FrameworkDriver: XPath**  
**Value in FrameworkDriver: //\*[@id='JKL']**  
**Column in FrameworkDriver: DataKey**  
**Value in FrameworkDriver: JKL**  
**Framework Driver Logic ended**

It can be seen from the output that the database connection is closed before further processing of the obtained resultset takes

place. This is the magic of This completes the chapter on database logic.

## Conclusion

This chapter covered database retrieval. We went over the basics of database CRUD operations. We also saw the basics of Joins in MySQL. We saw how to create database extract logic to pull data from 4 connected tables utilizing the power of joins. Next, we explored the power of disconnected RowSets with CachedRowSet example and eventually integrated the database logic with our framework.

The next chapter will explore a testing framework which is very helpful while creating frameworks. We will eliminate the main() method and trigger test execution from an xml file called testng.xml. We will see the power of TestNG when the number of test cases increases in our project. We will also explore the concept of DataProviders in which will help to pull data from the CachedRowSet row by row into the framework logic.

## Questions

Explain the difference between inner join and left join

Write a program to show the working of a right join

What is the use of

Write a program showing the use of ResultSetMetaData

Add a new TestCase and try extracting records for that test case

*Get Introduced to TestNG*

A testing framework comes in very handy when we need to run our framework end to end. Up until now, we have been using the main method to run our tests. We will see how to run tests without the main method. For this purpose, we will see what a TestNG XML is and how it simplifies the creation of a framework. We get introduced to the concept of a DataProvider. We also see the different annotations that the TestNG framework provides us.

Understanding of a testing framework is very important if one has to acquire mastery in the creation of his own framework. We have taken TestNG as our framework because it provides many annotations like AfterSuite, AfterMethod, AfterClass, etc. which was missing in a similar framework used in unit testing, which is JUnit. TestNG has gained popularity over JUnit in the past few years.

After reading and comprehending this chapter, the reader will be able to create a framework quickly and easily, thus helping the testing and project team.

## Structure

Here's what we will learn in this chapter:

Learn what a testing framework means

Get introduced to TestNG

Understand the TestNG xml

Learn the different annotations that TestNG provides

Understand the concept of Test classes

Learn what Test Suites are

## Objectives

Understand what a testing framework is

Get introduced to the TestNG framework

Learn about the TestNG xml

Learn various annotations that TestNG provides

Understand the DataProvider annotation

Create Test classes for executing different scenarios

We begin by first understanding what a testing framework is.

## What is a testing framework?

A testing framework is a set of guidelines or rules through which efficient test cases can be designed. Specifically, for test automation, a testing framework is one that provides the automation developer with a set of methods and utilities through which the framework creation process can be simplified. A testing framework provides an environment for the execution of automated tests. It gives the facility to plug into the application under test. It provides us the facility to execute tests and report the results after the completion of test execution. A testing framework is independent of the application under test, and one can easily expand the framework by adding more test cases. Execution of several test cases that are either dependent or independent on each other is made possible through a testing framework. There are some prerequisite conditions that have to be sometimes made available before executing a set of tests, which is made possible by annotations in TestNG, for example, opening a browser before test execution.

Predominantly, there are two frameworks available in the market, which are JUnit and TestNG. JUnit, being the older of the two, is popular among developers as a Unit Testing Framework. We are going to learn about TestNG in this chapter, which stands for Test Next Generation.

Let's start with an introduction to TestNG.

## What is TestNG

TestNG, where NG represents the next generation, is a test automation framework, which is a combination of JUnit (in Java) and NUnit (in C#). It is frequently used for unit, functional, integration, and end-to-end testing. TestNG has become extremely popular in the test automation world within a short time and is one of the most widely used testing frameworks among developers (both application and test automation developers). It mainly uses annotations in Java to configure and write test methods.

TestNG was introduced by Cedric Beust. He developed it to overcome shortcomings in JUnit. A few of the features of TestNG are:

Additional before and after annotations like @BeforeTest, @AfterTest, @BeforeClass, @AfterClass

Grouping test methods by creating TestNG groups

Incorporating dependencies using dependsOn

Executing tests in parallel

Basic test execution reporting

TestNG is a testing framework, which is written in Java and can be used with Java as well as with Java-related languages such as Groovy. In TestNG, suites and tests are configured through the use of XML files. By default, the name of this file is testng.xml but can be given another name if needed.

TestNG enables us to do test configuration through XML files and allows us to include (or exclude) respective packages, classes, and methods in the test suite. It also allows users to group test methods into particular named groups, which can be included or excluded as part of the test execution.

Parameterization of test methods is easy using TestNG, and it also provides the facility of creating data-driven tests.

TestNG has exposed its API, which has made it easy to add custom functionalities or extensions if required.

Let's have a look at the testng.xml next, which will be used for configuring our tests.

## Configure TestNG

Follow the steps shown below in order to configure TestNG.

### **Step 1: Add the Maven dependency**

In order to use the TestNG framework, we should first add the TestNG dependency shown below in pom.xml. This dependency can be found at

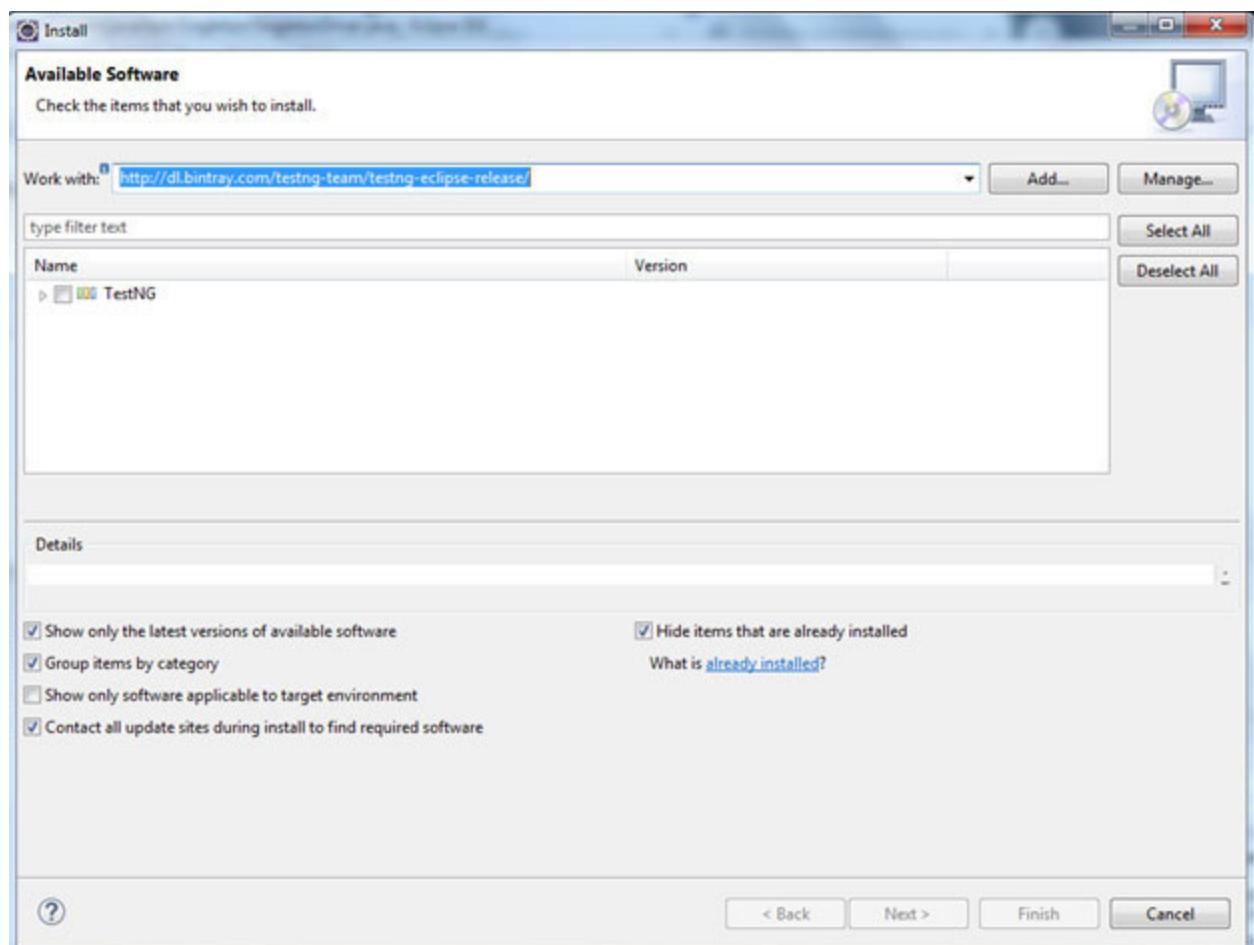
<https://mvnrepository.com/artifact/org.testng/testng/6.14.3>

org.testng  
testng  
6.14.3  
test

Save the project. Right click on the project in **Project Explorer** and select **Maven | Update**

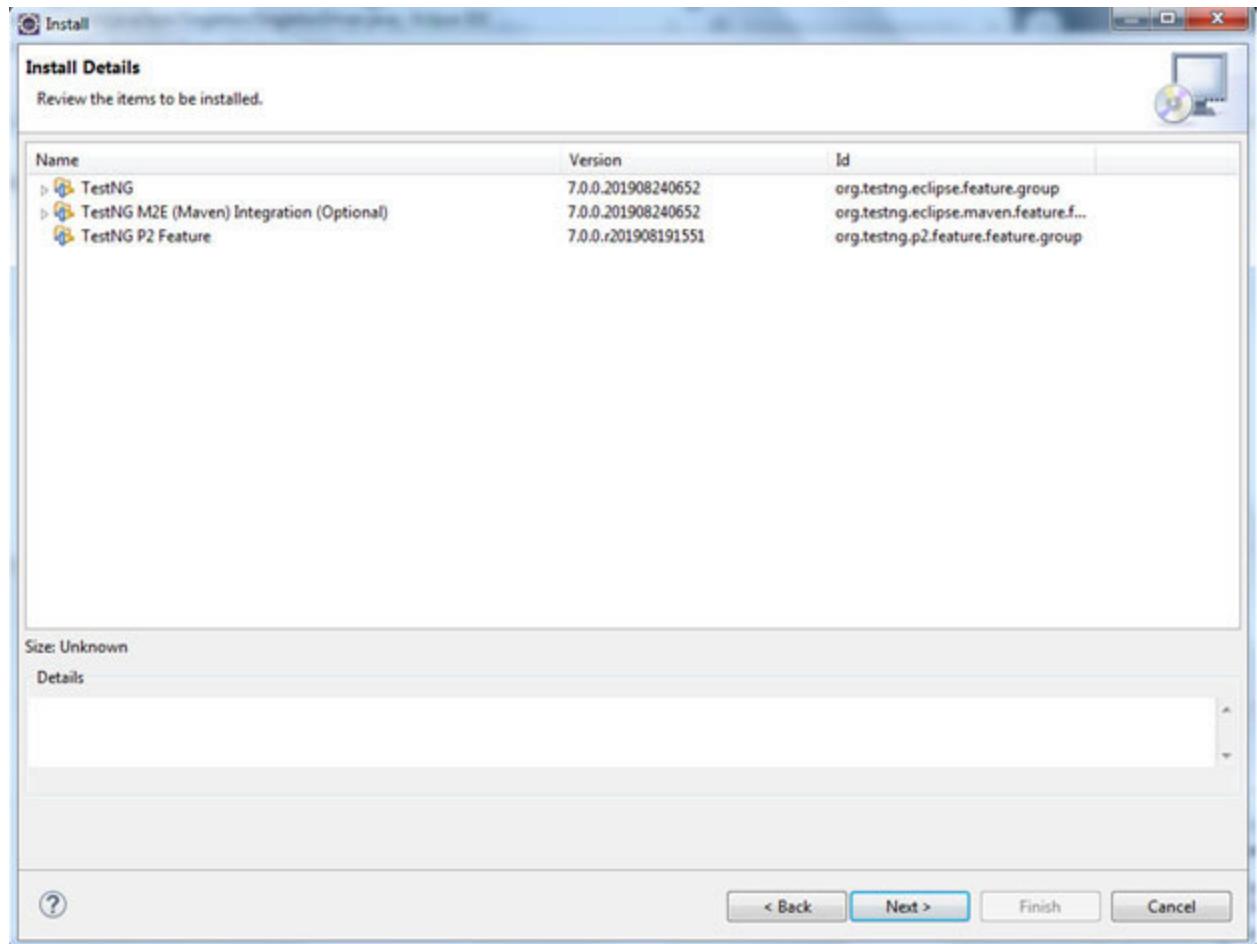
### **Step 2: Add the TestNG Plugin to Eclipse**

Go to **Help | Install New** The dialog box shown below is displayed. Enter <http://dl.bintray.com/testng-team/testng-eclipse-release/> in the text box. Select the **TestNG** checkbox and click the **Next** button after it gets enabled.



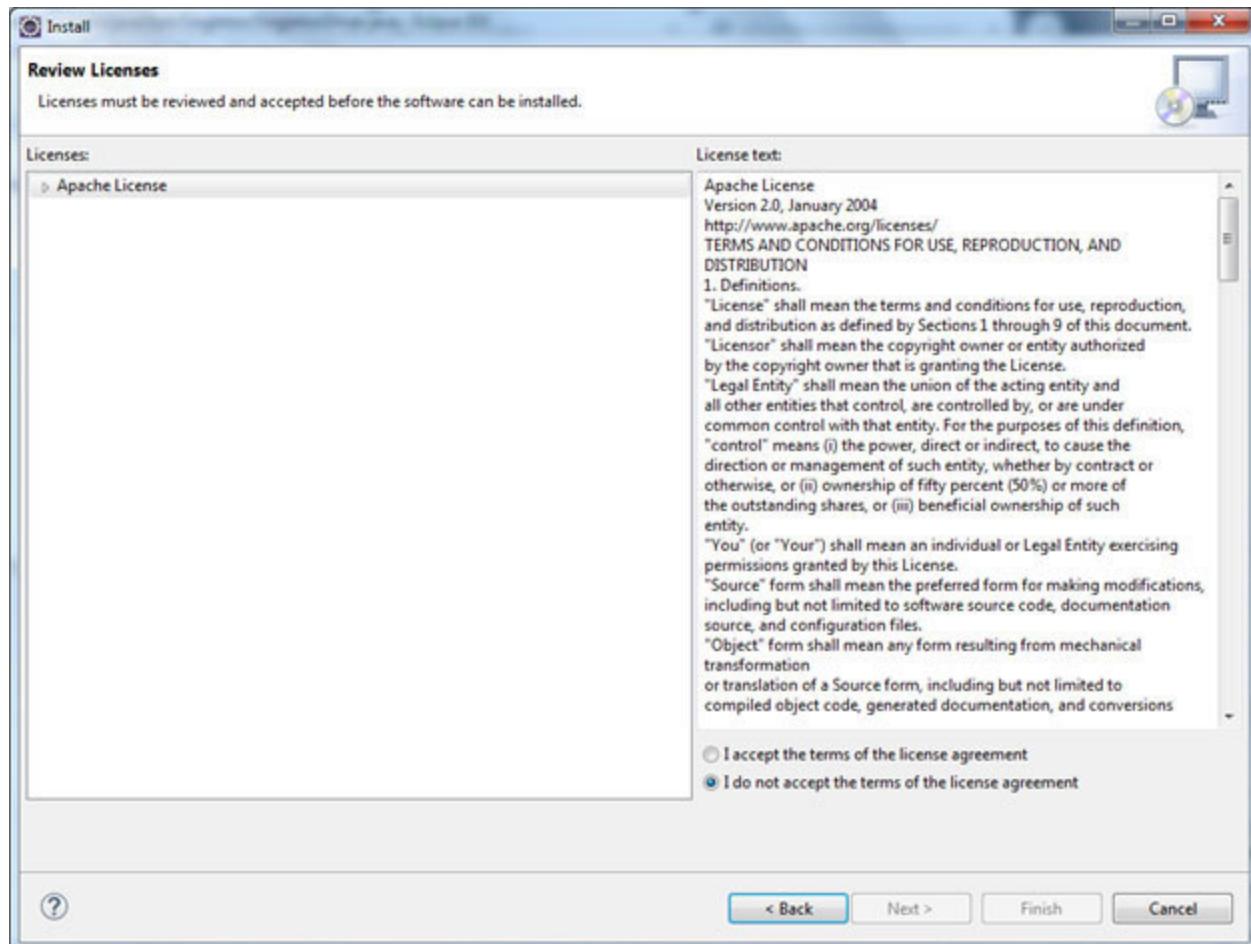
*Figure 8.1*

On clicking the screen shown below is displayed.



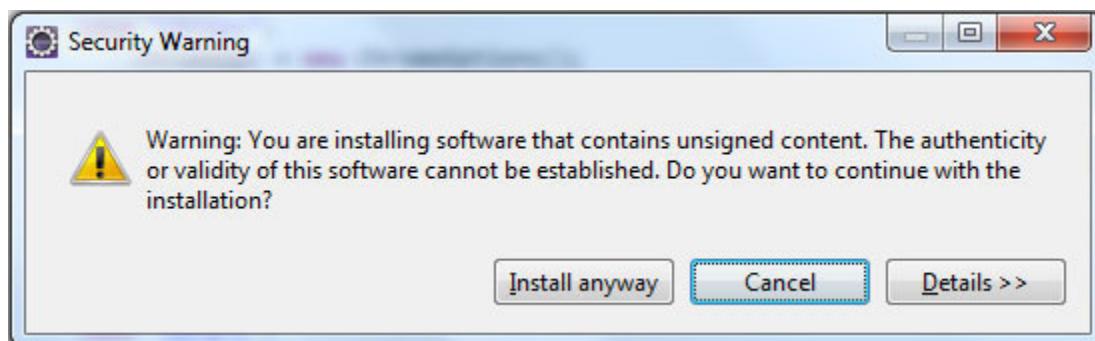
*Figure 8.2*

Click



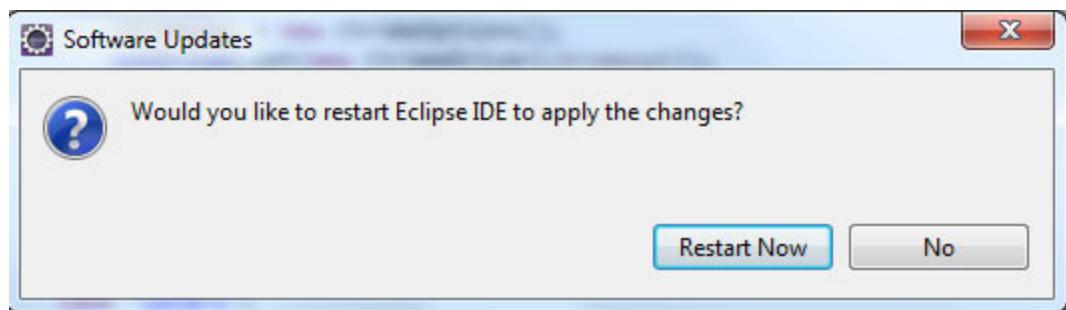
*Figure 8.3*

Click the **I accept...** radio button and click on **Finish** once it gets enabled. If the message box shown below gets displayed, click on **Install**



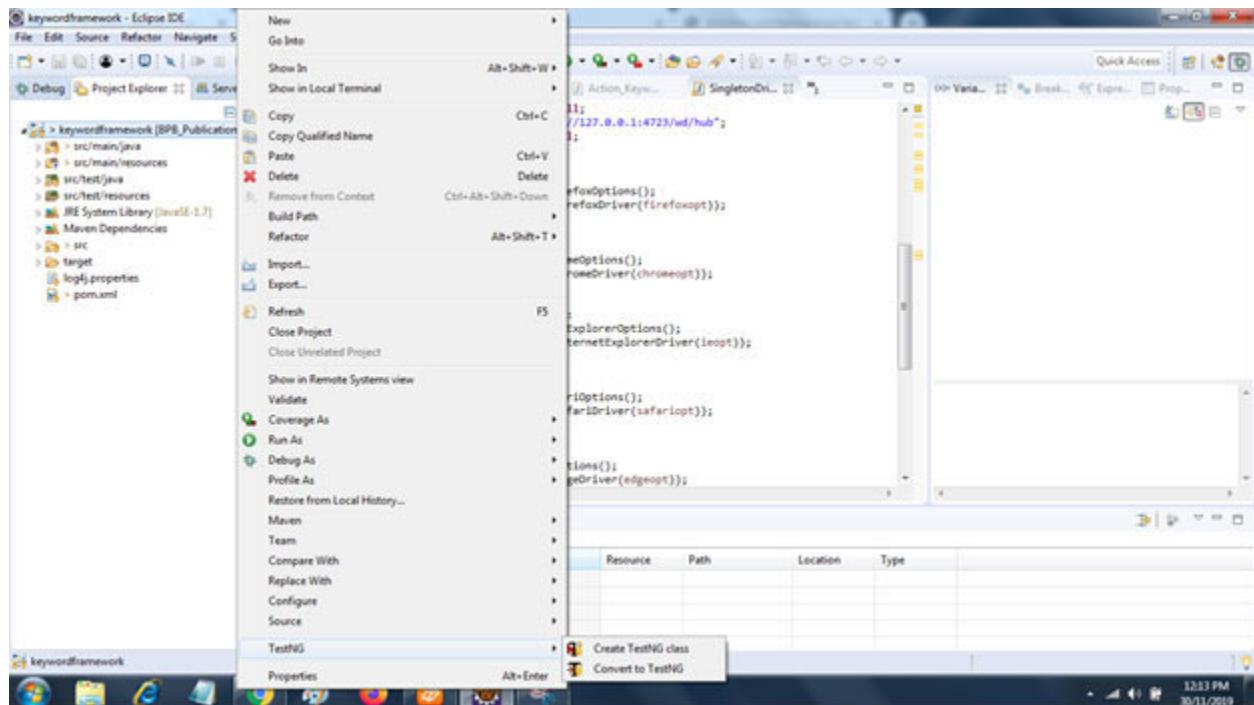
*Figure 8.4*

Once the installation completes, click **Restart Now** on the message box that appears:



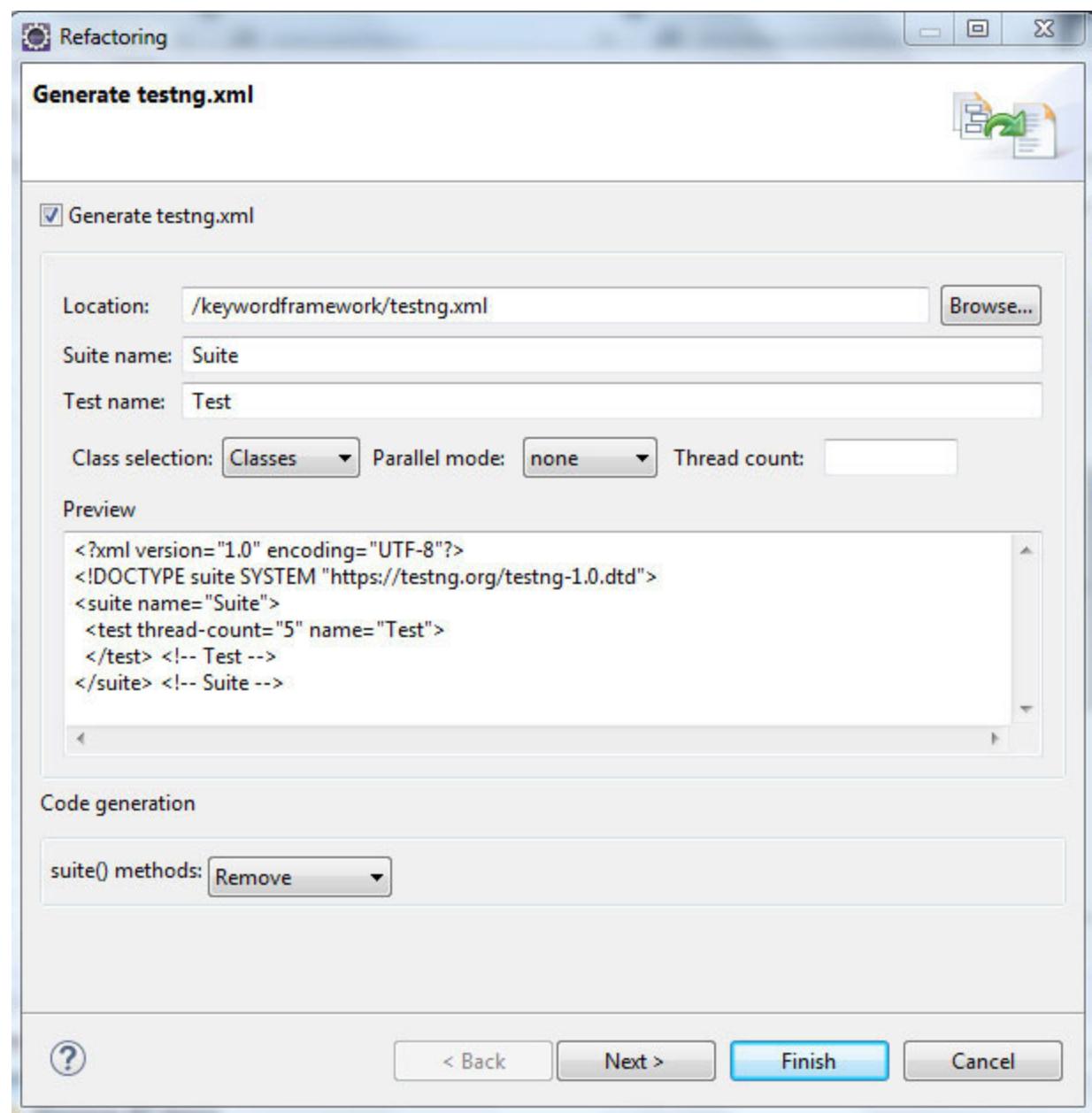
**Figure 8.5**

After Eclipse restarts, right-click on the project in project explorer. The TestNG option should get displayed, as shown below:



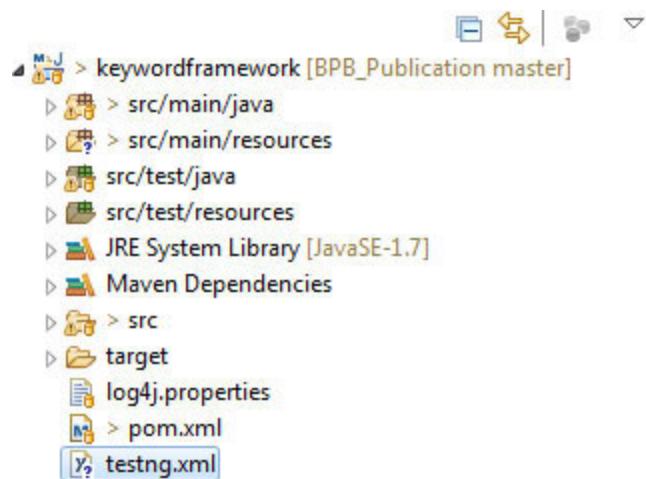
**Figure 8.6**

Click on **Convert** to The dialog for generating a file called testng.xml appears as shown below:



**Figure 8.7**

A file with the name testng.xml appears in the project explorer:

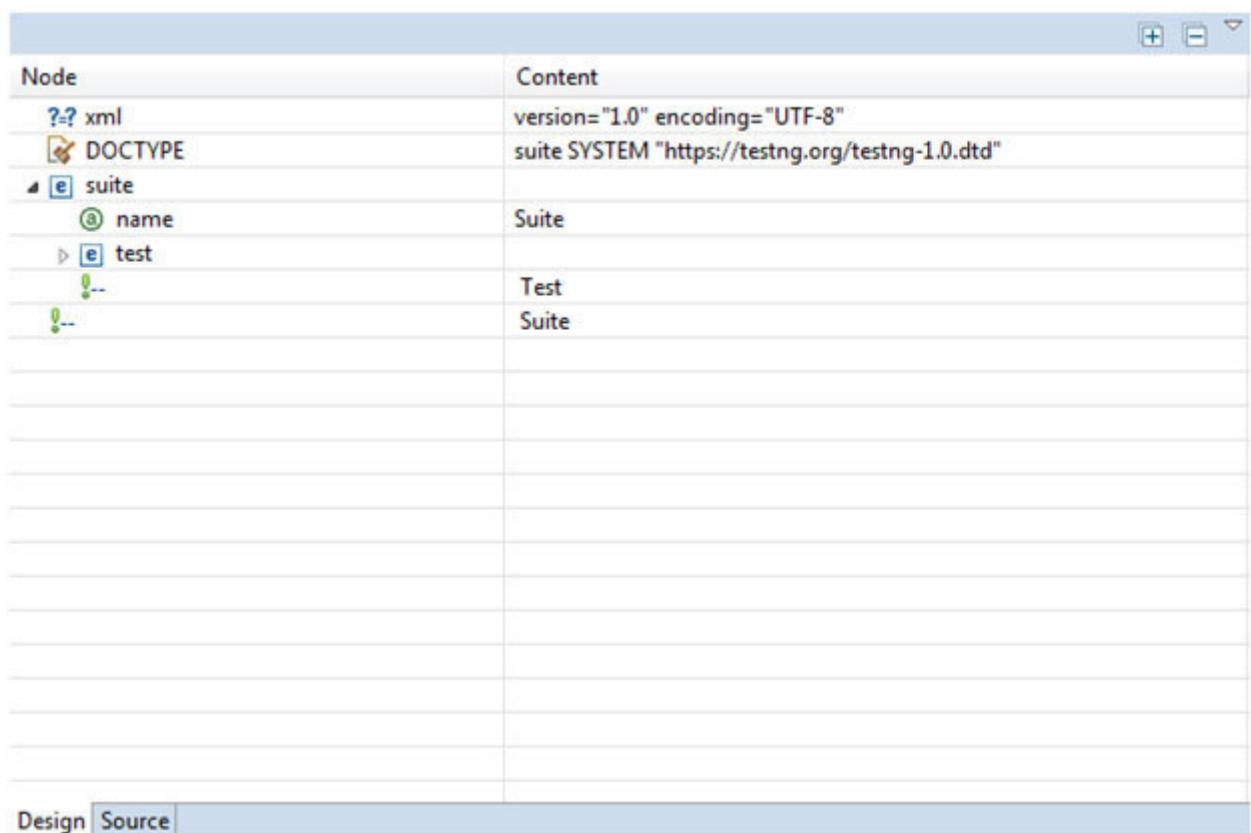


**Figure 8.8**

Let's now see what the testng.xml file is all about.

## Understanding TestNG XML

In order to understand the TestNG xml, lets open it in Eclipse by double clicking it in the project explorer. This is what the screen looks like with two tabs displayed: **Design** and Click on Source to edit the TestNG XML.



*Figure 8.9*

The unedited TestNG xml is shown below:

```
version="1.0" encoding="UTF-8"?>
```

```
suite SYSTEM "https://testng.org/testng-1.0.dtd">
name="Suite">
thread-count="5" name="Test">
```

Apart from the mandatory xml and DOCTYPE tags, there are and tags displayed. suite represents a collection of tests and test refers to a single test.

Let's understand what a test class is next.

## What are the Test classes?

Test classes are classes that contain annotations. These annotations can be @Test, @BeforeTest, etc. Test classes contain full test cases. A full test can consist of opening the browser, navigating to the URL, entering login credentials, performing some optional action, and eventually logging out.

Edit the testng.xml as shown below:

```
version="1.0" encoding="UTF-8"?>
suite SYSTEM "http://testng.org/testng-1.0.dtd">
name="Suite">
name="Test">
name="keywordframework.TestClass1"/>
```

Below is the test class TestClass1.java which performs a log-in log-out test with the FreeCRM URL. Notice how we have invoked keywords from the ActionKeywords and

```
package keywordframework;

import java.util.concurrent.TimeUnit;

import org.testng.annotations.AfterTest;
import org.testng.annotations.BeforeTest;
```

```
import org.testng.annotations.Test;

import bpb.Singleton.SingletonDriver;
import bpb.WaitLogic.JavascriptUtility;
import bpb.WaitLogic.WaitClass;
import bpb.keywords.Action_Keywords;

public class TestClass1 {

    Action_Keywords actKeywords = new Action_Keywords();

    WaitClass wdWait = null;
    String btnLogin = "//a[@href='https://ui.freecrm.com']";
    String uNameXPath = "//input[@name='email']";
    String passwordXPath = "//input[@name='password']";
    String btnSubmit = "//div[text()='Login']";
    String logoutParent = "//div[@class='ui buttons']/div";
    String btnLogout ="//span[text()='Log Out']";
    String callQueue = "//span[text()='Call Queue']";

    public TestClass1() {
        System.out.println("hi");
    }

    @BeforeTest
    public void setUp() {
        actKeywords.openBrowser("chrome");
        wdWait = new WaitClass();
        wdWait.waitImplicitly();
    }
}
```

```
}
```

```
@Test  
public void test1() {  
actKeywords.navigateURL("http://www.freecrm.com");  
actKeywords.clickElement(btnLogin);  
actKeywords.enterText(uNameXPath, "chaubalpinakin@gmail.com");  
actKeywords.enterText(passwordXPath, "Pc9121975!");  
actKeywords.clickElement(btnSubmit);  
}
```

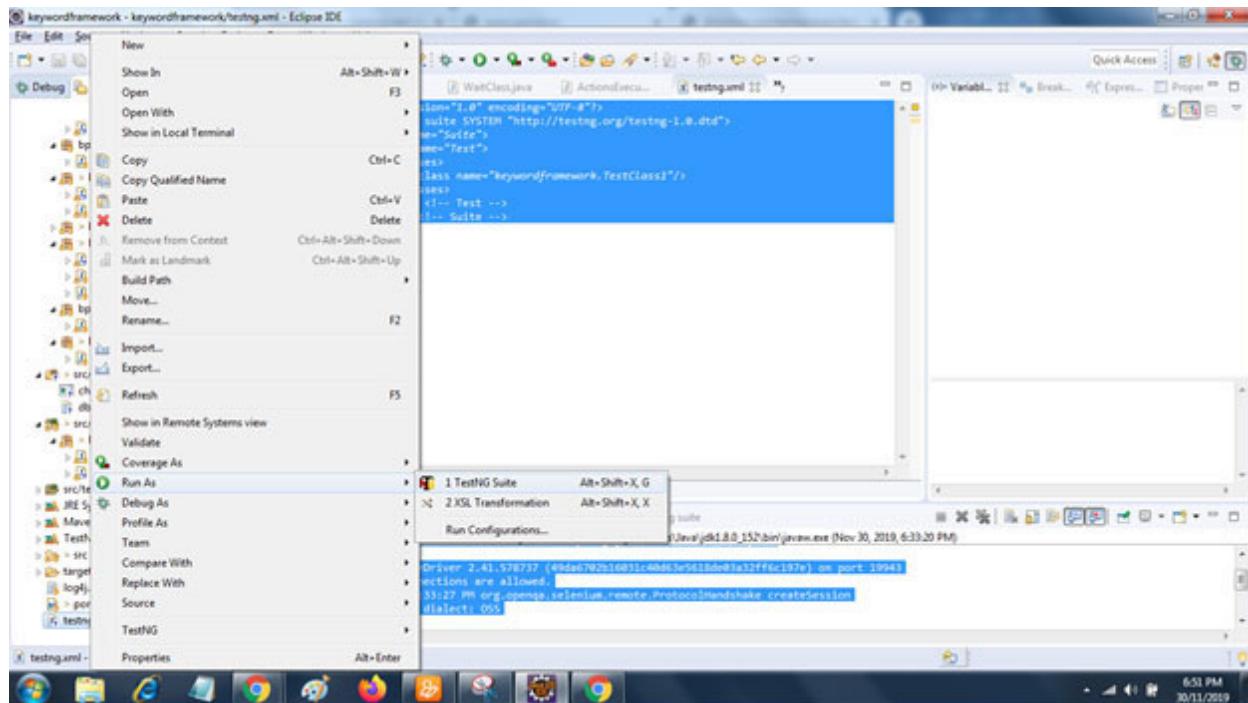
```
@AfterTest  
public void tearDown() {
```

```
actKeywords.clickElement(logoutParent);  
actKeywords.clickElement(btnLogout);  
actKeywords.closeBrowser();  
}  
}
```

Notice that we have added a method for waiting implicitly called `waitImplicitly()` in the `WaitClass` class as shown below

```
public void waitImplicitly() {  
SingletonDriver.getInstance().getDriver().manage().timeouts().implicitly  
Wait(60, TimeUnit.SECONDS);  
}
```

Right click on the `testng.xml` and click **Run As | TestNG**



**Figure 8.10**

## OUTPUT:

A chrome browser gets opened, navigation takes place to the FreeCRM website, login takes place followed by logout and eventually the browser closes.

Notice that we have used the annotations @Test, @BeforeTest, and @AfterTest. These are TestNG annotations, which we will see next.

## Learning TestNG annotations

Let's go through the annotations available in TestNG:

BeforeSuite: The method annotated with this annotation runs only once before all tests in the suite run

AfterSuite: The method annotated with this annotation runs only once after all tests in the suite have run

BeforeClass: The method annotated with this annotation runs only once before the first test method in the current class are executed

AfterClass: The method annotated with this annotation runs only once after all test methods in the current class is executed

BeforeTest: The method annotated with this annotation runs before any test method belonging to the classes inside the tag

AfterTest: The method annotated with this annotation runs after all test methods belonging to the classes inside the tag

Test: Marks a class or method as a part of the test

Parameters: Describes how to pass parameters to a test method

Listeners: Defines Listeners for a test class

DataProvider: Identifies a method as supplying data to a test method. The annotated method must return an Object[ ][ ] two-dimensional array, where each Object[ ] can be assigned the parameter list of the test method. The @Test method that wants to fetch data from this DataProvider needs to use a dataProvider name that matches the name of this annotation

BeforeMethod: The method annotated with this annotation runs before each test method

AfterMethod: The method annotated with this annotation runs after each test method

BeforeGroups: The list of groups that the method annotated will run before. This method is guaranteed to run shortly before the first test method that belongs to any of these groups is invoked

AfterGroups: The list of groups that the method annotated will run after. This method is guaranteed to run shortly after the last test method that belongs to any of these groups is invoked

Let's now see how we can pass parameters in our test.

TestNG provides us with two mechanisms through which we can pass parameters to our test. The mechanisms are listed below:

Using testng xml

Using DataProvider

Let's run through parameterization using

## Parameterization using testng.xml

This mechanism is used to pass simple parameters such as string type to the test methods at run time. In order to pass parameters, the annotation @Parameter is used. These parameters are passed through the testng.xml, as shown below. We have defined a parameter at the suite level. Parameter Test1 has access only to the parameter defined at the suite level. Parameter Test2 has access to the parameter defined at suite level as well as paramtest2 parameter that is local to that test. Parameter Test3 has overridden the parameter at the suite level as well as has to access to the which is local to the test.

Given below is the TestNG XML used for this purpose

```
name="Parameter test Suite" verbose="1">
  name="suite-param" value="parameter at suite level" />
  name="Parameter Test1">
    name="keywordframework.ParamTest1">
      name="prameterTest1" />
    name="Parameter Test2">
      name="paramtest2" value="parameter for test 2" />

    name="keywordframework.ParamTest1">
      name="prameterTest2" />
    name="Parameter Test three">
      name="suite-param" value="suite parameter overridden" />
```

```
name="paramtest3" value="parameter for test 3" />
name="keywordframework.ParamTest1">
name="prameterTest3" />
```

The java code given below makes use of this testing.xml.

```
package keywordframework;
import org.testng.annotations.Parameters;
import org.testng.annotations.Test;

public class ParamTest1 {
    @Parameters({ "suite-param" })
    @Test

    public void prameterTest1(String param) {
        System.out.println("Test one suite parameter is: " + param);
    }
    @Parameters({ "paramtest2" })
    @Test
    public void prameterTest2(String param) {
        System.out.println("Test two parameter is: " + param);
    }

    @Parameters({ "suite-param", "paramtest3" })
    @Test
    public void prameterTest3(String param, String param2) {
        System.out.println("Test three suite parameter is: " + param);
        System.out.println("Test three parameter is: " + param2);
    }
}
```

**OUTPUT:**

```
Test one suite parameter is: parameter at suite level
Test two parameter is: parameter for test 2
Test three suite parameter is: suite parameter overridden
Test three parameter is: parameter for test 3

=====
Parameter test Suite
Total tests run: 3, Passes: 3, Failures: 0, Skips: 0
=====
```

***Figure 8.11***

Let's understand how to use Data Providers next.

## Data Providers

An important feature provided by TestNG is the DataProvider feature. It helps in writing data-driven tests, which essentially means that the same method can be run multiple times with different data-sets.

Please note that @DataProvider is the second way of passing parameters to test methods. It helps in providing complex parameters to the test methods, as it is not possible to do this from XML.

To use the DataProvider feature in the tests, you have to declare a method annotated by @DataProvider and then use the given method in the test method using thedataProvider attribute in the @Test annotation.

The TestNG XML used for this purpose is given below:

```
version="1.0" encoding="UTF-8"?>
suite SYSTEM "http://testng.org/testng-1.0.dtd">
name="Suite">
class-name="bpb.listeners.MyListener2"/>
name="Test">
name="bpb.dataprovider.KeywordProvider"/>
```

Once this suite file is executed, along with the other outputs, we get the output listed below.

## OUTPUT:

```
TestNG default output directory = C:\Users\Bhagyashree\git\BPB_Publication\keywordframework
\test-output\suite

TestNG has invoked methods = [KeywordProvider.testKeywords(java.lang.String, java.lang.String,
java.lang.String, java.lang.String, java.lang.String)[pri:0,
instance:bpb.dataprovider.KeywordProvider@3cef309d]SmokeTest1 TS001 openbrw null null
1022308509, KeywordProvider.testKeywords(java.lang.String, java.lang.String, java.lang.String,
java.lang.String, java.lang.String)[pri:0, instance:bpb.dataprovider.KeywordProvider@3cef309d]
SmokeTest1 TS002 navigate null null 1022308509, KeywordProvider.testKeywords(java.lang.String,
java.lang.String, java.lang.String, java.lang.String, java.lang.String)[pri:0,
instance:bpb.dataprovider.KeywordProvider@3cef309d]SmokeTest1 TS003 enterText //*[@id='ABC']
ABC 1022308509, KeywordProvider.testKeywords(java.lang.String, java.lang.String,
java.lang.String, java.lang.String, java.lang.String)[pri:0,
instance:bpb.dataprovider.KeywordProvider@3cef309d]SmokeTest1 TS004 enterText //*[@id='DEF']
DEF 1022308509, KeywordProvider.testKeywords(java.lang.String, java.lang.String,
java.lang.String, java.lang.String, java.lang.String)[pri:0,
instance:bpb.dataprovider.KeywordProvider@3cef309d]SmokeTest1 TS005 enterText //*[@id='GHI']
GHI 1022308509, KeywordProvider.testKeywords(java.lang.String, java.lang.String,
java.lang.String, java.lang.String, java.lang.String)[pri:0,
instance:bpb.dataprovider.KeywordProvider@3cef309d]SmokeTest1 TS006 click //*[@id='JKL'] JKL
1022308509]
```

*Figure 8.13*

## *ITestListener interface*

The working of this listener is exactly the same as the only difference being; it makes the call before and after the Test and not the Suite. It has a total seven methods in it.

**onFinish():** Invoked after all the tests have run and the corresponding annotated methods have been called.

**onStart():** Invoked after the test class is instantiated and before any annotated method is called.

Invoked each time a method fails but has been annotated with successPercentage, and this failure keeps it within the success percentage.

**onTestFailure(ITestResult result):** Invoked each time when a test fails.

**onTestSkipped(ITestResult result):** Invoked each time when a test is skipped

**onTestStart(ITestResult result):** Invoked each time before a test is about to be invoked.

onTestSuccess(ITestResult result): Invoked each time a test is successful.

Let's have a look at an implemented

```
public class MyListener2 implements ITestListener{

    @Override
    public void onTestStart(ITestResult result) {
        System.out.println("Test Started. "+result.getStartMillis());
    }

    @Override
    public void onTestSuccess(ITestResult result) {
        System.out.println("Test Success. "+result.getEndMillis());
    }

    @Override
    public void onTestFailure(ITestResult result) {
        System.out.println("Test Failed. "+result.getTestName());
    }

    @Override
    public void onTestSkipped(ITestResult result) {
        System.out.println("Test Skipped. "+result.getTestName());
    }

    @Override
```

```
public void onTestFailedButWithinSuccessPercentage(ITestResult result) {  
}  
}
```

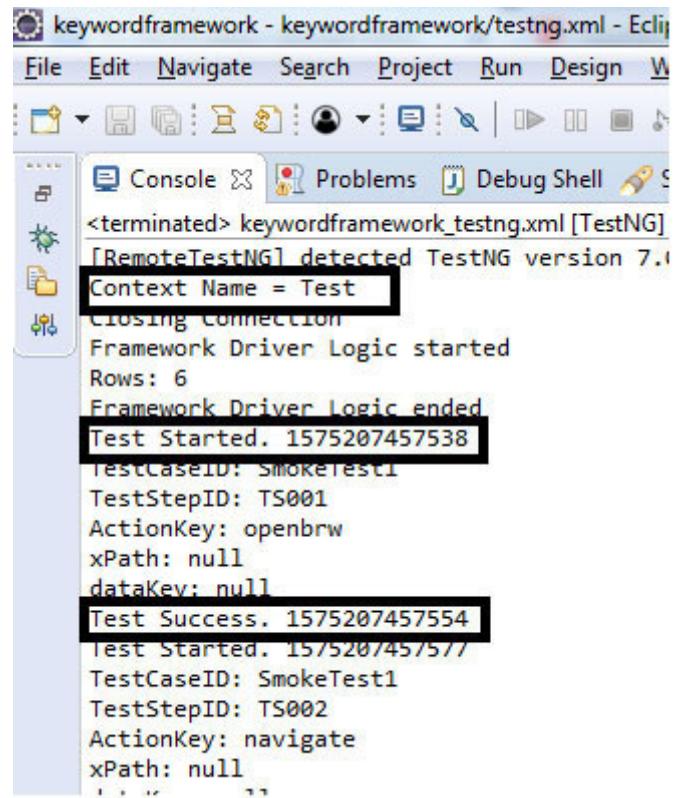
```
@Override  
public void onStart(ITestContext context) {  
    System.out.println("Context Name = "+context.getName());  
}
```

```
@Override  
public void onFinish(ITestContext context) {  
    System.out.println(context.getPassedTests());  
}  
}
```

Let's change the TestNG XML to use this listener:

```
version="1.0" encoding="UTF-8"?>  
suite SYSTEM "http://testng.org/testng-1.0.dtd">  
name="Suite">  
class-name="bpb.listeners.MyListener2"/>  
  
name="Test">  
name="bpb.dataprovider.KeywordProvider"/>
```

When this testing.xml is run, apart from the normal output, we get the output shown below:



The screenshot shows the Eclipse IDE interface with the title bar "keywordframework - keywordframework/testng.xml - Eclipse IDE for Java Developers". The menu bar includes File, Edit, Navigate, Search, Project, Run, Design, Window, Help, and a toolbar with various icons. The left sidebar has icons for file, folder, and search. The central area is the "Console" tab, which displays the following log output:

```
<terminated> keywordframework_testng.xml [TestNG]
[RemoteTestNG] detected TestNG version 7.0
Context Name = Test
closing connection
Framework Driver Logic started
Rows: 6
Framework Driver logic ended
Test Started. 1575207457538
TestCaseID: SmokeTest1
TestStepID: TS001
ActionKey: openbrw
xPath: null
dataKey: null
Test Success. 1575207457554
Test Started. 1575207457577
TestCaseID: SmokeTest1
TestStepID: TS002
ActionKey: navigate
xPath: null
... .. "
```

*Figure 8.14*

With this we come to an end of this chapter.

## Conclusion

This chapter covered the popular TestNG framework used in test automation. We covered a lot of important concepts like TestNG XML, TestNG Suites, TestNG Parameters. We went over the DataProvider annotation and learned how we could extract data from the CachedRowSet object.

The next chapter will explore the parallel execution. We will run our test cases in three groups. Moreover, we will understand the concept of TestNG groups and group test suites with respect to GroupA, GroupB, and We will make changes to the TestConfig table to incorporate parallel execution.

## Questions

Explain the difference between TestNG class and TestNG Test

Write a program to execute three tests in serial fashion

What is the use of Parameters in TestNG?

Write a program using DataProvider

Explain the difference between ISuiteListener and ITestListener

*Parallel Execution*

Parallel test execution involves running test automation in parallel rather than sequentially. The tester need not execute your tests one by one. Instead of that, the entire batch can be split up into multiple groups, and each group can be executed in parallel. This technique has advantages that can effectively reduce the test execution time and improve your overall software development life cycle.

First, running your tests in parallel significantly decreases the test execution time. If one has enough resources to run all the tests at once, then the duration of the test execution will last only as long as the slowest running test. For example, if you have 100 tests and 100 virtual machines on which to run these tests, then dividing these tests in parallel will allow running 100 tests at once. If your slowest test among the set of 100 takes two minutes to run, then that is how long it will take for the entire test suite to complete execution.

Once the reader completes reading this chapter, he/she will have sufficient knowledge to setup a test suite that will have tests that execute in parallel.

## Structure

Here's what we will learn in this chapter.

Introduction to TestNG groups

Learn how to execute tests in a test suite in serial fashion

Get introduced to various options available for parallel execution in TestNG XML

Creating three tests which execute the same class but with different parameters

Fetching test cases from a database based on these parameters

## Objectives

Understand TestNG groups

Create a test suite with three tests with three groups

Execute tests in the test suite in serial fashion

Execute the same tests in parallel by adding configuration in TestNG XML

We begin by first understanding what TestNG Groups are.

## What are TestNG groups?

Grouping of test methods is possible in TestNG. TestNG can be invoked and asked to include a certain set of groups (or regular expressions) and excluding another. This gives the tester maximum flexibility in partitioning the tests on a need basis. Groups help in grouping test sets based on sanity, smoke, or regression.

We will start with a basic example of using groups.

## TestNG XML for incorporating groups

In order to use groups, changes need to be made in the TestNG XML. The changes are shown below. Notice that we have different groups in each test. For the first test, the groups annotated with GroupA will be run; for the second test, the groups annotated with GroupB will be run, and so on. We also have a parallel attribute that helps us to run the tests, methods, or classes in parallel.

```
name="Parameter test Suite" verbose="1" parallel="false" thread-
count="3">
    name="suite-param" value="parameter at suite level" />
    name="Group Test1">
        name="GroupA"/>
        name="keywordframework.TestClass1">
        name="Group Test2">
        name="GroupB"/>

    name="keywordframework.TestClass1">
    name="Group Test3">
    name="GroupC"/>

    name="keywordframework.TestClass1">
```

As can be seen, the single test class that we are using here is which is shown below. We have three tests here which have been assigned three groups which are GroupA, GroupB and Whenever TestNG encounters the tag in the TestNG XML, it also sees the groups tag shown below:

```
name="GroupA"/>
```

Seeing this tag, TestNG comes to know which method to execute. It finds the method tagged with the For this particular case, it knows that Test method test1 has to be executed.

```
public class TestClass1 {
```

```
Action_KeywordsactKeywords = new Action_Keywords();
WaitClasswdWait = null;
String btnLogin = "//a[@href='https://ui.freecrm.com']";
String uNameXPath = "//input[@name='email']";
String passwordXPath = "//input[@name='password']";
String btnSubmit = "//div[text()='Login']";
String logoutParent = "//div[@class='ui buttons']/div";
String btnLogout ="//span[text()='Log Out']";
String callQueue = "//span[text()='Call Queue']";
```

```
public TestClass1() {
System.out.println("Constructor called");
}
```

```
@BeforeTest(groups= {"GroupA","GroupB","GroupC"})
```

```
public void setUp() {  
    actKeywords.openBrowser("chrome");  
    wdWait = new WaitClass();  
    wdWait.waitImplicitly();  
}  
  
  
@Test(groups= "GroupA")  
public void test1() {  
    System.out.println("Before starting test1");  
    actKeywords.navigateURL("http://www.freecrm.com");  
    actKeywords.clickElement(btnLogin);  
  
    actKeywords.enterText(uNameXPath, "chaubalpinakin@gmail.com");  
    actKeywords.enterText(passwordXPath, "Pc9121975!");  
    actKeywords.clickElement(btnSubmit);  
    System.out.println("After completing test1");  
}  
  
  
@Test(groups= "GroupB")  
public void test2() {  
    System.out.println("Before starting test2");  
    actKeywords.navigateURL("http://www.freecrm.com");  
    actKeywords.clickElement(btnLogin);  
    actKeywords.enterText(uNameXPath, "chaubalpinakin@gmail.com");  
    actKeywords.enterText(passwordXPath, "Pc9121975!");  
    actKeywords.clickElement(btnSubmit);  
    System.out.println("After completing test2");  
}  
  
  
@Test(groups= "GroupC")  
public void test3() {
```

```
System.out.println("Before starting test3");
actKeywords.navigateURL("http://www.freecrm.com");
actKeywords.clickElement(btnLogin);
actKeywords.enterText(uNameXPath, "chaubalpinakin@gmail.com");
actKeywords.enterText(passwordXPath, "Pc9121975!");
actKeywords.clickElement(btnSubmit);
System.out.println("After completing test3");
}
```

```
@AfterTest(groups= {"GroupA","GroupB","GroupC"})
public void tearDown() {
```

```
    actKeywords.clickElement(logoutParent);
    actKeywords.clickElement(btnLogout);
    actKeywords.closeBrowser();
}
```

Right-click on the TestNG XML and click **Run As |TestNG** The three methods get executed one after the other in a sequential manner.

#### **OUTPUT:**

```

<terminated> keywordframework_testng2.xml [TestNG] C:\Program Files\Java\jdk1.8.0_152\bin\javaw.exe (Dec 14, 2019, 10:38:45 AM)
[RemoteTestNG] detected TestNG version 7.0.0
[TestNGContentHandler] [WARN] It is strongly recommended to add "<!DOCTYPE suite SYSTEM \"https://testng.org/testng-1.0.dtd\">" at the top of your file, otherwise TestNG may fail or not work correctly
[log4j]ERROR Could not find value for key log4j.appenders.LOGFILE.layout
Constructor called
[log4j]ERROR Could not find value for key log4j.appenders.LOGFILE.layout
Constructor called
[log4j]ERROR Could not find value for key log4j.appenders.LOGFILE.layout
Constructor called
Starting ChromeDriver 2.41.578737 (49da6702b16031c40d63e5618de03a32ff6c197e) on port 29262
Only local connections are allowed.
Session is fd957372469181784ec0a8215ad7e28c
Dec 14, 2019 10:38:52 AM org.openqa.selenium.remote.ProtocolHandshake createSession
INFO: Detected dialect: OSS
Before starting test1
After completing test1
Starting ChromeDriver 2.41.578737 (49da6702b16031c40d63e5618de03a32ff6c197e) on port 32648
Only local connections are allowed.
Session is 49a8fbadf1ac6162b192e60a7b744e026
Dec 14, 2019 10:39:12 AM org.openqa.selenium.remote.ProtocolHandshake createSession
INFO: Detected dialect: OSS
Before starting test2
After completing test2
Starting ChromeDriver 2.41.578737 (49da6702b16031c40d63e5618de03a32ff6c197e) on port 19219
Only local connections are allowed.
Session is 59bc1df4abc7e2218f33a9e89a733a16
Dec 14, 2019 10:39:44 AM org.openqa.selenium.remote.ProtocolHandshake createSession
INFO: Detected dialect: OSS
Before starting test3
After completing test3

=====
Parameter test Suite
Total tests run: 3, Passes: 3, Failures: 0, Skips: 0
=====
```

**Figure 9.1**

Note that since we used parallel="false", the three tests executed in a serial fashion. Notice that SessionIds is getting printed in the console. This happens because we have extracted the SessionId in the code snippet below. This code snippet is the openBrowser method. It is very important to cast the to RemoteWebDriver and then surround the entire code with round brackets. Only after this, you will get the getSessionId() method. It can be seen from the console output that the three extracted session ids are different for the three browser sessions that get opened.

```

public void openBrowser(String browser) {

    try {
        SingletonDriver.getInstance().setDriver("chrome", "windows",
        "windows");
        SingletonDriver.getInstance().getDriver();
        SessionId session =
        ((RemoteWebDriver)SingletonDriver.getInstance().getDriver()).getSessi
        onId();
```

```
System.out.println("Session is "+session);
SingletonDriver.getInstance().getDriver().manage().window().maximize
();
} catch (Exception e) {
e.printStackTrace();
}
}
```

Next, let's get a feel of parallel execution

## Introduction to parallel execution

Parallel testing can be accomplished by making changes in the TestNG XML for the parallel attribute in the suite tag. The parallel attribute can take the values mentioned below:

**Tests:** All test cases in the tag will be executed in parallel

**Classes:** All test cases inside a Java class run in parallel

**Methods:** all methods annotated with @Test run in parallel

**Instances:** Test cases in the same instance will execute in parallel, but two methods of two different instances will run in a different thread.

We will change our TestNG XML to have Right-click on the TestNG XML and click **Run As | TestNG** Now it is seen that three browser sessions open at the same time, and execution takes place parallelly in the three browser sessions. The following gets output to the console:

```
<terminated> keywordframework_testing2.xml [TestNG] C:\Program Files\Java\jdk1.8.0_152\bin\javaw.exe (Dec 14, 2019, 11:11:37 AM)
[RemoteTestNG] detected TestNG version 7.0.0
[TestNGContentHandler] [WARN] It is strongly recommended to add "<!DOCTYPE suite SYSTEM "https://testing.org/testng-1.0.dtd">" at the top of your file, otherwise TestNG may fail or n
log4j:ERROR Could not find value for key log4j.appenders.LOGFILE.layout
Constructor called
log4j:ERROR Could not find value for key log4j.appenders.LOGFILE.layout
Constructor called
Constructor called
log4j:ERROR Could not find value for key log4j.appenders.LOGFILE.layout
Starting ChromeDriver 2.41.578737 (49da6702b16031c40d63e5618de03a32ff6c197e) on port 45120
Only local connections are allowed.
Starting ChromeDriver 2.41.578737 (49da6702b16031c40d63e5618de03a32ff6c197e) on port 5520
Only local connections are allowed.
Starting ChromeDriver 2.41.578737 (49da6702b16031c40d63e5618de03a32ff6c197e) on port 14982
Only local connections are allowed.
Dec 14, 2019 11:11:28 AM org.openqa.selenium.remote.ProtocolHandshake createSession
INFO: Detected dialect: OSS
Session is 08ec78372c57eb2a4275d72cce1320f7
Before starting test1
Dec 14, 2019 11:11:28 AM org.openqa.selenium.remote.ProtocolHandshake createSession
INFO: Detected dialect: OSS
Session is b842b30f509cf55d7f12ced2ebbf302
Dec 14, 2019 11:11:28 AM org.openqa.selenium.remote.ProtocolHandshake createSession
INFO: Detected dialect: OSS
Session is b2a224be0b1b184f87020bc60c6a4de4f
Before starting test3
Before starting test2
After completing test2
After completing test1
After completing test3

=====
Parameter test Suite
Total tests run: 3, Passes: 3, Failures: 0, Skips: 0
=====
```

*Figure 9.2*

Let's integrate the tests with the database.

## Changes to the database for parallel execution

Add a column called Group to the TestConfig table as shown below:

Column Name	Datatype
TestCaseID	VARCHAR(10)
Description	VARCHAR(20)
Execute	VARCHAR(1)
Group	VARCHAR(8)

**Figure 9.3**

The TestNG xml is shown below:

```
name="Parameter test Suite" verbose="1" parallel="tests" thread-
count="3">
  name="suite-param" value="parameter at suite level" />
  name="Group Test1">
    name="GroupA"/>
    name="param1" value="GroupA" />

  name="keywordframework.TestClass1">

  name="Group Test2">
    name="GroupB"/>
```

```
name="param2" value="GroupB" />
name="keywordframework.TestClass1">
name="Group Test3">
name="GroupC"/>
name="param3" value="GroupC" />
name="keywordframework.TestClass1">
```

Change the DBExtract3 code as shown below. Notice that in the code below we are passing the group as a TestNG parameter. With the help of this group we are extracting records only for that group from the database

```
public class DBExtract3 {

    private static String pageID;
    private static String objectID;
    private static String xPath;
    private static Connection conn;
    private static CachedRowSet cachedRowset;
    private static Statement stmt;

    @SuppressWarnings("finally")
    public static CachedRowSet extractRecords(String group) {
        ResultSet resultSet = null;
        ResultSetMetaData rsmd = null;
        Properties prop = new Properties();
        InputStream iStream = null;
        String dbURL=null;
        String dbUName=null;
        String dbPwd=null;
        System.out.println("Group is: "+group);
```

```

/*****Create Connection, Statement and Resultset objects and
execute SQL Query and fetch metadata*****
try {
iStream=DBExtract2.class.getClassLoader().getResourceAsStream("dbc
onfig.properties");
prop.load(iStream);
dbURL=prop.getProperty("DB_URL");
dbUName=prop.getProperty("DB_UNAME");
dbPwd=prop.getProperty("DB_PASSWORD");
conn = DriverManager.getConnection(
dbURL, dbUName,
dbPwd);
stmt = conn.createStatement();

resultSet = stmt
.executeQuery("select
A.TestCaseID,A.TestStepID,A.ActionKey,B.XPath,C.DataKey from
testcases A left outer join object_repository B on A.ORID=B.ORID
left outer join testdata C on A.TestCaseID=C.TestCaseID and
A.TestStepID=C.TestStepID where A.TestCaseID in (select TestCaseID
from testconfig TC where TC.Execute='Y' and
TC.Group="""+group+""") order by A.TestStepID asc");
rsmd = resultSet.getMetaData();

RowSetFactory rsFactory = RowSetProvider.newFactory();
cachedRowset = rsFactory.createCachedRowSet();
cachedRowset.populate(resultSet);
} catch (Exception e) {
// TODO Auto-generated catch block
e.printStackTrace();
} finally {
try {

```

```

System.out.println("Closing Connection");
stmt.close();
resultSet.close();

conn.close();
} catch (SQLException e) {
// TODO Auto-generated catch block
e.printStackTrace();
}

return cachedRowset;

}
}
}

```

The TestClass that gets executed is shown below:

```

public class TestClass1 {

Action_Keywords actKeywords = new Action_Keywords();
WaitClass wdWait = null;

public TestClass1() {
System.out.println("Constructor called");
}

@BeforeTest(groups= {"GroupA","GroupB","GroupC"})

```

```

public void setUp() {
    actKeywords.openBrowser("chrome");
    wdWait = new WaitClass();
    wdWait.waitImplicitly();
}

@Test(groups= "GroupA")
@Parameters({ "param1" })
public void test1(String param1) {
    System.out.println("Before starting test1");
    ResultSet resultSet = null;

    try {
        CachedRowSet crs = DBExtract3.extractRecords("GroupA");
        ResultSetMetaData rsmd = crs.getMetaData();
        crs.next();
        while (!crs.isAfterLast()) {
            String testcaseid = crs.getString("TestCaseID");
            String teststepid = crs.getString("TestStepID");

            String actionKey = crs.getString("ActionKey");
            System.out.println("ActionKey"+actionKey);
            String xpath = crs.getString("XPath");
            System.out.println("XPath"+xpath);

            String datakey = crs.getString("DataKey");
            System.out.println("DataKey"+datakey);
            switch(actionKey) {
                case "navigate":
                    actKeywords.navigateURL("http://www.freecrm.com");
                    break;
            }
        }
    } catch (Exception e) {
        e.printStackTrace();
    }
}

```

```

case "click":
actKeywords.clickElement(xpath);
break;
case "enterText":
actKeywords.enterText(xpath, datakey);
break;
}

crs.next();
}
} catch (SQLException e) {
e.printStackTrace();
}
System.out.println("After completing test1");
}

@Test(groups= "GroupB")
@Parameters({ "param2" })
public void test2(String param2) {
System.out.println("Before starting test2");
ResultSet resultSet = null;

try {
CachedRowSet crs = DBExtract3.extractRecords("GroupB");
ResultSetMetaData rsmd = crs.getMetaData();
//System.out.println("Framework Driver Logic started");

crs.next();
while (!crs.isAfterLast()) {
String testcaseid = crs.getString("TestCaseID");
String teststepid = crs.getString("TestStepID");

```

```
String actionKey = crs.getString("ActionKey");
String xpath = crs.getString("XPath");
String datakey = crs.getString("DataKey");
switch(actionKey) {
case "navigate":
actKeywords.navigateURL("http://www.freecrm.com");
break;
case "click":
actKeywords.clickElement(xpath);
break;
case "enterText":
actKeywords.enterText(xpath, datakey);
break;
}
```

```
crs.next();
}
} catch (SQLException e) {
e.printStackTrace();
}
```

```
System.out.println("After completing test2");
}
@Test(groups= "GroupC")
@Parameters({ "param3" })
public void test3() {
System.out.println("Before starting test3");
```

```
ResultSet resultSet = null;
```

```
try {
```

```

CachedRowSet crs = DBExtract3.extractRecords("GroupC");
ResultSetMetaData rsmd = crs.getMetaData();
crs.next();
while (!crs.isAfterLast()) {
    String testcaseid = crs.getString("TestCaseID");
    String teststepid = crs.getString("TestStepID");
    String actionKey = crs.getString("ActionKey");
    String xpath = crs.getString("XPath");
    String datakey = crs.getString("DataKey");
    switch(actionKey) {
        case "navigate":
            actKeywords.navigateURL("http://www.freecrm.com");
            break;
        case "click":
            actKeywords.clickElement(xpath);
            break;
        case "enterText":
            actKeywords.enterText(xpath, datakey);
            break;
    }
    crs.next();
}
} catch (SQLException e) {
    e.printStackTrace();
}
System.out.println("After completing test3");
}

@AfterTest(groups= {"GroupA","GroupB","GroupC"})

```

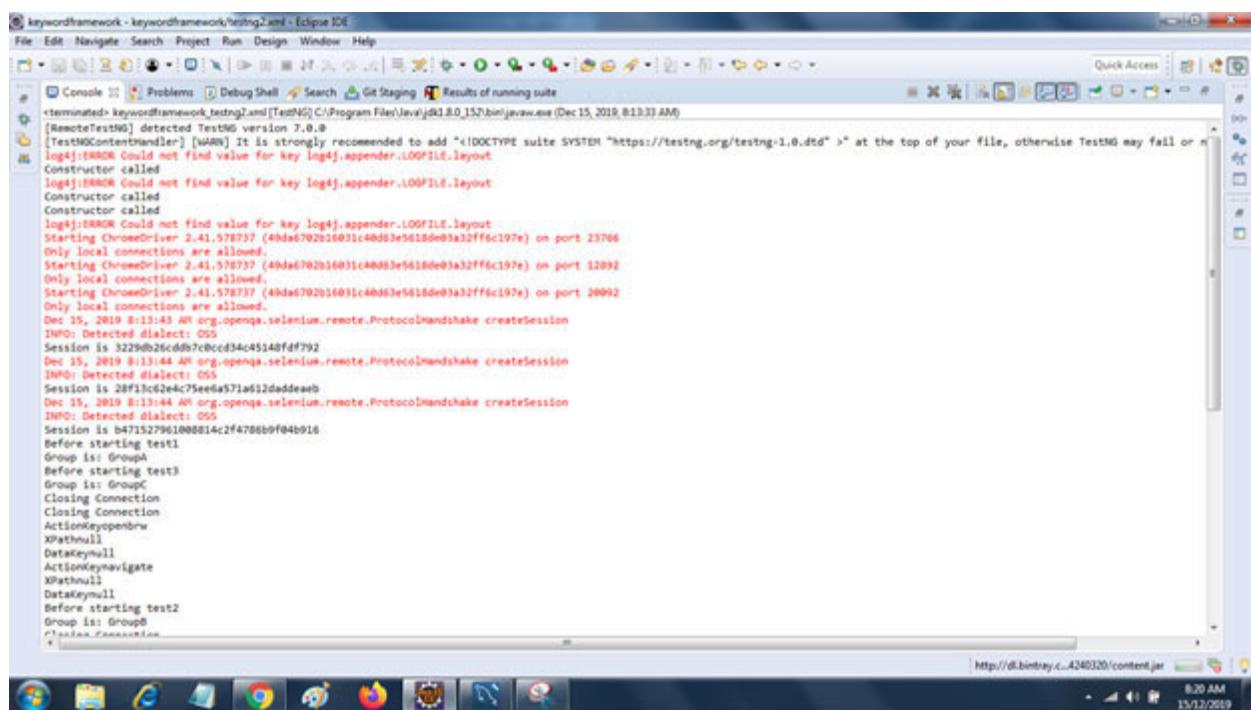
```

public void tearDown() {
    actKeywords.closeBrowser();
}
}

```

## OUTPUT:

Three chrome windows open at the same time, and three tests get executed parallelly. This time the keywords are pulled from the database based on the group passed as a parameter from The console output is shown below:



The screenshot shows the Eclipse IDE interface with the 'Console' tab selected. The output window displays the following log entries:

```

keywordframework - keywordframework/testing2.xml - Eclipse IDE
File Edit Navigate Search Project Run Design Window Help
Console Problems Debug Shell Search Get Staging Results of running suite
<terminated> keywordframework.testing2.xml [TestNG] C:\Program Files\Java\jdk-8.0_157\bin\java.exe (Dec 15, 2019, 8:13:33 AM)
[RemoteTestNG] detected TestNG 7.0.0
[TestNGContentHandler] [WARN] It is strongly recommended to add "<!DOCTYPE suite SYSTEM "https://testng.org/testng-1.0.dtd" >" at the top of your file, otherwise TestNG may fail or n
log4j:WARN Could not find value for key log4j:appender.LOGFILE.layout
Constructor called
log4j:WARN Could not find value for key log4j:appender.LOGFILE.layout
Constructor called
Constructor called
log4j:WARN Could not find value for key log4j:appender.LOGFILE.layout
Starting ChromeDriver 2.41.578737 (49da6702816031c40dd63e5d18de03a32ff6c197e) on port 23766
Only local connections are allowed.
Starting ChromeDriver 2.41.578737 (49da6702816031c40dd63e5d18de03a32ff6c197e) on port 12892
Only local connections are allowed.
Starting ChromeDriver 2.41.578737 (49da6702816031c40dd63e5d18de03a32ff6c197e) on port 20992
Only local connections are allowed.
Dec 15, 2019 8:13:43 AM org.openqa.selenium.remote.ProtocolHandshake createSession
INFO: Detected dialect: OSS
Session is 32290b26cd97c0cd34c45148fd792
Dec 15, 2019 8:13:44 AM org.openqa.selenium.remote.ProtocolHandshake createSession
INFO: Detected dialect: OSS
Session is 28f13c62ec75eeda571a612daddeseb
Dec 15, 2019 8:13:44 AM org.openqa.selenium.remote.ProtocolHandshake createSession
INFO: Detected dialect: OSS
Session is b471527961088814c2f4786b9f04b916
Before starting test1
Group is: GroupA
Before starting test3
Group is: GroupC
Closing Connection
Closing Connection
ActionKeyOpenbrw
XPathnull
Datakeynull
ActionKeyNavigate
XPathnull
Datakeynull
Before starting test2
Group is: GroupB
C:/andrea/Downloads

```

The status bar at the bottom right shows the URL <http://dl.bintray.com/4240320/content.jar>, the time 8:20 AM, and the date 15/12/2019.

*Figure 9.4*

```
<terminated> keywordframework_testng2.xml [TestNG] C:\Program Files\Java\jdk1.8.0_152\bin\java.exe (Dec 15, 2019, 8:13:37 AM)
Closing CONNECTION
ActionKeyopenbrw
XPathnull
DataKeynull
ActionKeynavigate
XPathnull
DataKeynull
Before starting test2
Group is: Group0
Closing Connection
ActionKeyclick
XPath//a[@href="https://ui.freecrm.com"]
DataKeynull
ActionKeyenterText
XPath//input[@name='email']
DataKeychubalpnaik@gmail.com
ActionKeyenterText
XPath//input[@name='password']
DataKeychubalpnaik751
ActionKeyclick
XPath//div[text()='Login']
DataKeynull
ActionKeyclick
XPath//div[@class='ui_buttons']/div
DataKeynull
After completing test3
ActionKeyclick
XPath//span[text()='Log Out']
DataKeynull
After completing test2
After completing test1
After completing test0
=====
Parameter test Suite
Total tests run: 1, Passes: 1, Failures: 0, Skips: 8
```

**Figure 9.5**

Notice the three different session ids for three sessions.

This completes the chapter on parallel execution

## Conclusion

This chapter covered the concept of TestNG groups and parallel execution using ThreadLocal and parallel attribute in the Suite tag of We first saw serial execution, and later we explored parallel execution. With parallel execution, we can save test execution time to a great extent. The maximum time required for a test suite execution will be the time taken by the slowest running test.

The next chapter will make the reader understand Maven and the POM.xml in detail. We will learn how to create a Maven build and execute the build from the command prompt. This will later help us in working with Jenkins when we create a Jenkins job to trigger the test execution.

## Questions

What is the use of groups in TestNG?

Practice using the various options available with the parallel attribute of the Suite tag of TestNG.xml

Write a program to execute three tests in parallel

Write a program to replace the looping logic in the individual tests with a data provider

Explain the usage of the tag in TestNG groups.

**Understanding Maven**

We are going to learn Maven as a build tool and understand the build process in Maven. We will be building our project through Eclipse and through Command prompt. We will be studying Assertions for verifying texts and attributes on web pages. We will understand the POM.xml , which is the heart of Maven. We will understand the Fork tag used to facilitate parallel execution.

This is a very important chapter from the point of view of learning the build process, which helps us in executing the project from Jenkins. This build can be automated so that it can run in the off-time when none of the testers or developers are using the AUT. This way, we can utilize the power of Maven and Jenkins to smoothen the build and execute process.

After reading this chapter, the reader will be conversant with the build process in Maven. The reader will also understand how to execute the Eclipse project from a command prompt.

## Structure

Here's what we will learn in this chapter:

Introduction to Maven

Setting-up Maven

Maven build lifecycle

Maven command line calls

Goals in Maven

Packaging

Plug-ins

Triggering tests from TestNG xml

Using dataprovider instead of For loops

Using assertions

Incorporating extent reports

## Introduction to Git and GitHub

## Objectives

Maven Set-up

Build a life cycle in Maven

Understand goals and plug-ins in Maven

Trigger Maven from the Command Line

Creating Maven build

Understanding assertions

Learning to use extent reports

Integrating the project with Git and GitHub

We begin with an introduction to Maven.

## [Introducing Maven](#)

Apache Maven is a popular tool for build automation, mainly for Java projects. Maven takes into account two facets of building software projects, which are the build procedure and the required dependencies. It uses convention over configuration for the build procedure. An XML file called `pom.xml` describes the software project being built, its dependencies on other external components, directories, the build order, and plugins. It has predefined targets to perform certain tasks, such as code compilation and packaging. Maven dynamically downloads Java libraries and Maven plugins from one or more remote repositories, such as the Maven Central Repository, and stores them locally on the hard drive.

We will be using Maven for building and executing our project first from the command prompt and later from Jenkins. We will have a look at adding the project to GitHub. Our code will then download the code from GitHub, build the project, and execute it.

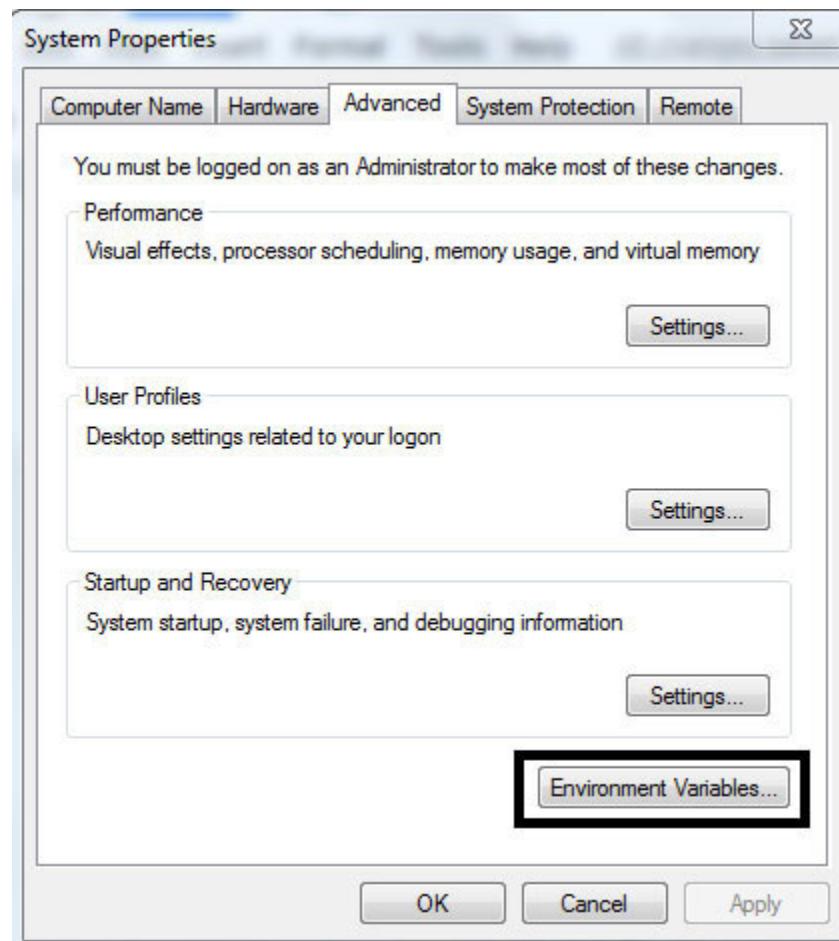
We start with the setup of Maven.

## Setting up Maven

In order to set-up Maven, follow the steps mentioned below

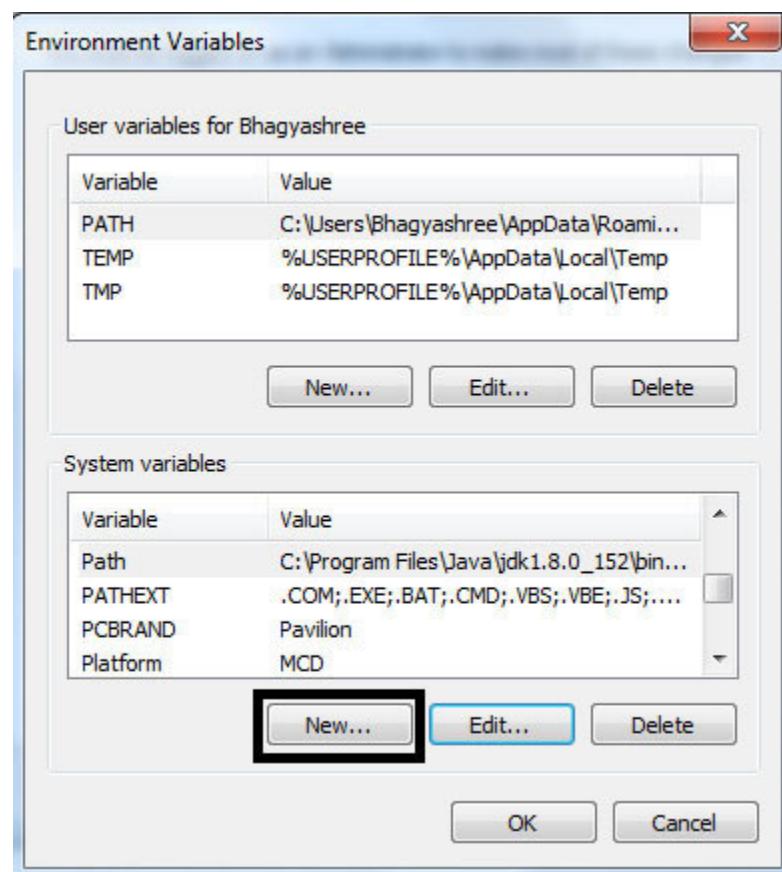
Download Maven from Unzip the file to the folder of your choice.

Open the **System Properties** dialog as shown below and click **Environment**



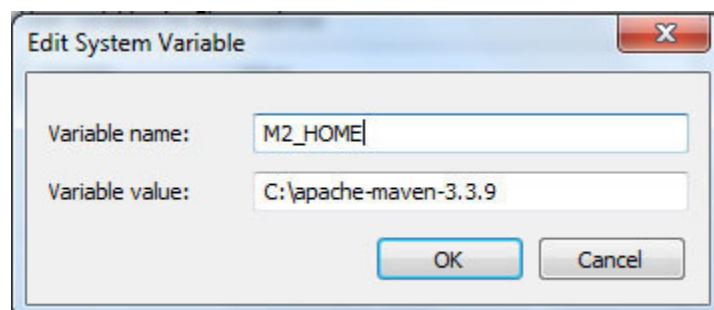
**Figure 10.1**

Click the **New** button as shown below



**Figure 10.2**

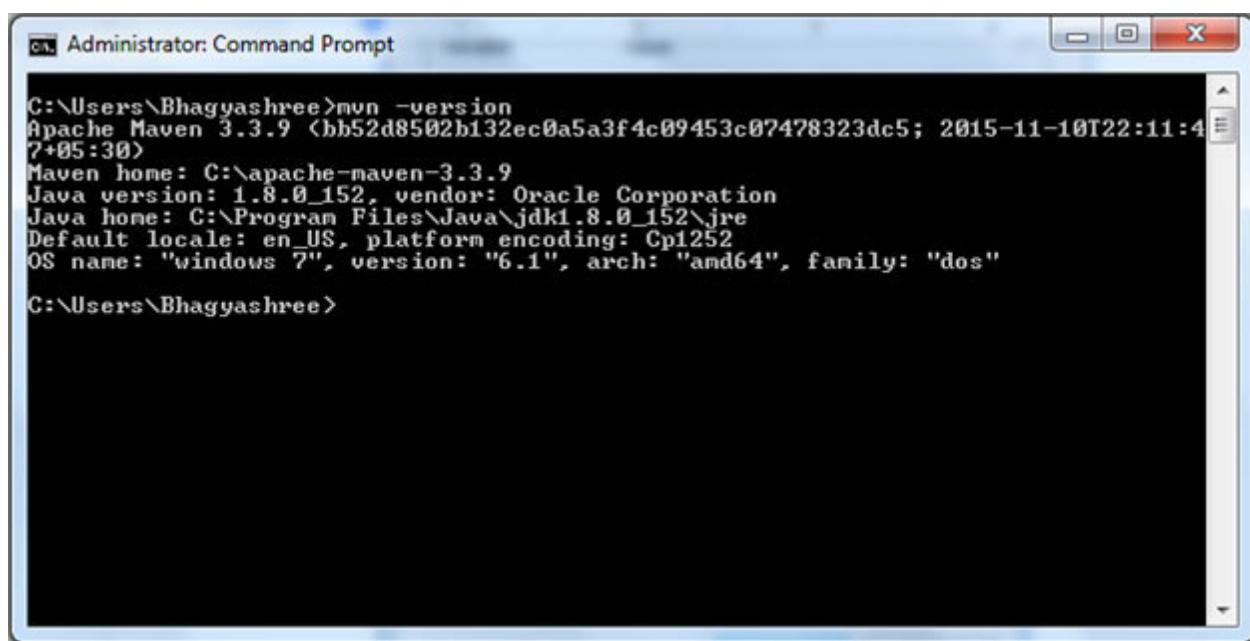
Add a new system variable M2\_HOME as shown below



**Figure 10.3**

Append the path system variable with C:\apache-maven-3.3.9\bin

Check your Maven installation by typing mvn -version at the command prompt. If the output is displayed as shown below, then Maven has been installed successfully/p>



The screenshot shows a Windows Command Prompt window titled "Administrator: Command Prompt". The command entered is "mvn -version". The output displays the following information:

```
C:\Users\Bhagyashree>mvn -version
Apache Maven 3.3.9 (bb52d8502b132ec0a5a3f4c09453c07478323dc5; 2015-11-10T22:11:47+05:30)
Maven home: C:\apache-maven-3.3.9
Java version: 1.8.0_152, vendor: Oracle Corporation
Java home: C:\Program Files\Java\jdk1.8.0_152\jre
Default locale: en_US, platform encoding: Cp1252
OS name: "windows 7", version: "6.1", arch: "amd64", family: "dos"
C:\Users\Bhagyashree>
```

**Figure 10.4**

Now that we have Maven setup, let's dig deeper into the features of Maven. We begin with the build life cycle of Maven

## Maven build life cycle

Maven is based around the central concept of a build lifecycle. What this means is that the process for building and distributing a particular project is clearly defined.

For the developer building a project, this means that it is only necessary to learn a small set of commands to build any Maven project, and the POM will ensure they get the results they desired.

There are three built-in build lifecycles:

default: The default lifecycle handles your project deployment

clean: The clean lifecycle handles project cleaning

site: The site lifecycle handles the creation of your project's site documentation.

A build life cycle is comprised of phases that we see next.

## Maven phases

Each of these build lifecycles is defined by a different list of build phases, where a build phase represents a stage in the lifecycle.

For example, the default lifecycle is made up of the following phases:

the project is correct, and all necessary information is available

compile: Compiles the source code of the project

test: Test the compiled source code using a unit testing framework, which in our case is TestNG. These tests should not require that the code be packaged or deployed

package: Take the compiled code and package it in its distributable format, such as a JAR

verify: Run any checks on results of integration tests to ensure quality criteria are met

install: Install the package into the local repository, to be used as a dependency in other projects locally

deploy: Done in the built environment, copies the final package to the remote repository for sharing with other developers and projects.

These lifecycle phases (of course, this is just a subset of phases) are executed sequentially to complete the default lifecycle. Given the lifecycle phases above, this means that when the default lifecycle is used, Maven will first validate the project, then will try to compile the sources, run those against the tests, package the binaries (e.g., jar), run integration tests against that package, verify the integration tests, install the verified package to the local repository, then deploy the installed package to a remote repository.

Next, let's look at the command line calls for Maven.

## Maven command line calls

Command-line calls can be used with Maven. Let's have a look at the usual Command-Line Calls

In a development environment, the following call is used to build and install artifacts into the local repository:

```
mvn install
```

This command executes each default life cycle phase in order (validate, compile, package, etc.), before executing install. You only need to call the last build phase to be executed, in this case, install:

In a build environment, the following call is used to cleanly build and deploy artifacts into the shared repository.

```
mvn clean deploy
```

The same command can be used in a multi-module scenario (i.e., a project with one or more subprojects). Maven traverses into every subproject and executes clean, then executes deploy (including all of the prior build phase steps).

Let's explore the concept of plug-in goals next.



## Plugin goals

A build phase is made up of plugin goals.

However, even though a build phase is responsible for a specific step in the build lifecycle, the manner in which the responsibilities are executed may vary. This is accomplished by declaring the plugin goals which are bound to those build phases.

A plugin goal represents a definitive task that takes part in the building and managing of a project. A plugin goal may be bound to zero or more build phases. A goal that is not bound to any build phase can be executed outside of the build lifecycle by invoking directly. The order of execution depends on the order in which the goals and the build phases are called. For example, the clean and package arguments are build phases, while the dependency:copy-dependencies is a goal (of a plugin).

```
mvn clean dependency:copy-dependencies package
```

If this were to be executed, the clean phase would be executed first (all preceding phases of the clean phase will be run, plus the clean phase itself), and then the dependency:copy-dependencies goal will get executed, before finally running the package phase (and all its preceding build phases of the default lifecycle).

Moreover, if a goal is bound to one or more build phases, that goal gets called in all those phases.

Moreover, a build phase can also have zero or more goals attached to it. If a build phase has no goals attached to it, that build phase will not execute. But if it has one or more goals attached to it, all those goals get executed.

**Note that, some phases are not generally invoked from the Command-Line.**

Phases starting with hyphenated-words like pre-\*, or process-\* are not usually directly called from the command line. These phases sequence the build, producing intermediate results that are not useful outside of the build.

## Setting up your project to use the build lifecycle

The build lifecycle is simple enough to use, but when you are constructing a Maven build for a project, how do we assign tasks to each of those build phases?

## Packaging

The first and most common way is to specify how the project will be packaged. The POM element is used for this purpose. Some of the valid packaging values are jar, war, ear, and pom. If no packaging value is specified, it will default to the jar.

Note that for some packaging types to be available, one may also need to include a particular plugin in the section of your POM and specify true for that plugin.

## Plugins

The second way to add goals to phases is to configure plugins in your project. Plugins are artifacts that provide goals to Maven. Furthermore, a plugin may have one or more goals wherein each goal represents the capability of that plugin. For example, the Compiler plugin has two goals: compile and testCompile. The former compiles the main source code, while the latter compiles the test code.

As you will see in the later sections, plugins can contain information that indicates which lifecycle phase to bind a goal to. Note that adding the plugin on its own is not enough - the goals you want to run as part of your build must be specified.

The goals that are configured will be added to the goals already bound to the lifecycle from the packaging. If more than one goal is bound to a particular phase, the order used is that those from the packaging are executed first, followed by those configured in the POM. The element can be used to order particular goals.

Let's have a look at two frequently used plugins.

## [Maven Compiler plugin](#)

The Maven Compiler plugin is used to compile the sources of your project. Since version 3.0, the default compiler is javax.tools.JavaCompiler (if you are using java 1.6) and is used to compile Java sources. If you want to force the plugin using javac, you must configure the plugin option

Also note that at present, the default source setting is 1.6, and the default target setting is 1.6, independently of the JDK you run Maven with. It is highly recommended to change these defaults by setting source and target as described in Setting the and -target of the Java compiler.

## [Maven SureFire plugin](#)

The Surefire Plugin is used during the test phase of the build lifecycle to execute the unit tests of an application. It generates reports in two different file formats:

Plain text files (\*.txt)

XML files (\*.xml)

By default, these files are generated in \${basedir}/target/surefire-reports/TEST-\*.xml.

Let's see how to use the Suite XML files for executing TestNG xml next.

## Using Suite XML files

In order to use TestNG suite XML files, a Surefire plugin with the configuration shown below should be added. This allows the flexible configuration of the tests to be run. These files are created, and then added to the Surefire plugin configuration:

```
org.apache.maven.plugins  
maven-surefire-plugin  
3.0.0-M4  
testng.xml  
3
```

This configuration will override the includes and excludes patterns and run all tests in the suite files.

## Specifying test parameters

Your TestNG test can accept parameters with the @Parameters annotation. You can also pass parameters from Maven into your TestNG test, by specifying them as system properties, like this:

```
org.apache.maven.plugins  
maven-surefire-plugin  
3.0.0-M4  
firefox
```

After adding the Maven Compiler plugin and the Maven Surefire plugin to the right click on the project and select **Maven | Update**

Right-click on the pom.xml and select **Run As | Maven Test**

### **OUTPUT:**

The three tests in the testng2.xml get executed parallelly due to the presence of

## Using the DataProvider to pull in the included group

We will be using the KeywordProvider class, which we created when we studied the DataProvider concept. We will change the method testKeywords to include all three groups, as shown below:

```
@TestdataProvider="keywords", groups=
{"GroupA","GroupB","GroupC"})
public void testKeywords(String testcasID, String teststepID, String
actionKey, String xPath, String dataKey)
xPath, String dataKey) {
System.out.println("TestCaseID: "+testcasID);
System.out.println("TestStepID: "+teststepID);
System.out.println("ActionKey: "+actionKey);
System.out.println("xPath: "+xPath);
System.out.println("dataKey: "+dataKey);
}
```

The DataProvider method with the @DataProvider annotation should be changed as follows

Replace the line CachedRowSet crs =  
DBExtract3.extractRecords("GroupA"); with the three lines shown below:

```
String[] group = itestContext.getIncludedGroups();
System.out.println("Group passed as parameter "+group[0]);
```

```
CachedRowSet crs = DBExtract3.extractRecords(group[o]);
```

The TestNG xml will be changed as shown below:

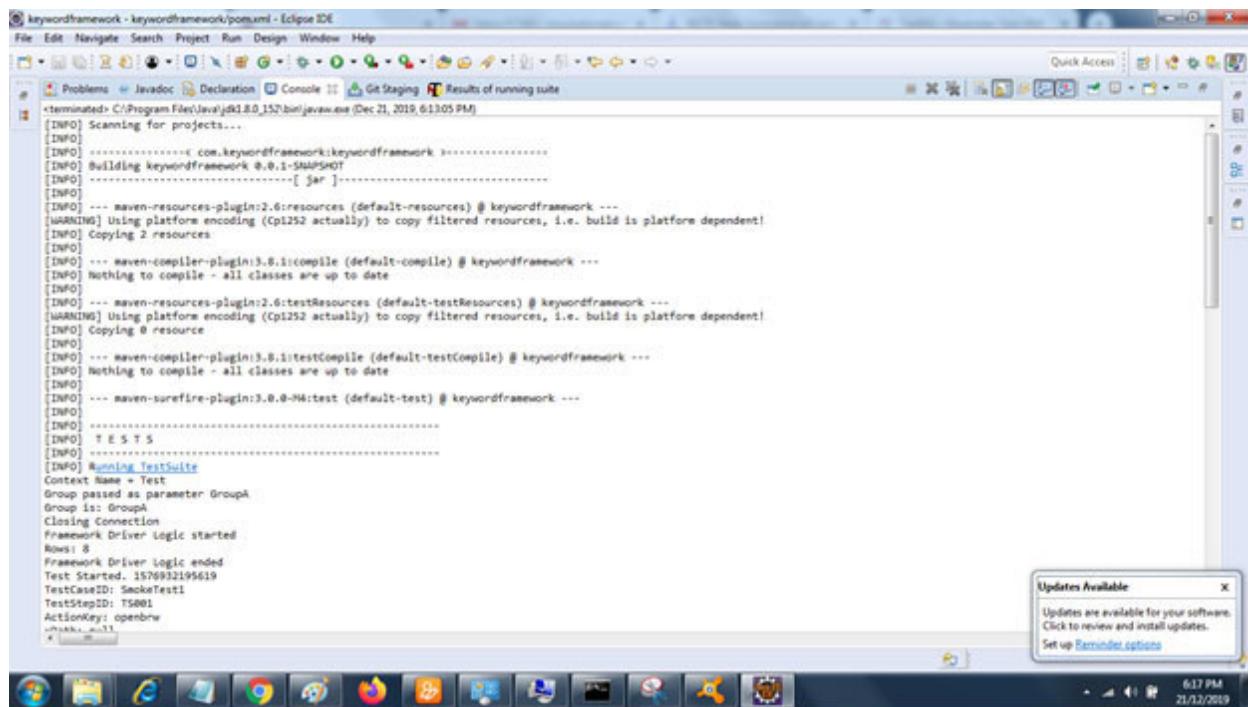
```
name="Suite">
class-name="bpb.listeners.MyListener2"/>
```

```
name="Test">
name="GroupA"/>
```

```
name="bpb.dataprovider.KeywordProvider"/>
```

Execute the POM xml by doing a right click on pom.xml and selecting **Run As | Maven**

## OUTPUT:



**Figure 10.5**

```
keywordframework - keywordframework/pom.xml - Eclipse IDE
File Edit Navigate Search Project Run Design Window Help
Problems Javadoc Declaration Console Git Staging Results of running suite
terminated: C:\Program Files\Java\jdk1.8.0_152\bin\java.exe (Dec 21, 2019, 6:13:05 PM)
TestCaseID: SmokeTest1
TestStepID: TS0001
ActionKey: openBrowser
xPath: null
dataKey: null
Test Success. 1576932195631
Test Started. 1576932195639
TestCaseID: SmokeTest1
TestStepID: TS0002
ActionKey: navigate
xPath: null
dataKey: null
Test Success. 1576932195642
Test Started. 1576932195643
TestCaseID: SmokeTest1
TestStepID: TS0003
ActionKey: click
xPath: //a[@href='https://ui.freecrm.com']
dataKey: null
Test Success. 1576932195647
Test Started. 1576932195648
TestCaseID: SmokeTest1
TestStepID: TS0004
ActionKey: enterText
xPath: //input[@name='email']
dataKey: chaudharynakin@gmail.com
Test Success. 1576932195665
Test Started. 1576932195666
TestCaseID: SmokeTest1
TestStepID: TS0005
ActionKey: enterText
xPath: //input[@name='password']
dataKey: PC9121975
Test Success. 1576932195676
Test Started. 1576932195677
TestCaseID: SmokeTest1
TestStepID: TS0006
ActionKey: click
+
=
```

Figure 10.6

```
keywordframework - keywordframework/pom.xml - Eclipse IDE
File Edit Navigate Search Project Run Design Window Help
Problems Javadoc Declaration Console Git Staging Results of running suite
terminated: C:\Program Files\Java\jdk1.8.0_152\bin\java.exe (Dec 21, 2019, 6:13:05 PM)
xPath: //input[@name='password']/following-sibling::p
dataKey: PC9121975
Test Success. 1576932195676
Test Started. 1576932195677
TestCaseID: SmokeTest1
TestStepID: TS0006
ActionKey: click
xPath: //div[text()='Login']
dataKey: null
Test Success. 1576932195682
Test Started. 1576932195683
TestCaseID: SmokeTest1
TestStepID: TS0007
ActionKey: click
xPath: //div[@class='ui buttons']/div
dataKey: null
Test Success. 1576932195685
Test Started. 1576932195686
TestCaseID: SmokeTest1
TestStepID: TS0008
ActionKey: click
xPath: //a[text()='Log Out']
dataKey: null
Test Success. 1576932195690
[ResultMap:[{TestResult{name=testKeywords status=SUCCESS method=keywordProvider.testKeywords(java.lang.String, java.lang.String, java.lang.String, java.lang.String, java.lang.String, java.lang.String)}]
[INFO] Tests run: 8, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 2.395 s - in TestSuite
[INFO]
[INFO] Results:
[INFO] Tests run: 8, Failures: 0, Errors: 0, Skipped: 0
[INFO]
[INFO] BUILD SUCCESS
[INFO]
[INFO] Total time: 7.626 s
[INFO] Finished at: 2019-12-21T18:13:16+05:30
[INFO]
```

Figure 10.7

If testng2.xml is mentioned in the SuiteXMLFiles tag and in the TestNG xml, the dataprovider is mentioned, then three browser sessions will be opened and three tests will be run in parallel. As an example, I have changed just one method to accept input from data provider, which is method test3 as shown below.

```
@Test(groups=
"GroupC",dataProvider="keywords",dataProviderClass=KeywordProvider.
class)
@Parameters({ "param3" })
public void test3(String testcaseID, String teststepID,String
actionKey
,String
xPath,String dataKey) {
System.out.println("Before starting test3");
ResultSet resultSet = null;

try {
switch(actionKey) {
case "navigate":
actKeywords.navigateURL("http://www.freecrm.com");
break;
case "click":
actKeywords.clickElement(xPath);
break;
case "enterText":
actKeywords.enterText(xPath, dataKey);
break;
}
}
```

```
} catch (Exception e) {  
e.printStackTrace();  
}  
System.out.println("After completing test3");  
  
}
```

**OUTPUT:**

Three browser sessions get opened, and test cases are executed in parallel in all three browsers.

Notice how we have reduced the duplicate code of looping through the database query results.

## Assertions in TestNG

Assertions in TestNG are used to perform any kind of validations in the tests. TestNG provides us with the Assert class that provides methods that assist in validating tests. When an assertion fails, the test script stops execution unless explicitly handled. There are two types of assertions in TestNG, as shown below:

Hard assertion

Soft assertion

Let's see each one in detail with examples.

### Hard assertion

A hard assertion is one that throws an exception immediately when an assert statement fails and continues with the next test in the test suite. In order to achieve this, we need to handle the assertion error that is thrown with a catch block similar to a Java exception. After the suite completes execution, the particular test is marked as passed instead of a FAIL.

The disadvantage of hard assertion is that the test gets passed even though there was assertion failure, which led to creating customized error handlers, which could fail the test as needed.

## Soft assertion

To deal with the disadvantage of Hard Assertions, we use something called a Soft Assertion, which is a customized error handler provided by TestNG. Soft Assertions do not throw an exception when an assertion fails and continue with the next step encountered after the assert statement. This is usually used when our test requires multiple assertions to be executed, and the user wants all of the assertions/codes to be executed before failing/skipping the tests.

The soft assertion is implemented by creating an instance of the class

Let's add a field called Expected in the testcases table. Enter the following entry in the table:

TestCaseID	TestStepID	ActionKey	ORID	Expected
SmokeTest4	TS025	navigate	NULL	NULL
SmokeTest4	TS026	click	OR01	NULL
SmokeTest4	TS027	enterText	OR02	NULL
SmokeTest4	TS028	enterText	OR03	NULL
SmokeTest4	TS029	click	OR04	NULL
SmokeTest4	TS030	verifyText	OR07	Deals Summary
SmokeTest4	TS031	click	OR05	NULL
SmokeTest4	TS032	click	OR06	NULL

**Figure 10.8**

Add the following entry in the testconfig table:

SmokeTest4	Assertions	Y	GroupA
------------	------------	---	--------

***Figure 10.9***

Add the following entries in the object\_repository table:

OR07	HPage	Deals	(//div[@class='header'])[1]
------	-------	-------	-----------------------------

***Figure 10.10***

Let's create a class for assertions.

## Class for assertions

We create a class for assertions, as shown below. We create an object for SoftAssert in the constructor. For hard assertions, we handle the assertion error by using the AssertionError class in the Java Util package. In case of soft assertions, we have added an assertAll() method, which groups the exceptions that are thrown and fail the test if required.

```
public class Assertions {  
    private SoftAssert softAssert=null;  
    public Assertions() {  
        softAssert = new SoftAssert();  
    }
```

```
    public void assertEquals(String actual,String expected) {  
        try {  
            Assert.assertEquals(actual, expected);  
        } catch (AssertionError e) {  
            System.out.println("Assertion Failed");  
        }  
        System.out.println("After Asserting");  
    }
```

```
    public void assertNotEquals(String actual,String expected) {  
        Assert.assertNotEquals(actual, expected);  
    }
```

```
public boolean checkVisible(WebElement elem) {  
    Assert.assertTrue(elem.isDisplayed());  
    return true;  
}  
  
public void assertEqualsSA(String actual, String expected) {  
    softAssert.assertEquals(actual, expected);  
  
    System.out.println("After Asserting");  
}  
  
public void assertNotEqualsSA(String actual, String expected) {  
    softAssert.assertNotEquals(actual, expected);  
  
}  
  
public void assertAll() {  
    softAssert.assertAll();  
}  
}
```

The assertAll() method will be called in the method annotated by the @AfterTest in the test class.

The test class TestClass2 is shown below with the required changes.

```
public class TestClass2 {
```

```
Action_Keywords actKeywords = new Action_Keywords();
Assertions assertions = new Assertions();
WaitClass wdWait = null;
String contactHeader = "(//div[@class='header'])[1]";
```

```
public TestClass2() {
    System.out.println("Constructor called");
}
```

```
@BeforeTest(groups= {"GroupA","GroupB","GroupC"})
public void setUp() {
    actKeywords.openBrowser("chrome");
    wdWait = new WaitClass();
    wdWait.waitImplicitly();
}
```

```
@Test(groups= "GroupA")
```

```
@Parameters({ "param1" })
```

```
public void test1(String param1) {
    System.out.println("Before starting test1");
    ResultSet resultSet = null;

    try {
        CachedRowSet crs = DBExtract3.extractRecords("GroupA");
        ResultSetMetaData rsmd = crs.getMetaData();
        crs.next();
        while (!crs.isAfterLast()) {
```

```
String testcaseid = crs.getString("TestCaseID");
String teststepid = crs.getString("TestStepID");

String actionKey = crs.getString("ActionKey");
System.out.println("ActionKey"+actionKey);
String xpath = crs.getString("XPath");
System.out.println("XPath"+xpath);
String datakey = crs.getString("DataKey");
System.out.println("DataKey"+datakey);
String verifyText = crs.getString("Expected");
System.out.println("DataKey"+verifyText);

switch(actionKey) {
case "navigate":
actKeywords.navigateURL("http://www.freecrm.com");
break;
case "click":
actKeywords.clickElement(xpath);
break;
case "enterText":
actKeywords.enterText(xpath, datakey);
break;
case "verifyText":

assertions.assertEqualsSA(SingletonDriver.getInstance().getDriver().fin
dElement(By.xpath(contactHeader)).getText(), verifyText);

break;
}

}
```

```
    crs.next();
}
} catch (SQLException e) {
e.printStackTrace();
}
System.out.println("After completing test1");
}
```

```
@Test(groups= "GroupB")
@Parameters({ "param2" })
public void test2(String param2) {
System.out.println("Before starting test2");
ResultSet resultSet = null;

try {
CachedRowSet crs = DBExtract3.extractRecords("GroupB");
ResultSetMetaData rsmd = crs.getMetaData();
crs.next();
while (!crs.isAfterLast()) {
String testcaseid = crs.getString("TestCaseID");
String teststepid = crs.getString("TestStepID");

String actionKey = crs.getString("ActionKey");
String xpath = crs.getString("XPath");
String datakey = crs.getString("DataKey");
switch(actionKey) {
case "navigate":

actKeywords.navigateURL("http://www.freecrm.com");
break;
```

```
        case "click":
            actKeywords.clickElement(xpath);
            break;
        case "enterText":
            actKeywords.enterText(xpath, datakey);
            break;

    }

    crs.next();
}

} catch (SQLException e) {
    e.printStackTrace();
}

System.out.println("After completing test2");

}

@Test(groups=
"GroupC",dataProvider="keywords",dataProviderClass=KeywordProvider
.class)
@Parameters({ "param3" })
public void test3(String testcaseID, String teststepID,String
actionKey
, String xPath, String dataKey) {
    System.out.println("Before starting test3");
    ResultSet resultSet = null;

    try {

```

```
switch(actionKey) {  
    case "navigate":  
        actKeywords.navigateURL("http://www.freecrm.com");  
        break;  
  
    case "click":  
        actKeywords.clickElement(xPath);  
        break;  
    case "enterText":  
        actKeywords.enterText(xPath, dataKey);  
        break;  
  
}  
  
}  
  
} catch (Exception e) {  
    e.printStackTrace();  
}  
System.out.println("After completing test3");  
  
}  
  
}  
  
@AfterTest(groups= {"GroupA","GroupB","GroupC"})  
public void tearDown() {  
    actKeywords.closeBrowser();  
    assertions.assertAll();  
}  
  
}
```

To check the failure message, disable all test cases other than test case 1 by using enabled="false" as shown in the TestNG XML below:

```
name="Parameter test Suite" verbose="1" parallel="false">
  name="suite-param" value="parameter at suite level" />
  name="Group Test1">

    name="GroupA"/>
    name="param1" value="GroupA" />
    name="keywordframework.TestClass2">

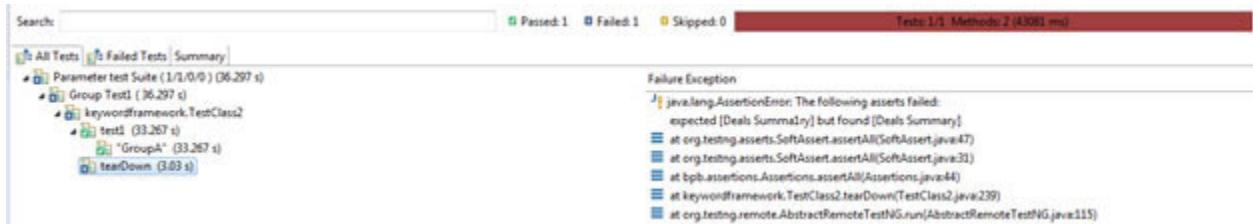
      name="Group Test2" enabled="false">
        name="GroupB"/>
        name="param2" value="GroupB" />
        name="keywordframework.TestClass2">
          name="Group Test3" enabled="false">
            name="GroupC"/>
            name="param3" value="GroupC" />

      name="keywordframework.TestClass2">
```

Execute the pom.xml and get the output as shown below.

#### **OUTPUT:**

The TearDown method fails with the error shown below:



**Figure 10.11**

Note that we did an AssertAll at the end of the test, and hence the Logout is done successfully before failing the test. Here I have just shown the The reader can practice with a hard assertion.

We now move onto extent reports.

## *Introducing extent reports*

Extent reports offer several features when compared to the built-in reports that are generated through JUnit and TestNG such as pie chart generation, test stepwise report generation, adding screenshots, etc., at every step and a beautiful user interface that can be shared with all stakeholders of the project.

### Advantages of using extent reports

There are several advantages provided by extent reports, and few of them are discussed below:

HTML report with stepwise and pie chart representation, which is customizable.

Displays the time taken for test case execution

Each test step can be provided with a screenshot.

Multiple test case runs within a single suite can be tracked easily.

It can be integrated with TestNG and JUnit frameworks quite easily.

## Components of extent reports

Extent reports comprise of two classes:

ExtentReports class

ExtentTest class

### **SYNTAX:**

```
ExtentReports reports = new ExtentReports("Path of directory where  
the resultant HTML file will be stored", true/false);
```

```
ExtentTest test = reports.startTest("TestName");
```

ExtentReports class is used to generate an HTML report on the path specified. The Boolean flag specified whether the existing report needs to be overwritten or a new report needs to be created. Value true is the default, which means all the already existing data will be overwritten.

ExtentTest class is used for logging test steps onto the generated HTML report.

### [\*\*Maven dependency for extent reports\*\*](#)

For using extent reports, the dependency shown below has to be included in

```
com.aventstack  
extreports  
4.0.9
```

Save the POM xml. Right click on project and select **Maven | Update Project.**

This download required Maven JARs.

## [Generating extent reports](#)

Let's now generate extent reports in the test class. Follow the steps below to generate the extent report:

Include the following class variables:

```
ExtentReports report = null;  
ExtentHtmlReporter reporter=null;  
ExtentTest test = null;
```

In the @BeforeTest method, set the various variables, as shown below:

```
reporter = new ExtentHtmlReporter("report.html");  
report = new ExtentReports();  
report.attachReporter(reporter);
```

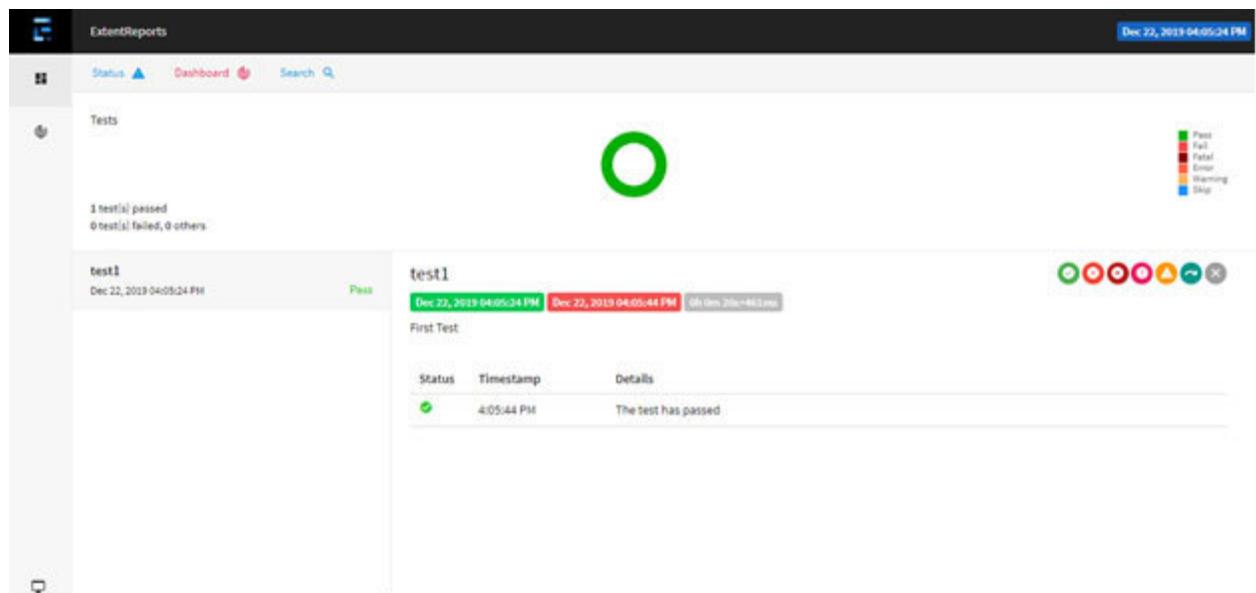
In the @Test method, write the code, as shown below:

```
test.log(Status.PASS, "The test has passed");
```

In the @AfterTest method, flush the report. Without this, the report will not get generated:

```
report.flush();
```

Run the program. Once the code completes executing, the report.html gets generated at the root. When the file is viewed in a browser, a graphical representation of the test is displayed



**Figure 10.12**

We will look at a better way of generating extent reports in the last chapter when we integrate with Jenkins.

## Integration with GitHub

Follow the steps below to integrate your project with GitHub:

Create an account on GitHub. Follow the instructions on  
[www.github.com](http://www.github.com)

Sign in using the credentials and click **Start a**

Give the repository name and give an optional description. Make the repository public

Click **Create**

Open Eclipse and click the **Open Perspective** button. Select **Git Repositories** perspective

Select **Clone a Repository** button. Fill in the repositories URL Enter authentication details and click

Click **Next** and then click

Right-click the project in Eclipse and select **Team | Share**

Select the repository and click

Right-click the project in Eclipse and click

In the **Git Staging** window, select all the unstaged changes and move to staged changes.

Enter the commit message and click **Commit and**

Click

Enter the username and password if prompted.

Click

Go to the GitHub repository and click

All the code is available on GitHub.

This completes the GitHub setup, and we come to the end of this chapter.

## Conclusion

This chapter covered the concept of Maven as a build tool. We saw the various aspects of Maven, understood the Maven lifecycle, plugin goals, and the various phases. We made use of the pom.xml file to run our project and also saw how to achieve parallel execution using We saw the command line commands to be used in running a Maven project, which we will use in the final chapter when we integrate with Jenkins. We had a look at hard and soft assertions. We saw how to generate Extent Reports. Finally, we added our project to GitHub.

The next chapter will make the reader conversant with Jenkins. We will see the command line execution of a Maven build followed by setting up a Jenkins job. We will see how to pull code from Maven for execution.

## Questions

What is the use of Maven?

Practice using the various phases of Maven

Explain the difference between hard assertions and soft assertions

Write a program to demonstrate hard assertions and soft assertions

Create an Eclipse project and add it to a GitHub repository

## *Jenkins Introduction and Scheduling*

We are going to learn Jenkins to trigger the build process. Jenkins is a build automation tool used for building projects. In this chapter, we will first see the actual batch file contents to execute the build from the command prompt. We will then create a build in Jenkins. We will see how to trigger the Jenkins job at a specific time of the day or night. We will see how to generate the extent report in Jenkins by downloading the HTML publisher plugin.

This is a very important chapter from the point of view of triggering the build process from Jenkins. This build will be automated so that it can run in the off-time when none of the testers or developers are using the AUT. This way, we will utilize the power of Maven and Jenkins to smoothen the Build and Execute process.

After reading this chapter, the reader will be conversant with the build job creation and triggering from Jenkins. The reader will become conversant with triggering the build at a specific time.

## Structure

Here's what we will learn in this chapter:

Introduction to Jenkins

Setting-up Jenkins

Executing the Maven build from the command line

create and execute a Jenkins job

Scheduling a Jenkins job

## Objectives

Learn how to setup Jenkins

Execute a Maven build from the command line

Creating and executing a Jenkins job

Scheduling a Jenkins job

We begin with an introduction to Jenkins.

## [\*Introducing Jenkins\*](#)

Jenkins is an open-source Continuous Integration server that is capable of executing a chain of actions that help to achieve Continuous Integration process through automation. Jenkins is free and written in Java. Jenkins is widely used for implementing continuous integration.

Jenkins can be run either standalone or along with a web server like Apache Tomcat. The reason Jenkins has become so popular is because of its monitoring of repeated tasks that come up during the development life cycle.

Jenkins helps to expedite the software development process, as Jenkins can automate build and test rapidly. Jenkins supports the complete software development life cycle from building, testing, documenting the software, deploying, and other stages of a software development lifecycle.

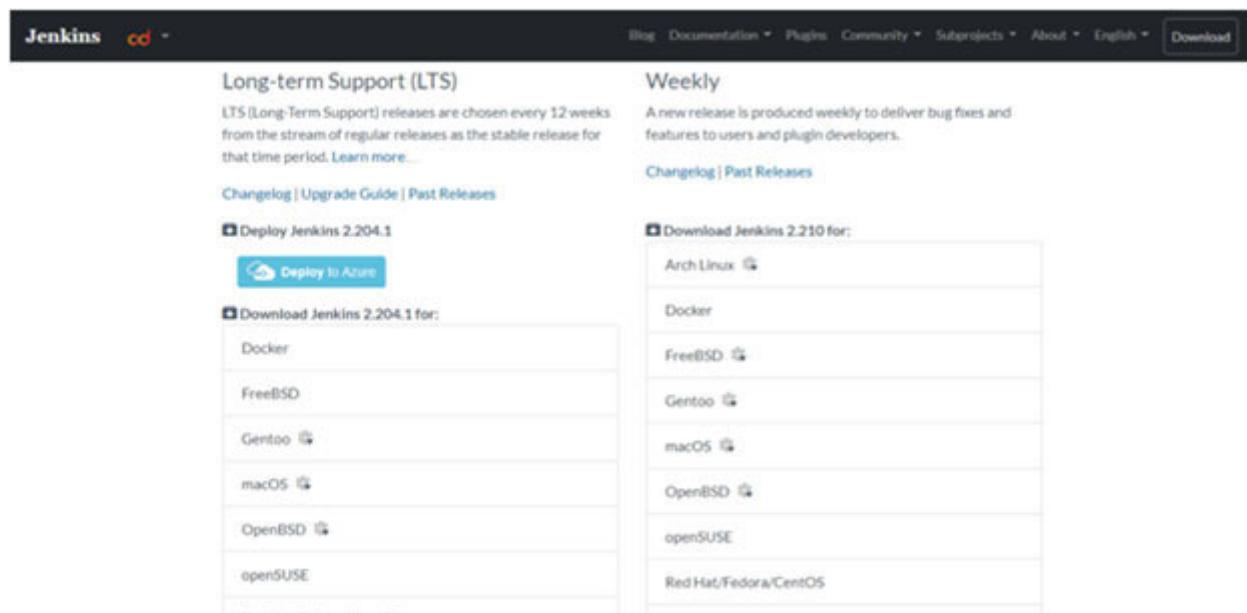
We start with the setup of Jenkins.

## Setting-up Jenkins

In this section, we will see how to download and setup Jenkins. Follow the steps below to download the Jenkins MSI file.

Navigate to Click on the **Download** button.

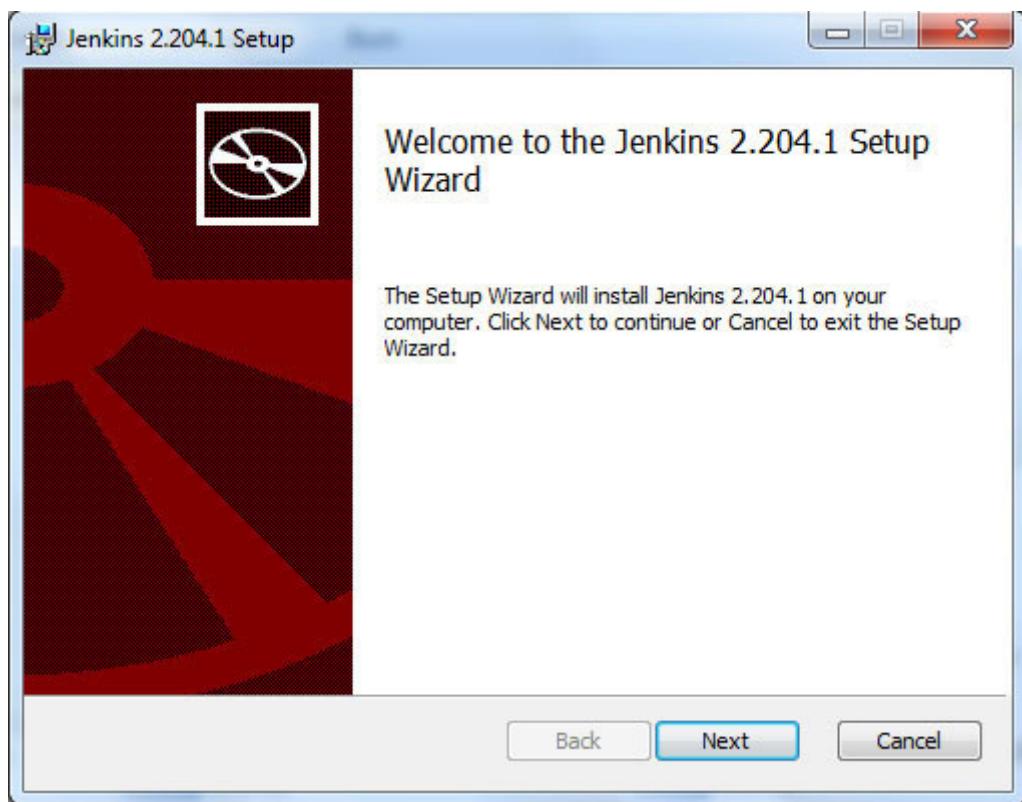
**There are two options available: Long Term Release (LTS) and Weekly.** LTS is a stable version, and Weekly release contains bug fixes and features to users. Select the **Long Term Release** and click **Windows** as the operating system.



**Figure 11.1**

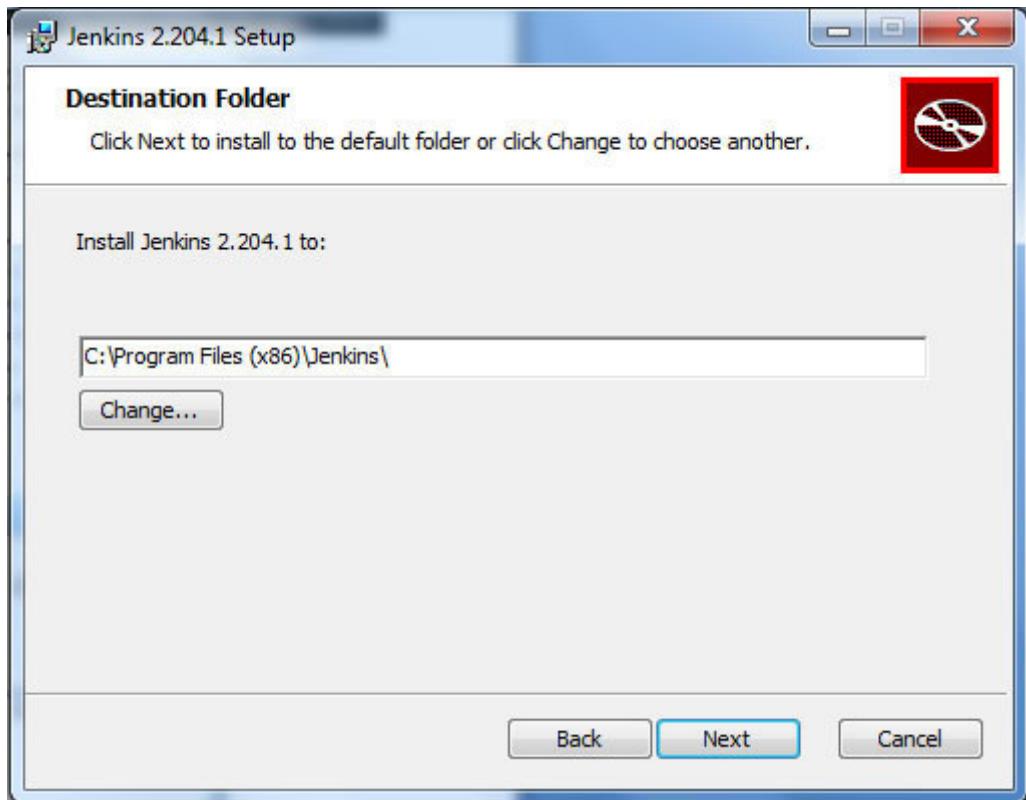
The zip file for Jenkins gets downloaded. Extract it to a suitable location

Execute the MSI file by double-clicking. Get the initial screen shown below



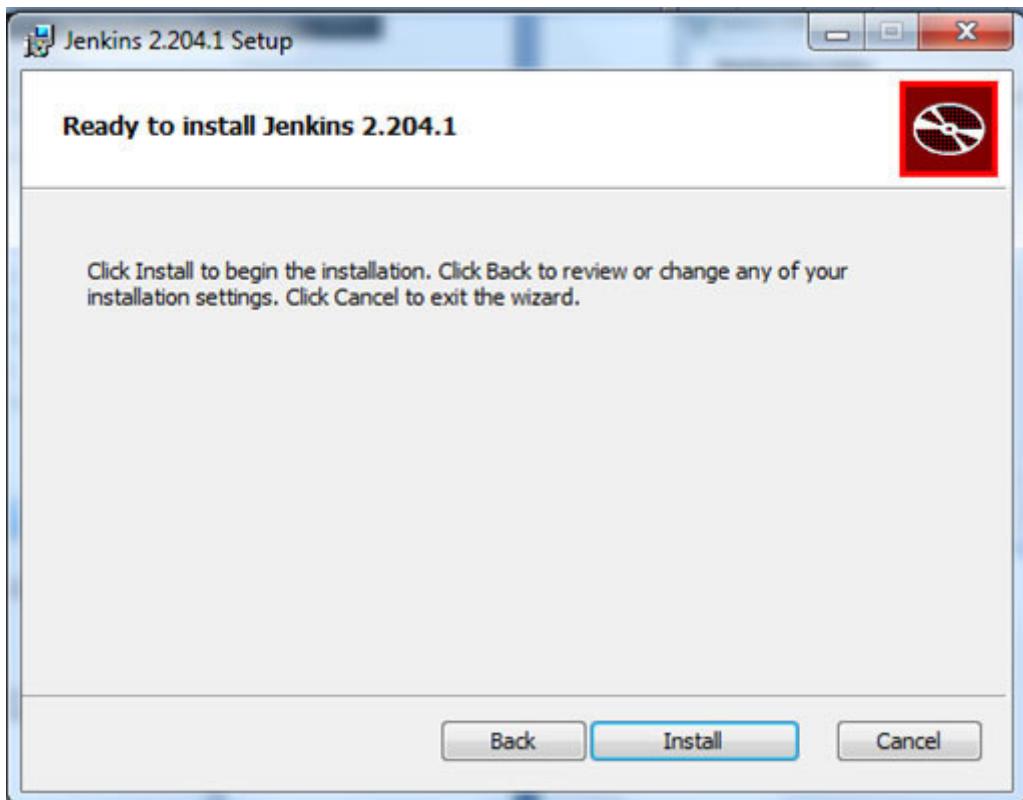
**Figure 11.2**

Click **Next** and enter the folder you want to install Jenkins. Click **Next** again.



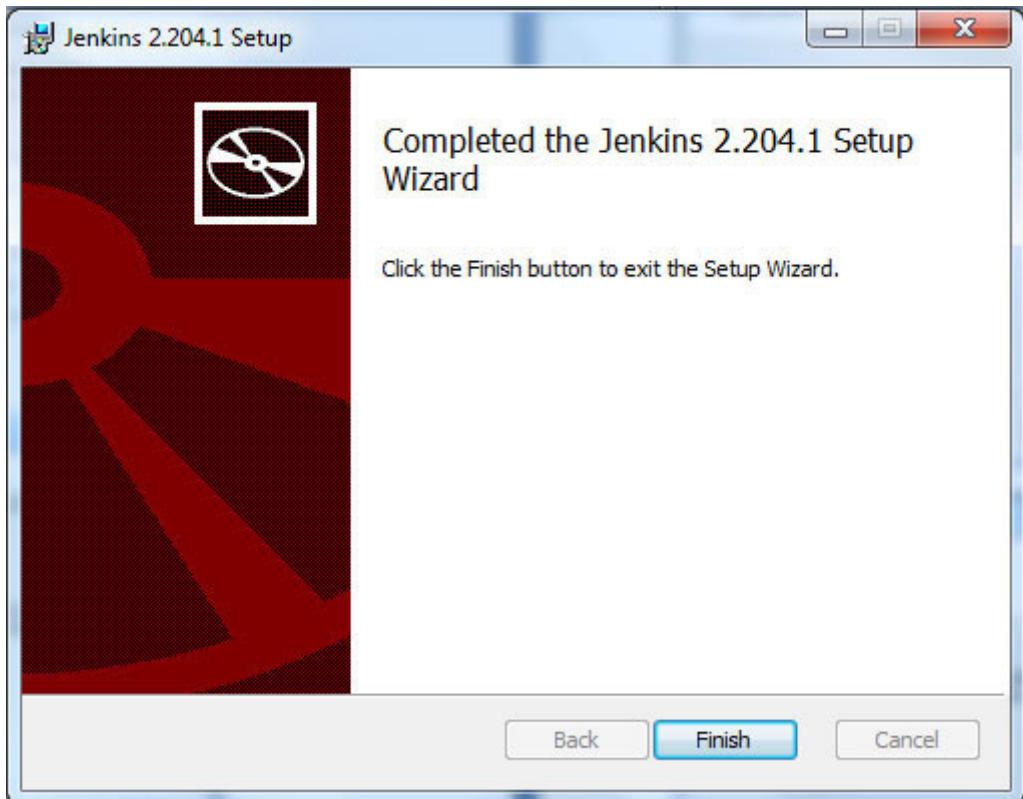
*Figure 11.3*

Click **Install** on the screen shown below. Jenkins will take around 15-20 minutes to install



*Figure 11.4*

Click **Finish** on the screen shown below



**Figure 11.5**

After clicking **Next >** You will be shown a screen to **Unlock Jenkins**. Enter the password present in the `InitialAdminPassword` file present in the `secrets` folder

Once the password is entered and the **Continue** button is clicked, you will be shown a screen with two options **Install suggested plugins** and **Select plugins to Click** **Install suggested** The installation will start.

Once the installation completes, **Create First Admin User** screen will be displayed

Enter the details. Click **Save and Finish**

Click **Start using Jenkins** on the next screen.

Click **Create New Jobs** on the next screen to create a new job

Now that Jenkins is up and running, let's see how to execute a Maven build from the command line.

## [\*Execute a Maven Build from the command line\*](#)

In the previous chapter, we saw how to execute a Maven build from Eclipse by doing a right-click on POM.xml and clicking **Run As | Maven**. Now we will see how to execute a Maven build from the command line.

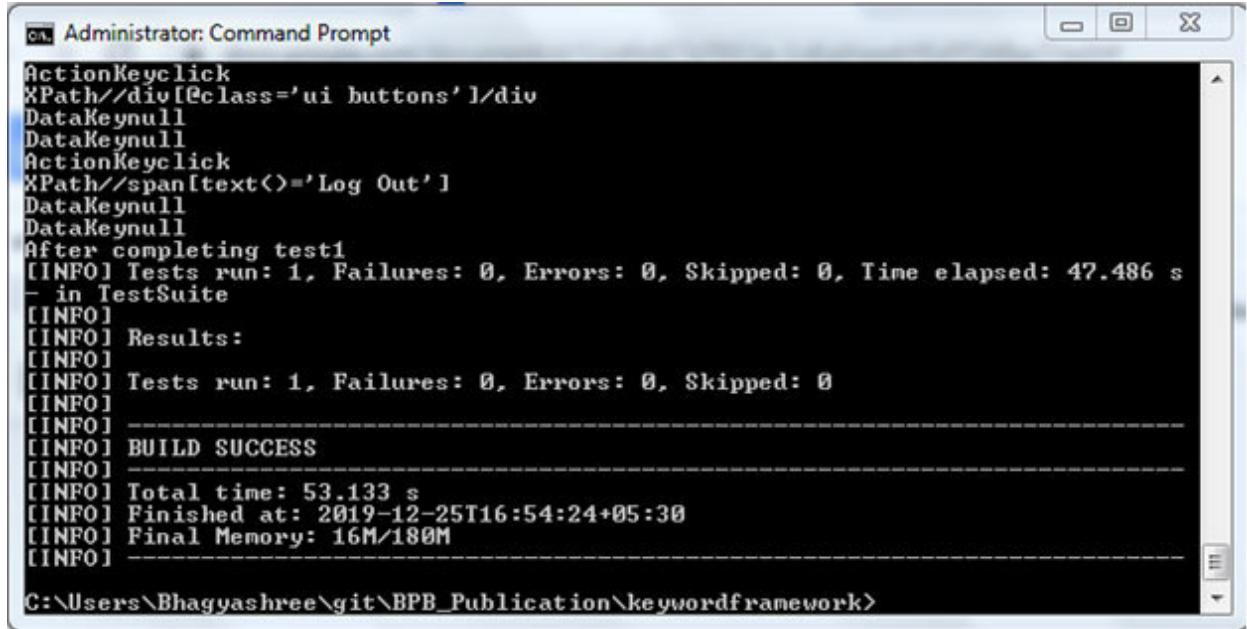
Follow the steps below to execute a Maven build from Command-Line.

Open the command prompt

Navigate to the folder of your Maven project

Type mvn Press **Enter** and the project starts building and executing

Eventually, get the **Build Success** message as shown below



The screenshot shows an 'Administrator: Command Prompt' window with the following text output:

```
ActionKeyclick
XPath//div[@class='ui buttons']/div
DataKeynull
DataKeynull
ActionKeyclick
XPath//span[text()='Log Out']
DataKeynull
DataKeynull
After completing test1
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 47.486 s
in TestSuite
[INFO]
[INFO] Results:
[INFO]
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0
[INFO]
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 53.133 s
[INFO] Finished at: 2019-12-25T16:54:24+05:30
[INFO] Final Memory: 16M/180M
[INFO] -----
```

C:\Users\Bhagyashree\git\BPB\_Publication\keywordframework>

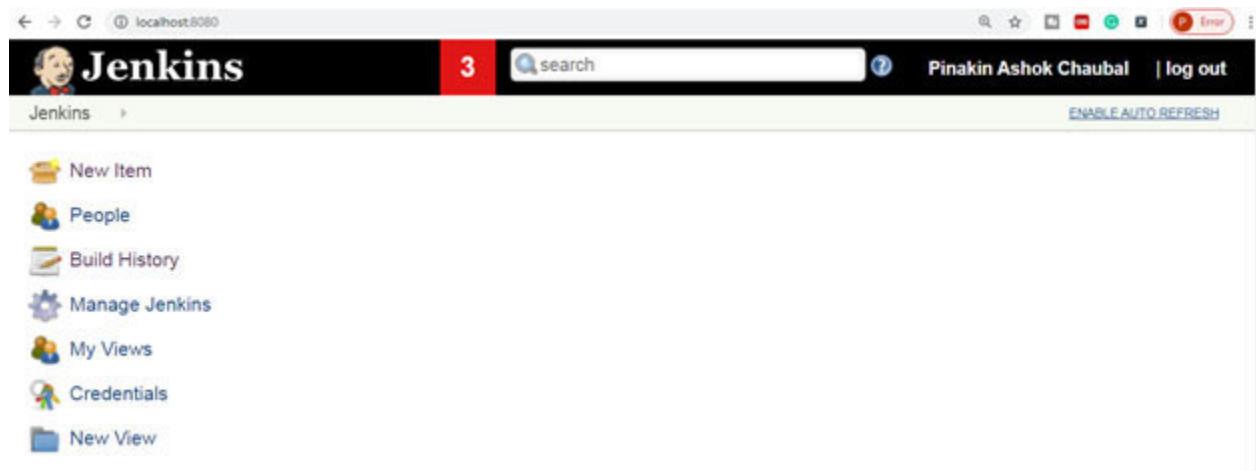
*Figure 11.6*

**Please Note:** If you see compile errors, check and see the value in the scope tag of the TestNG dependency. If it is “test” then change it to “compile” and it should work

Now we are ready to create our first Jenkins job. So let's do it!

## Creating a Jenkins job

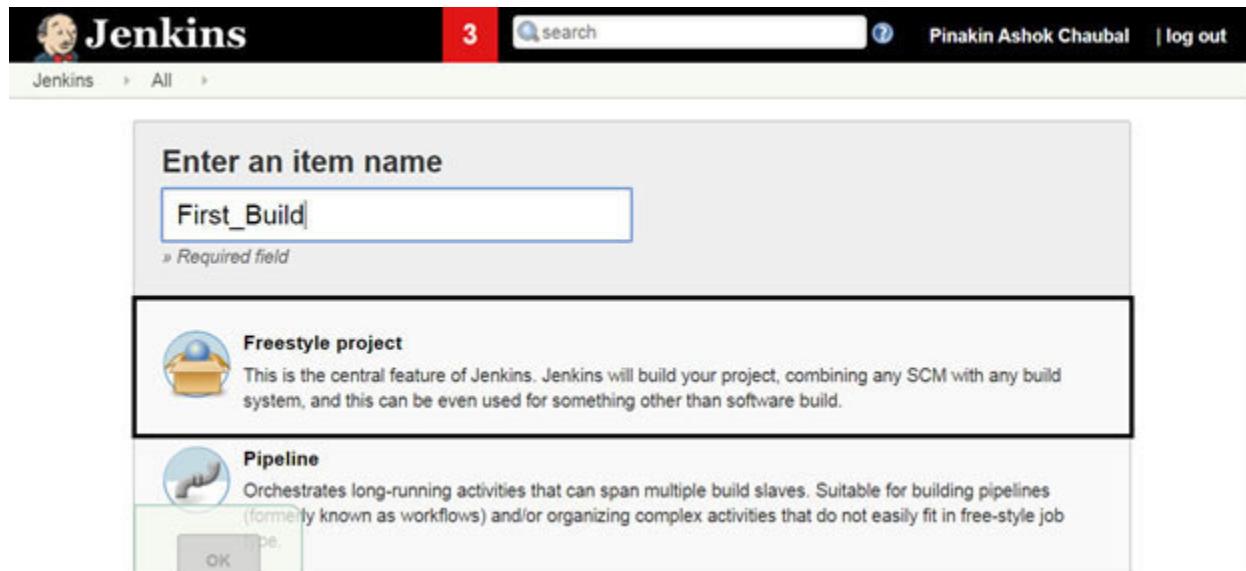
Since we have executed the MSI files for Jenkins and setup Jenkins, simply navigate to **http://localhost:8080** and sign in with the user that was created earlier. Get the screen similar to the one shown below.



**Figure 11.7**

Follow the steps below to create your first Jenkins job

Click on **New** Get the screen shown below. Enter the item name as **First\_Build** and click on the **Freestyle** Click



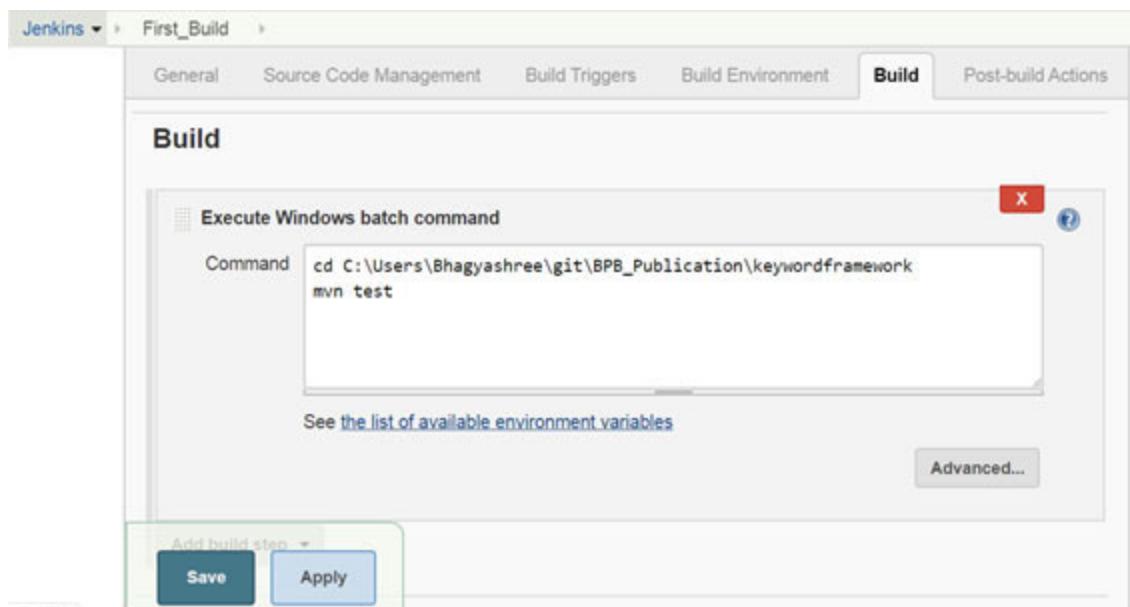
**Figure 11.8**

Get the screen shown below, enter a description as shown below

The screenshot shows the Jenkins configuration page for the "First\_Build" job. The top navigation bar shows the path "Jenkins > First\_Build >". Below the navigation bar, there are tabs for "General", "Source Code Management", "Build Triggers", "Build Environment", "Build", and "Post-build Actions". The "General" tab is selected. In the "General" tab, there is a "Description" field containing the text "This is our first build". Below the description field are several checkboxes: "Discard old builds", "GitHub project", "This project is parameterized", and "Throttle builds". At the bottom of the "General" tab, there are "Save" and "Apply" buttons. The "Apply" button is highlighted with a green box. A note below the "Apply" button says "This will apply changes to the job without restarting it if necessary".

**Figure 11.9**

Select the **Build** tab and type the contents, as shown below. Click



*Figure 11.10*

Now go to the home page and click **Build Now** after hovering on the job

Jenkins will start executing the job. The console output is shown below

```
Jenkins > First_Build > #1
[INFO]
[INFO] -----
[INFO] T E S T S
[INFO] -----
[INFO] Running TestSuite
[TestNGContentHandler] [WARN] It is strongly recommended to add "<!DOCTYPE suite SYSTEM "https://testng.org/testng-1.0.dtd">" at the top of your file, otherwise TestNG may fail or not work as expected.
log4j:ERROR Could not find value for key log4j.appenders.LOGFILE.layout
Constructor called
Starting ChromeDriver 2.41.578737 (49da6702b16031c40d63e5618de03a32ff6c197e) on port 1570
Only local connections are allowed.
Dec 25, 2019 5:50:54 PM org.openqa.selenium.remote.ProtocolHandshake createSession
INFO: Detected dialect: OSS
Session is bd9d3683acba5fe97709e9801a165b55
Before starting test1
Group is: GroupA
Closing Connection
ActionKeynavigate
XPathnull
DataKeynull
DataKeynull
ActionKeyclick
XPath//a[@href='https://ui.freecrm.com']
```

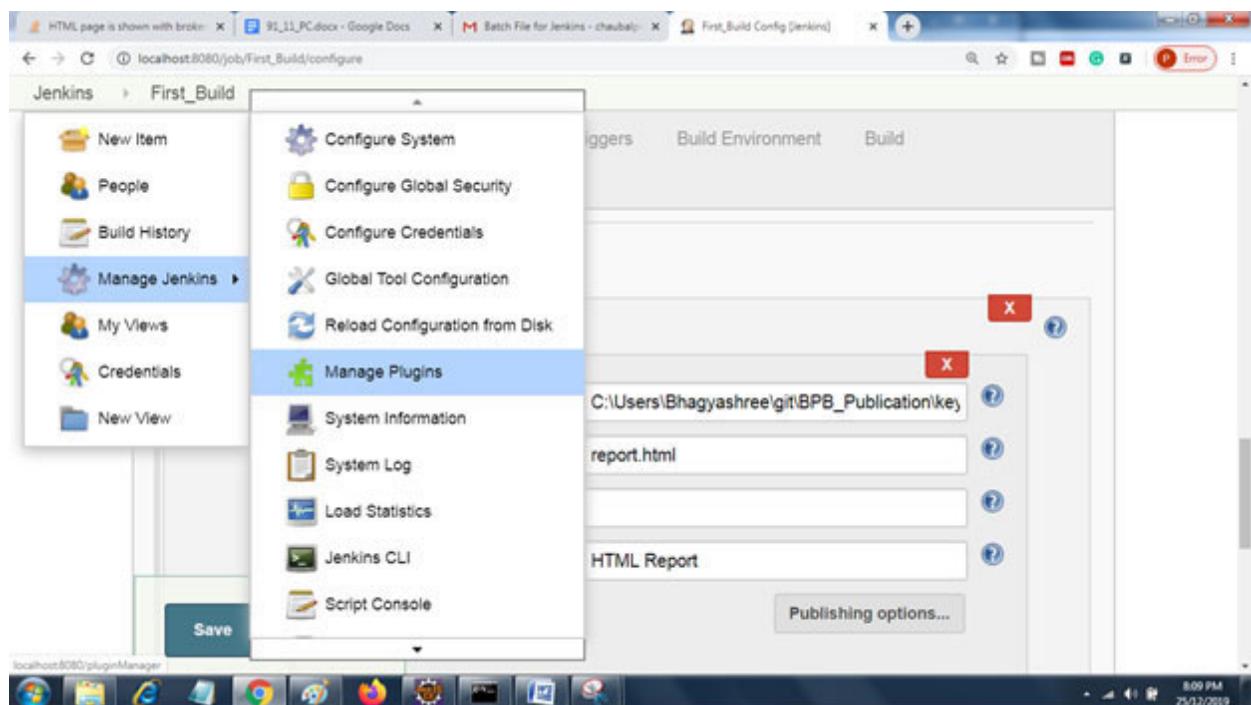
**Figure 11.11**

Jenkins executes in headless mode.

## Configuring extent report in Jenkins

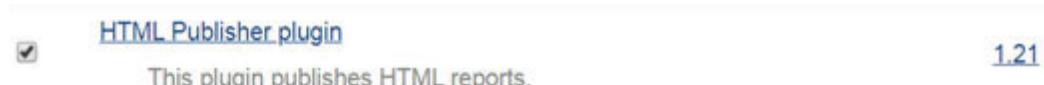
In order to configure extent report in Jenkins, follow the steps shown below

Install the HTML Publisher plugin by going to **Manage Plugins** as shown below



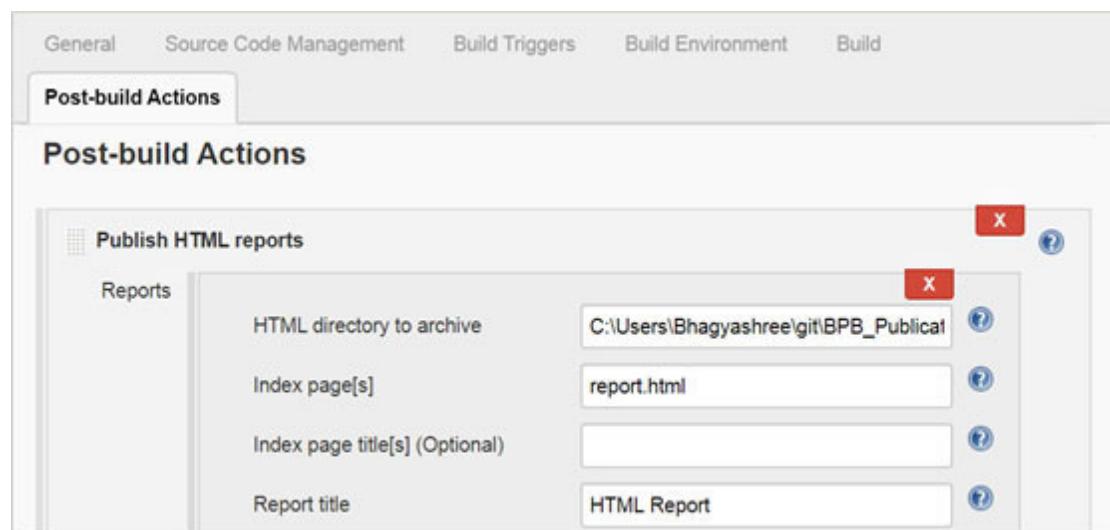
**Figure 11.12**

The HTML Publisher plugin will be in the available plugins and looks like the one shown below:



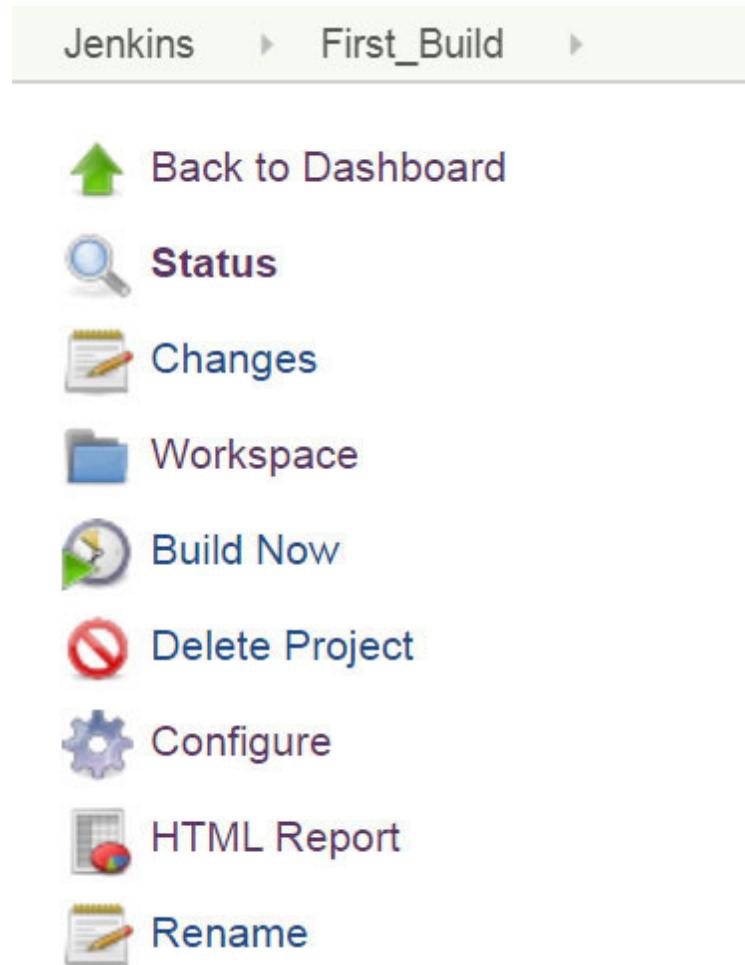
**Figure 11.13**

Go to **Post Build Actions** and configure the extent report, as shown below. Give the directory in which the report gets created as shown below



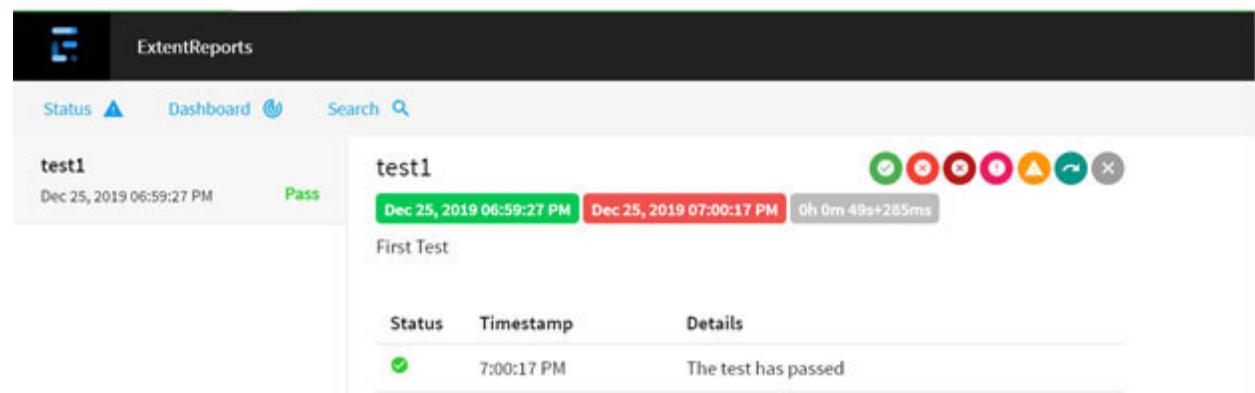
**Figure 11.14**

Build the job again and see that there is a link for HTML report as shown below



**Figure 11.15**

Click the HTML report link and get the extent report as shown below



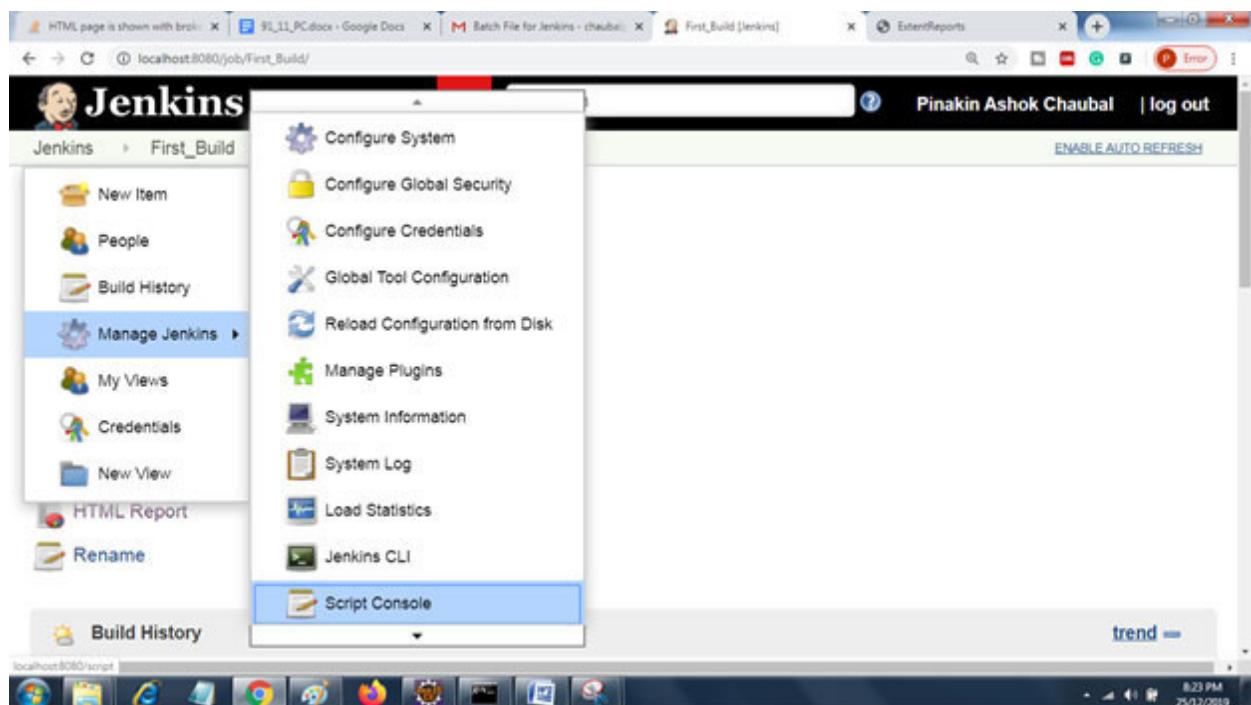
The screenshot shows the ExtentReports interface. At the top, there's a navigation bar with "Status" (blue triangle), "Dashboard" (blue circle), and "Search" (magnifying glass). The main area has a dark header with the "ExtentReports" logo. Below the header, there's a table-like structure:

test1	Dec 25, 2019 06:59:27 PM	Pass
		test1
	Dec 25, 2019 06:59:27 PM	Dec 25, 2019 07:00:17 PM
		0h 0m 49s+285ms
		First Test
Status	Timestamp	Details
✓	7:00:17 PM	The test has passed

To the right of the table, there's a row of colored circular icons: green, red, yellow, blue, orange, and grey.

**Figure 11.16**

In case the extent report appears broken, include these two lines in the script console and click **Run** as shown below



**Figure 11.17**

The script console appears, as shown below.



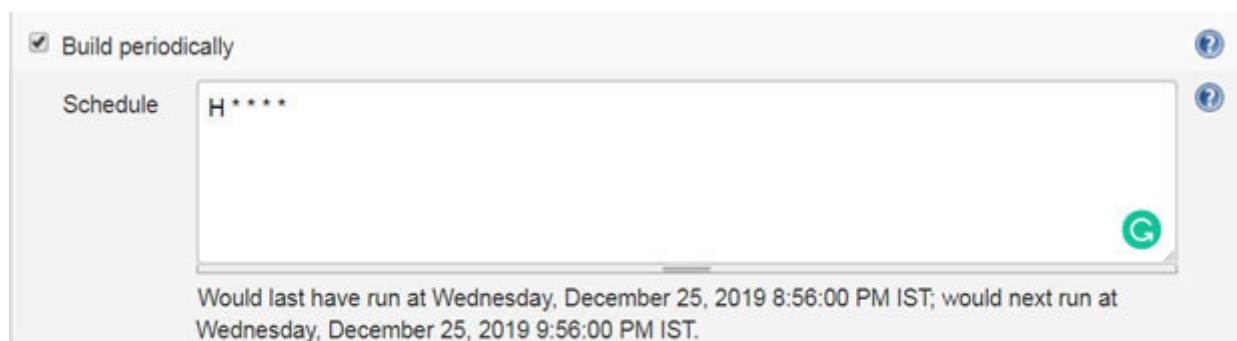
**Figure 11.18**

Go back to **HTML** and now the report appears proper.

Let's see how to schedule a Jenkins job to build periodically.

## Scheduling a Jenkins job to build periodically

In order to configure a Jenkins job to build periodically, change the Build Triggers as shown below



**Figure 11.19**

Here I have scheduled the job to run every hour. This is just an example. You can schedule it at your convenience.

With this, we come to the end of this chapter. This was a short yet important chapter on Jenkins.

## Conclusion

This chapter covered the concept of Jenkins as a build automation tool. We saw the various facets of Jenkins, understood how a build is executed from Command-Line, and later, we executed the same build from Jenkins. We installed the HTML publisher plugin and viewed the generated extent report in Jenkins. Finally, we saw how to build periodically.

The next chapter will be about executing in the cloud. We will use browser stack to execute in the cloud. This will be another interesting chapter to learn.

## Questions

What is the use of Jenkins?

Write a simple Java Program and schedule the Selenium build dependent on this program

Explain the Configure Jenkins part of Jenkins

Generate an Extent Report using HTML publisher plugin for a sample program

How to build every 1/2 hour in Jenkins

**Selenium Grid and Executing in the Cloud**

We are going to see how to build a Selenium grid in this chapter, which we will later execute in the cloud. We will see what a RemoteWebDriver is. We will change the default set of browsers in Selenium Grid. We will also add a parameter in TestNG XML to specify whether to execute from the cloud. In order to execute in the cloud, we will be using BrowserStack.

This is a very important chapter from the point of view of learning how to execute from the cloud. This chapter will highlight the changes required for running your Selenium tests from a cloud environment.

After reading this chapter, the reader will be conversant with the RemoteWebDriver, working with Selenium Grid, and executing a framework from a cloud solution.

## Structure

Here's what we will learn in this chapter:

Introducing RemoteWebDriver

Learning about Selenium StandAlone server

Learning about the RemoteWebDriver client

Steps to convert a regular script to use RemoteWebDriver Server

Looking at the hub

Knowing the node

Hub configuration parameters

Node configuration parameters

Specifying configuration using JSON files

Changes to the SingletonDriver class

Introducing BrowserStack

## Setting-up BrowserStack

## Objectives

Learn what a RemoteWebDriver is

Learn the concept of a Selenium Grid

Learn about the Hub and Node

Learn the different configuration parameters for the hub and node

Understand how to execute your test cases in the cloud

We begin with an introduction to the RemoteWebDriver.

## [Introducing RemoteWebDriver](#)

Up until now, we have created our test scripts and executed them on browsers. All of these tests were executed against the browsers that were installed on a local desktop or laptop, where the test scripts reside. This may not be feasible at all times. There are chances that you may be working on Mac or Linux but want to execute your tests on Chrome on a Windows machine. This is where RemoteWebDriver comes into the picture.

RemoteWebDriver is a class implementing the WebDriver interface that a test-script creator can use to execute test scripts via the Selenium Standalone server on a remote machine.

RemoteWebDriver consists of a server and a client. The test script uses the WebDriver client libraries, appropriate driver executable, and the actual browser sitting on the same machine.

The test script is located on a local machine, while the browsers are installed on a remote machine. This is where RemoteWebDriver comes into the play. As mentioned earlier, there are two components linked with the server and the client. Let's start with the Selenium standalone server.

## [Learning about Selenium standalone server](#)

Selenium standalone server listens on a port for various requests from a RemoteWebDriver client. Once it receives the requests, it sends them to any of the appropriate drivers, as requested by the RemoteWebDriver client.

The Selenium standalone server can be downloaded from Let's download version 3.14.0 of it, as we are using WebDriver Version 3.14.59. This server JAR should be downloaded and placed at an appropriate location on the remote machine on which the browsers are located. Also, make sure the remote machine has Java runtime installed on it. For this book, we will use the same machine as the server and client.

## Starting the server

Open your command prompt on the remote machine and navigate to the location where you have downloaded the JAR file. To start Selenium standalone server, execute the following command:

```
java -jar selenium-server-standalone-3.14.0.jar
```

Once this command is executed, the output shown below is displayed in the console:

```
C:\Users\Bhagyashree\Documents>java -jar selenium-server-standalone-3.14.0.jar
08:14:59.083 INFO [GridLauncherV3.launch] - Selenium build info: version: '3.14.0', revision: 'aaccce0'
08:14:59.085 INFO [GridLauncherV3$1.launch] - Launching a standalone Selenium Server on port 4444
2020-01-04 08:15:00.101:INFO::main: Logging initialized @1935ms to org.seleniumhq.jetty9.util.log.StdErrLog
08:15:02.153 INFO [SeleniumServer.boot] - Selenium Server is up and running on port 4444
```

**Figure 12.1**

The server has started and is listening on the of remote machine:>:4444 address for remote connections from the RemoteWebDriver client.

On the remote machine (note that here for this book, it is the same machine), the Selenium standalone server will form a bridge between the test script and the browsers. The test script

establishes a connection with the Selenium standalone server, which in turn forwards the commands to the browser installed on the remote machine.

## [Learning about the RemoteWebDriver client](#)

Now we have our Selenium Standalone server up and running. Let's create a RemoteWebDriver client. This client is nothing but the language-binding libraries that serve as a RemoteWebDriver client. RemoteWebDriver converts the test-script requests or commands to JSON payload and sends them to the RemoteWebDriver server using the JSON wire protocol.

When your tests are executed locally, the WebDriver client libraries interact with the appropriate driver directly. Now when you try to execute your tests remotely, the WebDriver client libraries communicate to Selenium Standalone Server, and the server in turn talks to the appropriate driver as requested by the test script, making use of the DesiredCapabilities class. We will explore the DesiredCapabilities class in the next section.

### Steps to convert a regular script to use RemoteWebDriver Server

For converting scripts to use the we will first see the constructor of RemoteWebDriver as shown below

```
RemoteWebDriver(java.net.URL remoteAddress, Capabilities  
desiredCapabilities)
```

The input parameters for the constructor include the hostname or IP of Selenium Standalone Server running on the remote machine and the desired capabilities required for running the test, which can be the name of the browser and/or operating system. These two are just examples. There can be more.

Follow the steps below to convert your scripts to use the

Create a DesiredCapabilities object and set the browser name on that object

```
DesiredCapabilities cap = new DesiredCapabilities();  
cap.setBrowserName("chrome");
```

Create an instance of RemoteWebDriver as shown below

```
driver = new RemoteWebDriver(new  
URL("http://localhost:4444/wd/hub"), caps);
```

We have created a RemoteWebDriver instance that tries to connect to where Selenium Standalone Server is running and listening for requests. Having done that, we also need to specify which browser your test case should execute on. This was done using the DesiredCapabilities instance.

Before running tests, Selenium Standalone Server has to be restarted by specifying the path of ChromeDriver as shown below

```
java -jar -
Dwebdriver.chrome.driver=C:\Users\Bhagyashree\workspace\Selenium
Framework\src\main\resources\chromedriver.exe selenium-server-
standalone-3.14.0.jar
```

Create a test class, as shown below:

```
public class TestClass3 {
    WebDriver driver=null;
    @Test
    public void test1() {
        DesiredCapabilities caps = new DesiredCapabilities();
        caps.setBrowserName("chrome");
        try {
            driver = new RemoteWebDriver(new
```

```

URL("http://localhost:4444/wd/hub"), caps);
} catch (MalformedURLException e) {
e.printStackTrace();
}
driver.get("http://www.freecrm.com");
}
}

```

Running the following test with RemoteWebDriver will launch the Chrome browser session and execute the test case on it. The console output is shown below. It is visible in the console where the server is running.

```

10:02:21.342 INFO [ActiveSessionFactory.apply] - Capabilities are: {
  "browserName": "chrome"
}
10:02:21.342 INFO [ActiveSessionFactory.lambda$apply$11] - Matched factory org.openqa.selenium.remote.server.ServicedSession$Factory <provider: org.openqa.selenium.chrome.ChromeDriverService>
Starting ChromeDriver 2.41.578737 <49da6702b16031c40d63e5618de03a32ff6c197e> on port 26746
Only local connections are allowed.
10:02:32.121 INFO [ProtocolHandshake.createSession] - Detected dialect: OSS
10:02:32.267 INFO [RemoteSession$Factory.lambda$performHandshake$0] - Started new session 226f01f24796e5e408387b3f1954ff5a <org.openqa.selenium.chrome.ChromeDriverService>

```

**Figure 12.2**

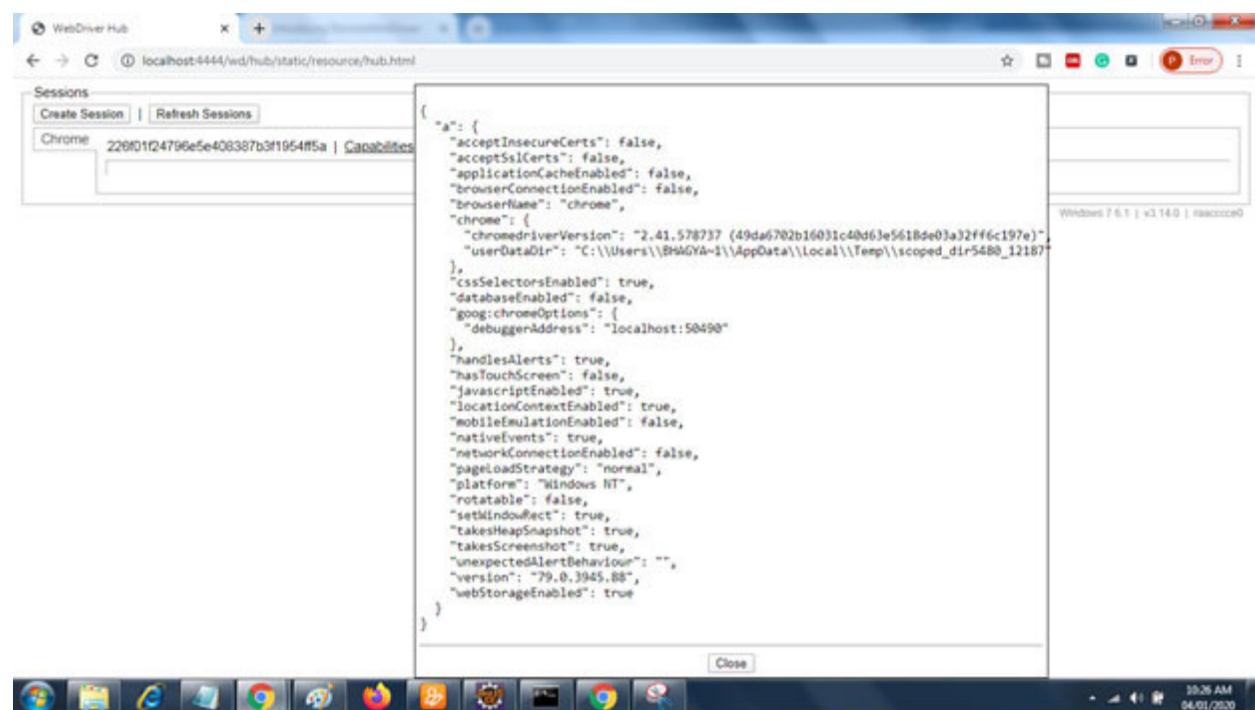
The console states that a new session with the desired capabilities is being created. Once the session is established, a session ID gets printed to the console. At any point in time, all of the sessions that are established with Selenium Standalone Server can be viewed by navigating to the host or IP of the machine where the Selenium server is running

This URL will give the entire list of sessions that the server is currently handling. The screenshot of this is shown below:



**Figure 12.3**

This is the portal that lets the test-script developer see all of the sessions created by the server and perform some basic actions on it, such as taking a screenshot of a session, loading a script to a session, terminating a session, and viewing all the desired capabilities of a session. The following screenshot shows all the default desired capabilities of our current session.



#### ***Figure 12.4***

These are the default desired capabilities that are set by the server for this session. Now we have successfully established a connection between our test script, using a RemoteWebDriver client, and the Selenium Standalone Server.

Now that we know what a RemoteWebDriver is let's understand the concepts behind Selenium Grid

## **Understanding the Selenium Grid**

Let's try to understand why we need Selenium Grid by analyzing a scenario. Suppose you have a web application that needs to be tested on the following browser-machine combinations:

Google Chrome on Windows 10

Internet Explorer on Windows 10

Firefox on Windows 10

We can simply alter the test script we created in the previous chapter and point to the Selenium Standalone Server running on each of these combinations (that is, Windows 10 with Google Chrome, Internet Explorer and Firefox), as shown in the following code.

For Windows 10 and Chrome:

```
DesiredCapabilities caps = new DesiredCapabilities();
caps.setBrowserName("chrome");
caps.setPlatform(Platform.WIN10);
try {
driver = new RemoteWebDriver(new
URL("http://localhost:4444/wd/hub"), caps);
```

```
} catch (MalformedURLException e) {
e.printStackTrace();
}
driver.get("http://www.freecrm.com");
}

For Windows 10 and Internet Explorer

DesiredCapabilities caps = new DesiredCapabilities();
caps.setBrowserName("internet explorer");
caps.setPlatform(Platform.WIN10);
try {

driver = new RemoteWebDriver(new
URL("http://localhost:4444/wd/hub"), caps);
} catch (MalformedURLException e) {
e.printStackTrace();
}
driver.get("http://www.freecrm.com");
}
```

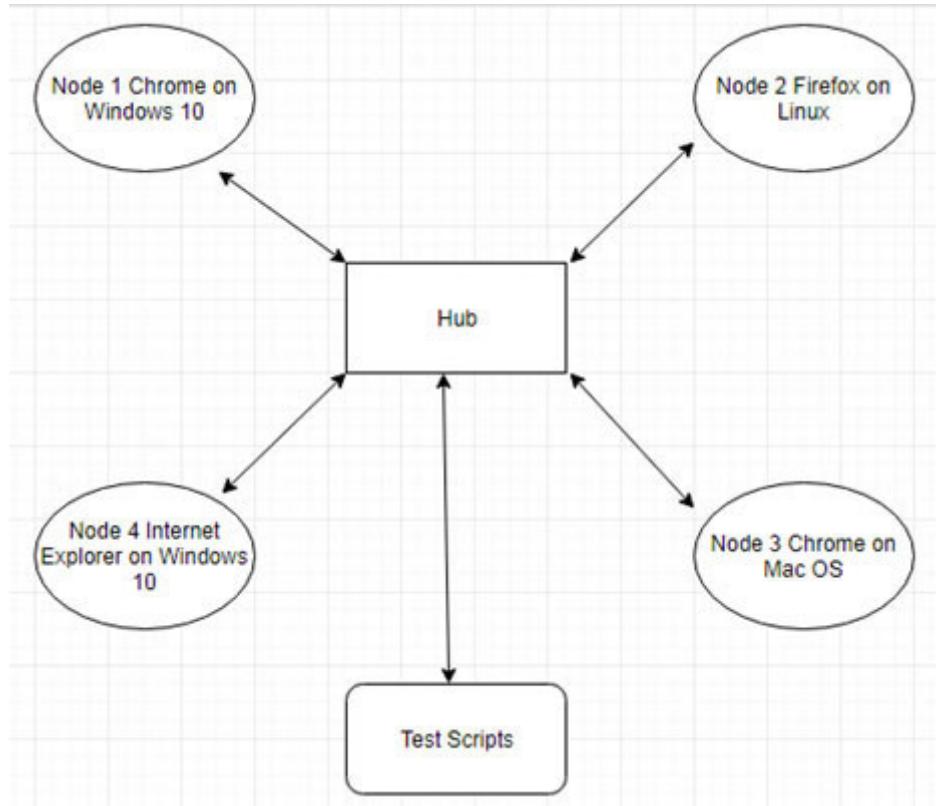
For Windows 10 and Firefox

```
DesiredCapabilities caps = new DesiredCapabilities();
caps.setBrowserName("firefox");
caps.setPlatform(Platform.WIN10);
try {
driver = new RemoteWebDriver(new
URL("http://localhost:4444/wd/hub"), caps);
} catch (MalformedURLException e) {
e.printStackTrace();
}
driver.get("http://www.freecrm.com");
```

}

In the preceding code, test scripts are tightly coupled to the machines that host the target platform and the target browsers. If the Windows 10 host changes, you need to alter your test script to handle that. This is not an optimized way to design your tests. The test scripts should focus on the functionality of the web application and not on the infrastructure that is used to execute these tests. There should be a central touchpoint to manage all the different environments. To solve this, Selenium Grid is used.

The Selenium Grid provides a cross-browser testing environment with several different platforms (such as Windows, Mac, and Linux) to execute our tests. The Selenium Grid is managed from a central point, known as the hub. The hub has the information of all the different testing platforms, also known as nodes. Nodes are the machines that have the desired operating systems and browser versions and are connected to the hub. The hub identifies and assigns these nodes to execute tests whenever the test scripts request them, based on the capabilities requested by the test script. The picture shown below gives a high-level view of the Selenium Grid:



**Figure 12.5**

In the diagram shown above, there is one hub, four nodes of different platforms, and the machine where the test scripts are located. The test script will communicate with the hub and, in turn, request a target platform to be executed. The hub allocates a node with the desired target platform to the test script. The node executes the test script and sends the result back to the hub, which forwards the results to the test script.

Now that we have seen how Selenium Grid works theoretically, let's learn the actual functioning of the Grid. We can use the same Selenium Server Standalone JAR file to be started in the hub mode on the hub machine, and a copy of the JAR file can

be started in the node mode on the node machine. Let's start by understanding what the hub is.

## [Looking at the hub](#)

The hub is the center of a Selenium Grid. It maintains a registry of all the available nodes that are connected and part of a particular grid. The hub is nothing but a Selenium Standalone server running in the hub mode, listening on port 4444 of a machine by default. The test scripts will try to connect to the hub on this port. The hub re-routes the test-script traffic to the appropriate test-platform node. Let's see how to start a hub node. Navigate to the location of your Selenium server JAR file and execute the following command:

```
java -jar selenium-server-standalone-3.14.0.jar -role hub
```

Running this command will start your server in the hub mode. By default, the server starts listening on port You can start the server on the port of your choice. Suppose you want to start the server on port it can be done as follows:

```
java -jar selenium-server-standalone-3.14.0.jar -role hub -port 2222
```

The console output is shown below where the Grid hub has started on port

```
12:25:24.906 INFO [GridLauncherV3.launch] - Selenium build info: version: '3.14.0', revision: 'aacccce0'
12:25:24.916 INFO [GridLauncherV3$2.launch] - Launching Selenium Grid hub on port 2222
2020-01-04 12:25:25.460:INFO::main: Logging initialized @1150ms to org.seleniumhq.jetty9.util.log.StdErrLog
12:25:27.080 INFO [Hub.start] - Selenium Grid hub is up and running
12:25:27.082 INFO [Hub.start] - Nodes should register to http://192.168.0.73:2222/grid/register/
12:25:27.082 INFO [Hub.start] - Clients should connect to http://192.168.0.73:2222/wd/hub
```

**Figure 12.6**

All the test scripts should connect to the hub on this port. Now launch your browser and connect to the machine that is hosting your hub on port Here, the machine that is hosting my hub has the IP address



**Figure 12.7**

The snapshot displayed above shows the version of the server that is being used as the Grid Hub. Click the Console link to navigate to the Grid Console and then click View



The screenshot shows the Selenium Grid Console interface. At the top, it says "Grid Console v.3.14.0". Below that, there's a "Help" link. The main content area displays the configuration for a hub:

```
Config for the hub:
browserTimeout:0
debug:false
host:192.168.0.73
port:2222
role:hub
timeout:1800
cleanUpCycle:5000
capabilityMatcher:org.openqa.grid.internal.utils.DefaultCapabilityMatcher
newSessionWaitTimeout:-1
throwOnCapabilityNotPresent:true
registry:org.openqa.grid.internal.DefaultGridRegistry
Config details:
hub launched with : -browserTimeout 0 -debug false -host 192.168.0.73 -port 2222 -role hub -timeout 1800 -cleanUpCycle 5000 -capabilityMatcher org.openqa.grid.internal.utils.DefaultCapabilityMatcher -newSessionWaitTimeout -1 -throwOnCapabilityNotPresent true -registry org.openqa.grid.internal.DefaultGridRegistry
the final configuration comes from :
the default :
browserTimeout:0
debug:false
port:4444
role:hub
timeout:1800
cleanUpCycle:5000
capabilityMatcher:org.openqa.grid.internal.utils.DefaultCapabilityMatcher
newSessionWaitTimeout:-1
throwOnCapabilityNotPresent:true
registry:org.openqa.grid.internal.DefaultGridRegistry
updated with params:
browserTimeout:0
debug:false
port:4444
role:hub
timeout:1800
cleanUpCycle:5000
capabilityMatcher:org.openqa.grid.internal.utils.DefaultCapabilityMatcher
newSessionWaitTimeout:-1
throwOnCapabilityNotPresent:true
registry:org.openqa.grid.internal.DefaultGridRegistry
```

**Figure 12.8**

As you can see, the web page talks about configuration parameters, which we will discuss when we configure the Selenium Grid.

## Knowing the node

We have our hub is up and running; let's start a node and connect it to our hub. Let's see how to start the node. The command to start the node and register with our hub is as follows:

```
java -Dwebdriver.chrome.driver=C:\Users\Bhagyashree\workspace\SeleniumFramework\src\main\resources\chromedriver.exe -Dwebdriver.ie.driver=C:\Users\Bhagyashree\workspace\SeleniumFramework\src\main\resources\IEDriverServer.exe -Dwebdriver.firefox.driver=C:\Users\Bhagyashree\workspace\SeleniumFramework\src\main\resources\geckodriver.exe -jar selenium-server-standalone-3.14.0.jar -role node -hub http://192.168.0.73:2222/grid/register -port 5555
```

Open a new command prompt and type the above command. The snapshot shown below is of the console. It shows that the node is registered with the hub and ready to use.

```
13:08:01.460 INFO [GridLauncherU3$3.launch] - Launching a Selenium Grid node on port 5555
2020-01-04 13:08:02.962:INFO::main: Logging initialized @2134ms to org.seleniumhq.jetty9.util.log.StdErrLog
13:08:03.303 INFO [SeleniumServer.boot] - Selenium Server is up and running on port 5555
13:08:03.304 INFO [GridLauncherU3$3.launch] - Selenium Grid node is up and ready to register to the hub
13:08:03.819 INFO [SelfRegisteringRemote$1.run] - Starting auto registration thread. Will try to register every 5000 ms.
13:08:03.820 INFO [SelfRegisteringRemote.registerToHub] - Registering the node to the hub: http://192.168.0.73:2222/grid/register
13:08:05.375 INFO [SelfRegisteringRemote.registerToHub] - The node is registered to the hub and ready to use
```

**Figure 12.9**

Navigate to **http://192.168.0.73:2222/grid/console** and see that by default, 5 Chrome, 5 Firefox, and 1 Internet Explorer instances are displayed.



**Figure 12.10**

We will now have a look at a few hub configuration parameters.

## Hub configuration parameters

In this section, we talk about some of the configuration parameters on the hub side.

### WaitTimeout of a new session

When a node is held up executing other test scripts, the latest test script will wait for the node to become available. There is no wait timeout by default; that is, the test script waits for the node to be available for an indefinite amount of time. To change that behavior and to let the test script throw an exception if it doesn't find the node within a limited time, Selenium Grid specifies a configuration that enables the test script to do so. The configuration parameter is newSessionWaitTimeout. The command for that is as follows:

```
java -jar selenium-server-standalone-3.14.0.jar -role hub -port 2222 -newSessionWaitTimeout 180000
```

Here, the test script will wait for three minutes before throwing an exception, saying it couldn't obtain a node to execute itself.

## [Waiting for DesiredCapability match](#)

When the test script asks for a test platform with the desired capability, the hub rejects the request if a suitable node with the desired capability is not found.

Modifying the value for the `throwOnCapabilityNotPresent` parameter changes this behavior. It is set to true by default, which means the hub will reject the request if it doesn't find a suitable node with the specified capability. Changing this parameter to false queues the request, and the hub waits until a node with that capability is added to the grid. The command that has to be specified is as follows:

```
java -jar selenium-server-standalone-3.14.0.jar -role hub -port 2222 -  
throwOnCapabilityNotPresent false
```

Now the hub does not reject the request but places the request in a queue and waits until the requested platform is available.

## Node configuration parameters

In this section, we will go through the configuration parameters for a node.

## Changing the default number of browsers

We already saw that, by default, there are 11 instances of browsers getting registered to a node. In this section, we will see how many instances of each browser we can allow in our node. For this to happen, Selenium Grid comes with a configuration parameter called maxInstances, using which we can specify the instances of a particular browser we want our node to provide.

The command to do that is given below:

```
java -  
Dwebdriver.chrome.driver=C:\Users\Bhagyashree\workspace\Selenium  
Framework\src\main\resources\chromedriver.exe -  
Dwebdriver.ie.driver=C:\Users\Bhagyashree\workspace\SeleniumFrame  
work\src\main\resources\IEDriverServer.exe -  
Dwebdriver.firefox.driver=C:\Users\Bhagyashree\workspace\SeleniumFr  
amework\src\main\resources\geckodriver.exe -jar selenium-server-  
standalone-3.14.0.jar -role node -hub  
http://192.168.0.73:2222/grid/register -port 5555 -browser  
browserName=chrome,maxInstances=2,platform=WINDOWS -browser  
browserName="internet  
explorer",maxInstances=2,platform=WINDOWS -browser  
browserName=firefox,maxInstances=2,platform=WINDOWS
```

Here we have allowed 2 instances of Chrome, Internet Explorer and Firefox each

It can also be seen that to register your node with the browsers of your choice, Selenium Grid provides a browser option called using which we can achieve this

## **Node timeouts**

This parameter is set when registering a node with a hub. The value provided to these parameters is the time in seconds that a hub actually waits before it terminates a test script execution on a node if the test script fails to perform any kind of activity on the node.

The command for configuring Node Timeout is given below. Here, we are registering a node with a node timeout value of 300 seconds. The hub terminates the test script if it doesn't perform any activity on the node for more than 500 seconds.

```
java -jar selenium-server-standalone-3.14.0.jar -role node -hub  
http://192.168.0.73:2222/grid/register -nodeTimeout 500
```

### **Browser timeouts**

This configuration lets the node know how long it should wait before it ends a test script session when the browser appears to hang. After this time, the node terminates the browser session and continues with the next waiting test script. The configuration parameter for this is The command is as follows:

```
java -jar selenium-server-standalone-3.14.0.jar -role node -hub  
http://192.168.0.73:2222/grid/register –browserTimeout 100
```

We have thus looked at some of the configuration parameters that you can specify at the node's end to have better control over the Selenium Grid environment.

## Registering a node automatically

If there is a failure in the Hub after a node registers to it, all the information of the nodes that is already registered is lost. Going back to each of the nodes and re-registering them manually is time-consuming. So, to handle this kind of situation, Selenium Grid provides a configuration parameter to a node, through which we can specify the node to re-register itself automatically to the hub after a specified period of time. The default time of reregistration is five seconds. This way, we don't have to worry; even if the hub crashes or there is a restart, our node will try to reregister every five seconds.

If you want to alter this time interval, the configuration parameter to deal with is The command to specify this is as follows:

```
java -jar selenium-server-standalone-3.14.0.jar -role node -hub  
http://192.168.0.73:2222/grid/register -registerCycle 10000
```

### [Unregister an unavailable node](#)

The unregisterIfStillDownAfter configuration will make the hub unregister the node if the poll doesn't produce an expected result. Let's say a node is down, and the hub tries to poll the node and is unable to connect to it after many attempts. At this point, how long the hub is going to poll for the availability of the node is determined by the unregisterIfStillDownAfter parameter. Beyond this time, the hub will unregister the node.

The command to do that is as under:

```
java -jar selenium-server-standalone-3.14.0.jar -role node -hub  
http://192.168.0.73:2222/grid/register -nodePolling 10 -  
unregisterIfStillDownAfter 40000
```

Here, the hub will poll the node every ten seconds; if the node is down, the polling will continue for 40 seconds, that is, the hub will poll four times and then unregister the node from the grid.

## Specifying configuration using JSON files

There are two ways to provide the configuration parameter to the Selenium Grid's hub and node. The first one is specifying the configuration parameters over the command line. The second way of doing it is by using a JSON file that contains all the configuration parameters.

A node configuration file (say, — a typical JSON file having all the configuration parameters — looks similar to the one shown below:

```
{  
  "class": "org.openqa.grid.common.CustomRequest",  
  "capabilities": [  
    {  
      "seleniumProtocol": "WebDriver",  
      "browserName": "chrome",  
      "version": "67",  
      "maxInstances": 3,  
      "platform" : "WINDOWS"  
    }  
  ],  
  "configuration": {  
    "port": 2222,  
    "register": true,  
    "host": "192.168.1.100",  
    "proxy": "org.openqa.grid.selenium.proxy.
```

```
    "DefaultRemoteProxy",
    "maxSession": 3,
    "hubHost": "192.168.1.101",
    "role": "webdriver",

    "registerCycle": 8000,
    "hub": "http://192.168.1.101:1111/grid/register",
    "hubPort": 2222,
    "remoteHost": "http://192.168.1.102:6666"
}
```

Once these files are configured, they can be provided to the node and the hub, using the following command:

```
java -jar selenium-server-standalone-3.14.0.jar -role node -
nodeConfigcustomConfig.json
```

## [\*Changes to the SingletonDriver class\*](#)

Previously we created the SingletonDriver.java Selenium driver class. The class has the setDriver method that takes the parameters passed into the suite for the browser, mobile device, platform, and environment, and process those while creating the driver instance.

Now, when running in a remote environment, we need to add several conditions to the setDriver methods to pass the desired capabilities and preferences to the RemoteWebDriver class, instead of the local WebDriver instance.

Let's look at these conditions for each setDriver method in this class.

## Changes to the setDriver method

In the main setDriver method, we had set up a series of switch cases for each browser. In those cases, we set the browser options. Once that was done, we cast them to the local and it was launched.

Now, we need to check and see if the environment parameter is passed in as local or remote and cast caps to the correct driver. The excerpt shown below is from the SingletonDriver class.

```
public final void setDriver(String browser, String environment,
String platform, MapObject>... optPref) throws Exception {
DesiredCapabilities caps = null;
FirefoxOptions firefoxopt = null;
ChromeOptions chromeopt = null;
InternetExplorerOptions ieopt = null;
SafariOptions safariopt = null;
EdgeOptions edgeopt = null;
String localHub = "http://127.0.0.1:4723/wd/hub";
String getPlatform = null;
String ffVersion = "55.0";
String remoteHubURL = "http://192.168.0.73:2222/wd/hub";
switch (browser) {
case "firefox":
if (environment.equalsIgnoreCase("local")) {
firefoxopt = new FirefoxOptions();
webDriver.set(new FirefoxDriver(firefoxopt));
```

```
break;  
} else {  
  
    caps.setCapability("browserName", browser);  
  
    caps.setCapability("version", ffVersion);  
    caps.setCapability("platform", platform);  
    caps.setCapability("applicationName",  
        platform.toUpperCase() + "-" +  
        browser.toUpperCase());  
  
    webDriver.set(new RemoteWebDriver(  
        new URL(remoteHubURL), caps));  
  
    ((RemoteWebDriver) webDriver.get()).setFileDetector(  
        new LocalFileDetector());  
}  
}  
}  
}  
}
```

In this example, the Firefox driver capabilities were set in the switch statement and either cast to local WebDriver or, if running remotely on the grid, cast to

The remote hub URL was passed to along with several capabilities that would cause the Selenium hub to direct traffic to the respective node. Those are version, platform, and

Also, RemoteWebDriver calls which allowed files residing in the local workspace to be uploaded to the application remotely.

## Changes to the TestNG XML

The default settings for the browser, platform, and environment by the parameters shown below

```
name="Parameter test Suite" verbose="1" parallel="false">
  name="suite-param" value="parameter at suite level" />
  name="Group Test1">
    name="GroupA"/>
    name="param1" value="GroupA" />
    name="browser" value="chrome" />
    name="platform" value="windows" />
    name="environment" value="remote" />

  name="keywordframework.TestClass2">
```

## Class for default global variables

In case the browser, platform, and environment are not set anywhere, and the user runs a test class or suite without them, then there should always be a default constant for browser, platform, and environment so that the test will run. Usually, that is set to the default development environment platform.

```
public class Default_Values {  
    public static final String BROWSER = "chrome";  
    public static final String PLATFORM = "Windows 10";  
    public static final String ENVIRONMENT = "local";  
    public static String DEFT_BROWSER = null;  
    public static String DEFT_PLATFORM = null;  
    public static String DEFT_ENVIRONMENT = null;  
}
```

## Fetching runtime parameters

There needs to be a place to process the system properties, suite parameters, or default variables when the test suite is run and passed to the setDriver method. This can be done by creating a Base Test class

As we are switching from a local run to a remote run on the Selenium Grid, the Global\_VARS.DEF\_ENVIRONMENT variable should be set as shown below

```
public class BaseClass {  
    @Parameters({"browser","platform","environment"})  
    @BeforeSuite(alwaysRun=true, enabled=true)  
    protected void suiteSetup(@Optional(Default_Values.BROWSER)  
    String browser,  
    @Optional(Default_Values.ENVIRONMENT) String  
    environment, @Optional(Default_Values.PLATFORM) String platform  
    )  
    throws Exception {  
  
    Default_Values.DEFT_BROWSER = System.getProperty("browser",  
    browser);  
    Default_Values.DEFT_PLATFORM = System.getProperty("platform",  
    platform);  
    Default_Values.DEFT_ENVIRONMENT =  
    System.getProperty("environment", environment);
```

```
SingletonDriver.getInstance().setDriver(Default_Values.DEFT_BROWSER,  
Default_Values.DEFT_ENVIRONMENT,Default_Values.DEFT_PLATFORM  
);  
  
}  
}
```

Our test class should extend this as shown below:

```
public class TestClass2 extends BaseClass
```

Having seen the local grid setup, we now explore the grid in the cloud using BrowserStack.

## [\*Introducing BrowserStack\*](#)

BrowserStack supports Selenium, and it is used for running Selenium WebDriver test cases in the cloud. It is mainly used for cross-browser testing where we not only need to test on various browsers like Safari, Microsoft Edge, etc. but also on specific versions of those. It also supports Android and IOS testing. With BrowserStack, there is no need to have all browsers and all mobile emulators available. The ones provided by BrowserStack can be utilized. Cross-browser testing is mostly used in companies to ensure that their website works fine on a variety of browsers.

BrowserStack allows you a free trial period with 100 hours of testing. The usage of BrowserStack is quite simple. Let's see the setup required.

## [Setting-up BrowserStack](#)

The prerequisites for setting up BrowserStack are given below:

BrowserStack account

Browser StackUserName and AccessKey

Selenium WebDriver JARS

TestNG JARS

Let's have a look at the steps to use BrowserStack:

### **STEP 1:**

Signup for the trial account on BrowserStack and note down the UserName and This process is fairly straightforward. Signup is possible through a Google account as well.

### **STEP 2:**

Create a Maven Project. We already have one.

### **STEP 3:**

Add the lines shown below to the SingletonDriver class:

```
public static final String USERNAME = "pinakinchaubal1";
public static final String AUTOMATE_KEY =
"2igoYXZ1GCa3Yc7JyWFp";
public static final String URL = "https://" + USERNAME + ":" +
AUTOMATE_KEY + "@hub.browserstack.com/wd/hub";
```

**STEP 4:**

Add the lines shown below to the method annotated with @Test:

```
DesiredCapabilities cap = new DesiredCapabilities();
cap.setPlatform(Platform.MAC);
cap.setBrowserName("firefox");
cap.setVersion("38");

cap.setCapability("browserstack.debug", "true");
// Create a URL object
URL browserStackUrl = new URL(URL);
// Create object of driver. We execute scripts remotely. So we use
RemoteWebDriver
driver = new RemoteWebDriver (browserStackUrl, capability);
```

**STEP 5:**

Make changes to the SingletonDriver as shown below in the case for Firefox:

```
case "firefox":  
    caps = DesiredCapabilities.firefox();  
    if (environment.equalsIgnoreCase("local")) {  
        firefoxopt = new FirefoxOptions();  
        webDriver.set(new FirefoxDriver(firefoxopt));  
        break;  
    } else if (environment.equalsIgnoreCase("remote")) {  
        caps.setCapability("browserName", browser);  
        caps.setCapability("version", ffVersion);  
        caps.setCapability("platform", platform);  
        caps.setCapability("applicationName",  
            platform.toUpperCase() + "-" +  
            browser.toUpperCase());  
        webDriver.set(new RemoteWebDriver(  
            new URL(remoteHubURL), caps));  
        ((RemoteWebDriver) webDriver.get()).setFileDetector(  
            new LocalFileDetector());  
        break;  
    } else {  
        caps.setCapability("browserName", browser);  
  
        caps.setCapability("version", ffVersion);  
        caps.setCapability("platform", System.getenv(platform));  
        caps.setCapability("applicationName",  
            platform.toUpperCase() + "-" +  
            browser.toUpperCase());  
        webDriver.set(new RemoteWebDriver(  
            new URL(URL), caps));  
        ((RemoteWebDriver) webDriver.get()).setFileDetector(  
            new LocalFileDetector());  
        break;  
    }  
}
```

}

## STEP 6:

Change the value of environment to bstack and platform to ANY as shown below:

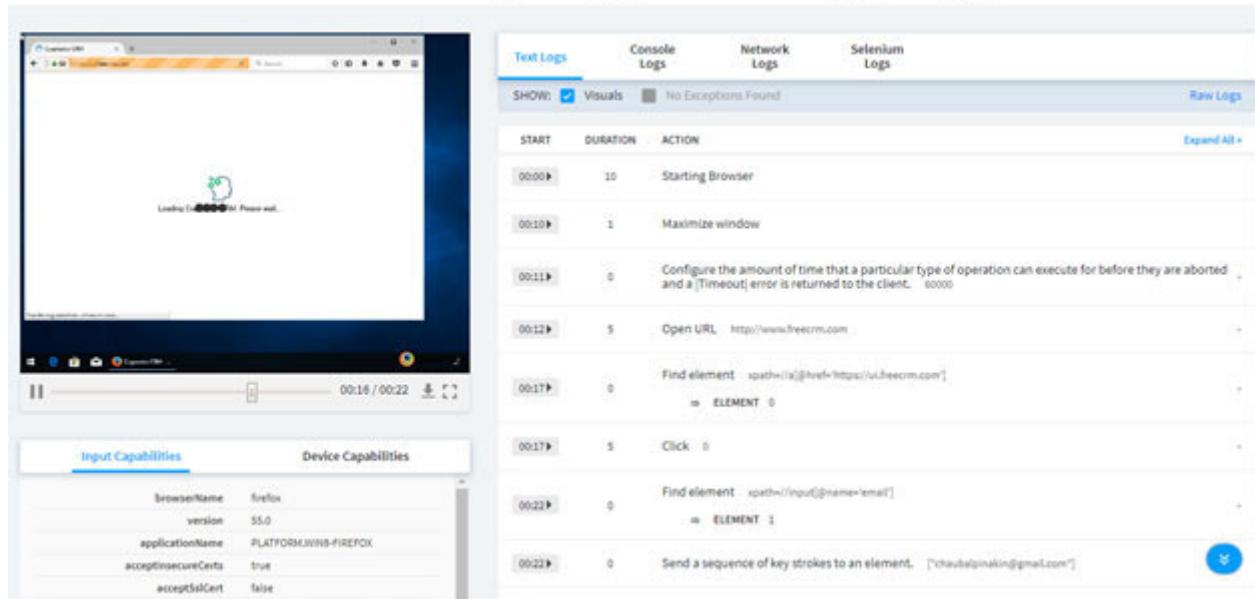
```
name="environment" value="bstack" />  
name="platform" value="ANY"/>
```

Execute the test and get the output in Browser Stack Dashboard as shown below:

SESSION NAME	SESSION STATUS	REST API	OS	BROWSER/DEVICE	DURATION	LAST UPDATED	ACTION
1de024daa84e0339419948c853bc2d8862e 0cd83 By: Pinakin Chauhan	COMPLETED	UNMARKED	WINDOWS 10	FIREFOX 55.0	0h 0m 27s	2 minutes ago	

**Figure 12.11**

Click on the entry shown above



**Figure 12.12**

A video showing the execution and the logs get generated as shown in the above snapshot.

**Please Note: For parallel execution, simply make the parallel="tests" in the TestNG XML**

With this, we come to the end of this chapter. This was an important chapter with regards to Selenium Grid.

## Conclusion

This chapter covered several concepts related to RemoteWebDriver and Selenium Grid. We saw how a RemoteWebDriver works and how to convert existing WebDriver scripts to use the We then moved on to the Selenium Grid and understood what a hub and node are. We understood various configuration parameters for the Hub and the Node. We finally had a look at how to execute test cases in the cloud using BrowserStack.

The next chapter will be about Executing test cases on Android and IOS devices or emulators using Appium. It will be fun to understand the exciting world of Mobile Automation in a nutshell.

## Questions

What is the use of Try converting one of your existing test cases to use RemoteWebDriver

What is a Selenium Grid? Explain with an example.

What are Hub and Nodes in a Selenium Grid?

Apart from the configuration parameters discussed in this book, try to learn other configuration parameters for the Hub and Node

Try executing a sample test case in the cloud using BrowserStack

**Mobile Test Automation Using Appium**

Till now, we have seen desktop browser automation. Now we move on to automating mobile apps. With the increasing number of mobile users, businesses worldwide have to serve their users on mobile devices as well. In this chapter, we will start with a brief introduction to Appium, look at Appium architecture, setup Appium and then write a small script to start the freecrm app. We will then see the code for the framework.

This is an important chapter from learning how to execute on a mobile device. This chapter will highlight the changes required for running your Selenium tests from an Android and IOS mobile.

After reading this chapter, the reader will be conversant with the Appium and create scripts for Mobile devices.

## **Structure**

Here's what we will learn in this chapter:

Types of Mobile applications

Introducing Appium

Learning the Appium architecture

Setting up Appium

Changes to pom.xml for Appium

Changes to the framework

## Objectives

Learn the types of Mobile applications

Learn the concepts of Appium, and it's architecture

Setup Appium

Create a simple script to work with Appium

Changes to the framework to incorporate Appium

We begin with an introduction to the different types of Mobile Applications.

## Types of mobile applications

There are three different forms in which an application can reach the end-user on a mobile platform:

**Native apps:** Native apps are specific to the mobile target platform on which they run. They are developed in the languages that the platform supports and are hard-wired to underlying SDKs. For iOS, applications are written in the Objective-C or Swift programming language and depend on iOS SDK; similarly, for Android, these apps are developed in Java or Kotlin and depend on Android SDK.

**m.website:** These are also known as mobile websites, and these are mini versions of a web application that loads on the browsers of your mobile devices. On iOS devices, these browsers can be Safari or Chrome, and on Android devices, these browsers can be the Android default browser or Chrome

**Hybrid apps:** The Hybrid app is a combination of the native app and the web app. When you develop a native app, HTML web pages are loaded into the app by some parts of it, which makes the users feel that they are using a native application. WebViews in native apps are used to load the web pages.

Let's move on to an introduction to Appium.

## [\*Introduction to Appium\*](#)

Appium is an open-source test automation tool developed and supported by Sauce Labs to automate native and hybrid mobile apps better known as a Cross-Platform Mobile Automation Tool. It internally uses the JSON wire protocol to interact with iOS and Android native apps using the Selenium WebDriver.

Appium, a node.js server, Automates hybrid and native mobile applications for Android and iOS. One of the advantages of Appium is that test scripts can be written in any framework or languages like Ruby on Rails, C#, and Java without having to modify the apps for automation purposes. The interaction between node.js server and Selenium client libraries ultimately works together with the mobile application. Appium is open source and can seamlessly run on a variety of devices and emulators, thus making it an apt choice for mobile test automation.

## Learning about Appium architecture

Now that we have an idea of what Appium is let's take a look at the architecture of Appium and how it is used to support mobile app automation. Appium is a web server written in Node.js. The server performs actions in the given order:

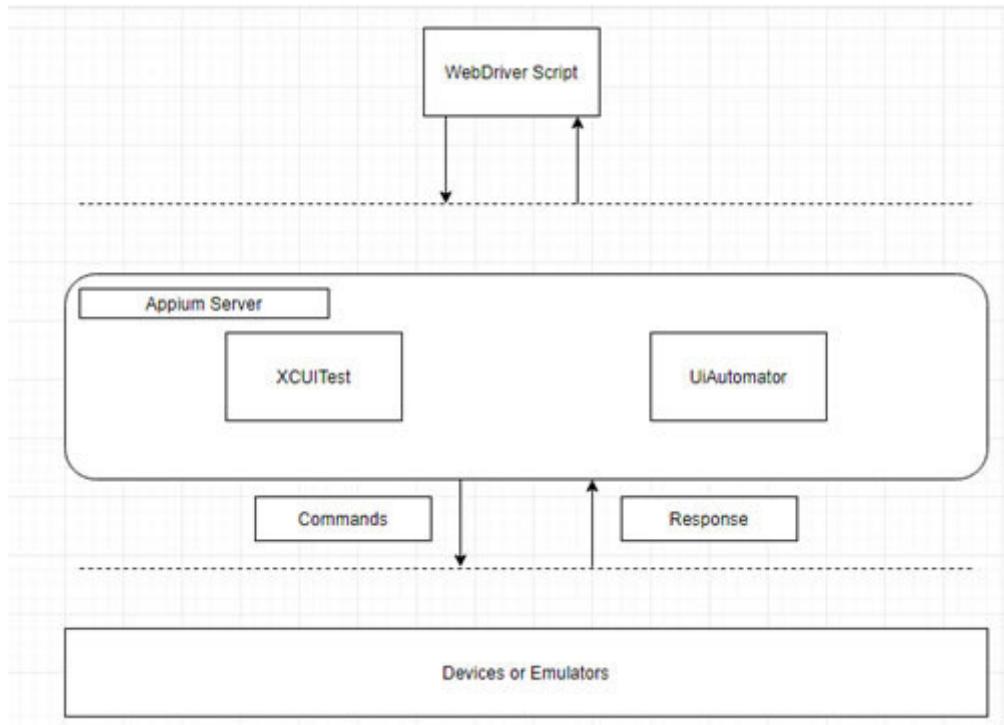
Initiates a session on receiving commands from the client

Listens for all commands issued

Executes the issued commands

Returns the execution status of the command

The Appium server receives a connection from the client in the form of a JSON object over the HTTP protocol. Once the server receives the request, it creates a session, as specified in the JSON object, and returns the session ID, which will be maintained until the Appium server session lasts. So, all testing will be happening in the context of this newly created session. The following is a diagram depicting the Appium architecture:



**Figure 13.1**

The Appium server is written in Node.js, and it can be installed via npm or by downloading.

## [Understanding UIAutomator2](#)

UIAutomator 2 is an automation framework that is based on Android instrumentation and allows the test script developer to build and run UI tests.

Appium uses Google's UIAutomator to execute commands on real devices and emulators. UIAutomator is a test framework from Google for native app automation at the UI level. Typical usage of UIAutomator2 would be to pass the following in desired capabilities:

```
automationName: uiautomator2
```

Starting with version 1.6, Appium has provided support to UIAutomator 2. Appium internally uses the appium-android-bootstrap module to interact with UI Automator. It lets commands to be sent to the device, which then gets executed on real devices using Android's UIAutomator testing framework.

When the Appium client requests a new AndroidDriver session, the client library passes the desired capability to the Appium server. The UIAutomator2 driver module then creates the session. It installs the UIAutomator2 server APK file on the Android device that is connected, starts the Netty server, and triggers a session. Once the Netty server session is started, the UIAutomator2 server

continues to listen on the device for requests and responds with the appropriate response:

## [Understanding XCUI Test](#)

XCUITest is an automation framework that has been introduced by Apple with the iOS 9.3 version.

Typical usage would be to pass the following in desired capabilities:

```
automationName: XCUI Test
```

Facebook WebDriverAgent is a WebDriver server implementation for iOS. It is used to control connected devices or simulators and allows the user to launch an app, perform commands (such as tap, double-tap, and scroll), as well as kill applications.

The UIAutomation library communicates with the bootstrap.js file, which is running inside the device or simulator to perform the commands received by the Appium client libraries:

## Setting up Appium

There are two ways to setup Appium:

Using NodeJS

Download Appium desktop client

## Using NodeJS to install Appium

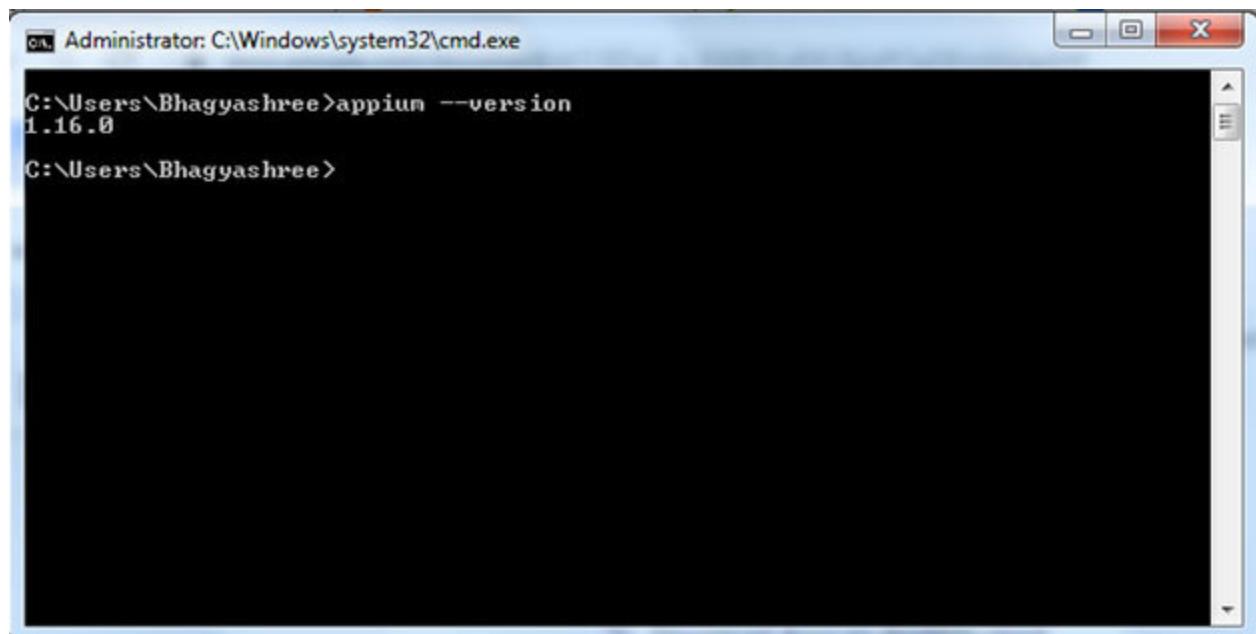
Follow the steps below to install the Appium server:

Make sure NodeJS is installed. For NodeJS installation instructions, go to <https://nodejs.org/en/download/>

Open a command prompt and type the command shown below

```
npm install -g appium
```

Once the installation completes, verify the Appium installation by typing appium --version



*Figure 13.2*



### Downloading Appium desktop client

Go to <http://appium.io/downloads.html> and click on the link shown below. Follow the instructions online.

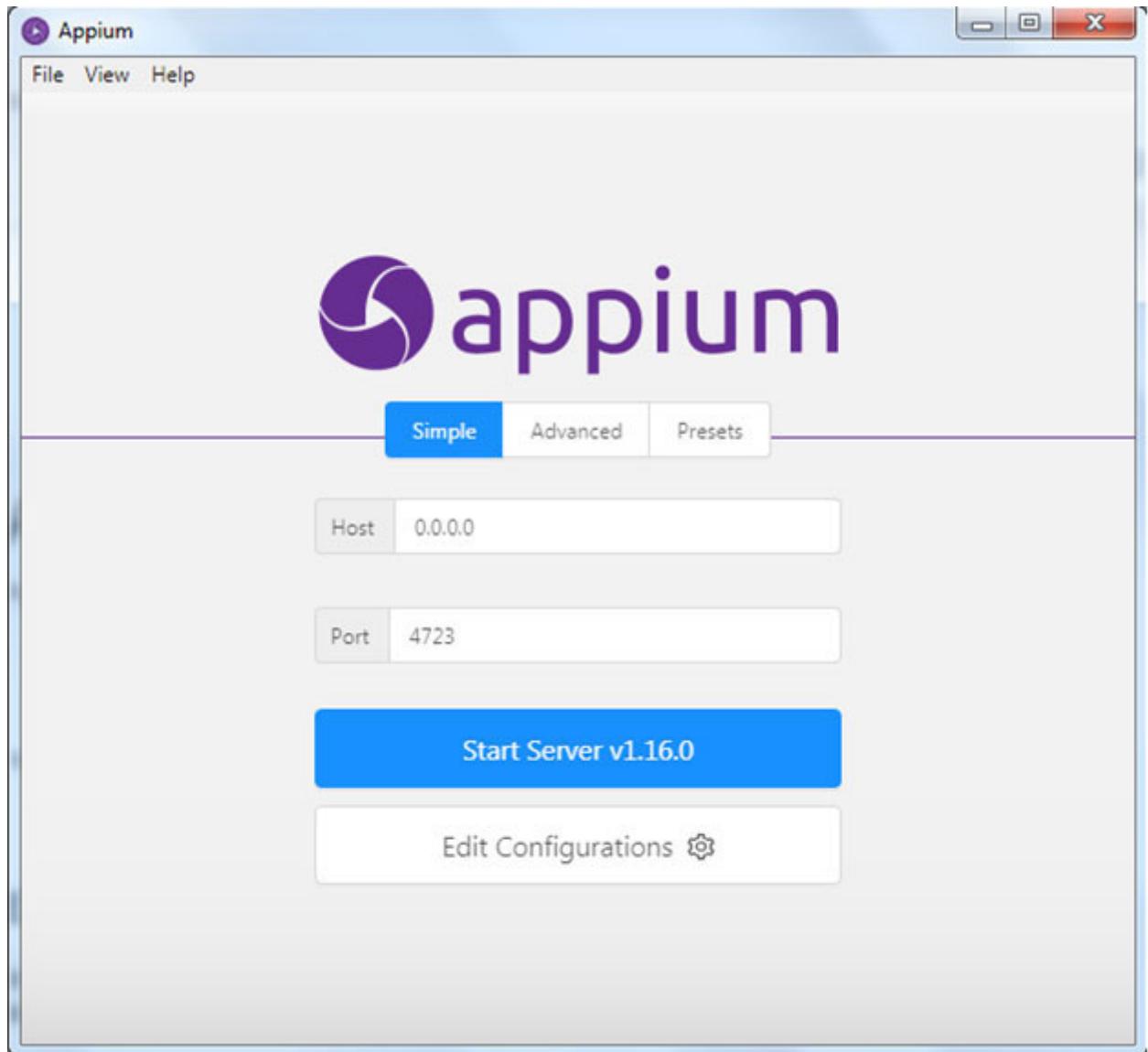
## Appium Desktop Apps

Appium's desktop app supports OS X, Windows and Linux

- [Appium-Desktop for OSX, Windows and Linux](#)

***Figure 13.3***

After successful installation of the Appium desktop client, it displays as shown below:

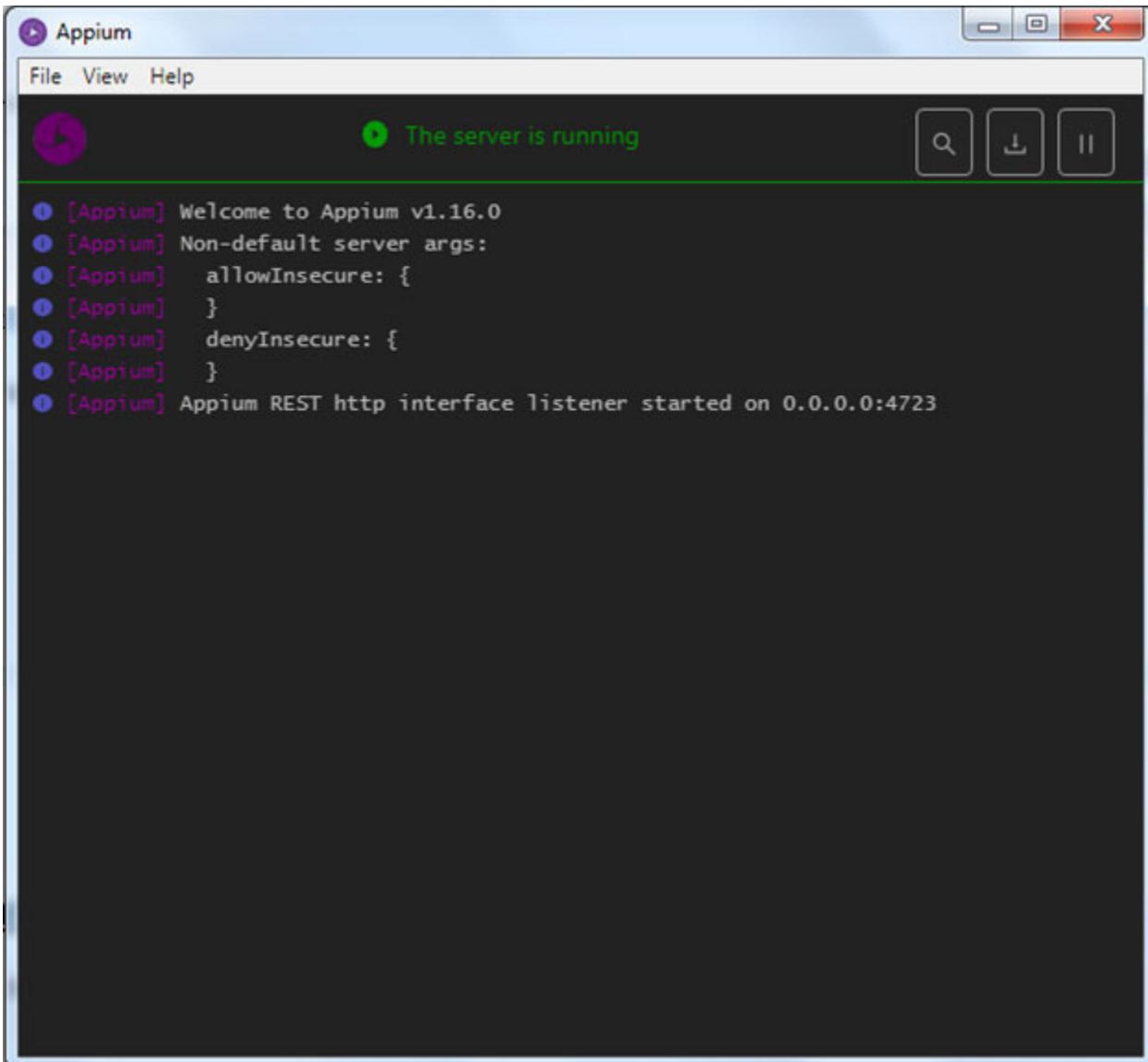


**Figure 13.4**

Now let's start the Appium Server.

### Start the Appium Server

Click the **Start Server** button in the above snapshot. The Appium server starts and the console display in the Appium Desktop client displays as shown below.



The screenshot shows the Appium desktop application window. The title bar says "Appium". The menu bar has "File", "View", and "Help". The main area has a dark background with white text. At the top center, there is a green circular icon with a play symbol and the text "The server is running". To the right of this are three small icons: a magnifying glass, a downward arrow, and a double-lined square. Below this, there is a list of log messages:

```
● [Appium] Welcome to Appium v1.16.0
● [Appium] Non-default server args:
● [Appium]   allowInsecure: {
● [Appium]     }
● [Appium]   denyInsecure: {
● [Appium]     }
● [Appium]   Appium REST http interface listener started on 0.0.0.0:4723
```

*Figure 13.5*

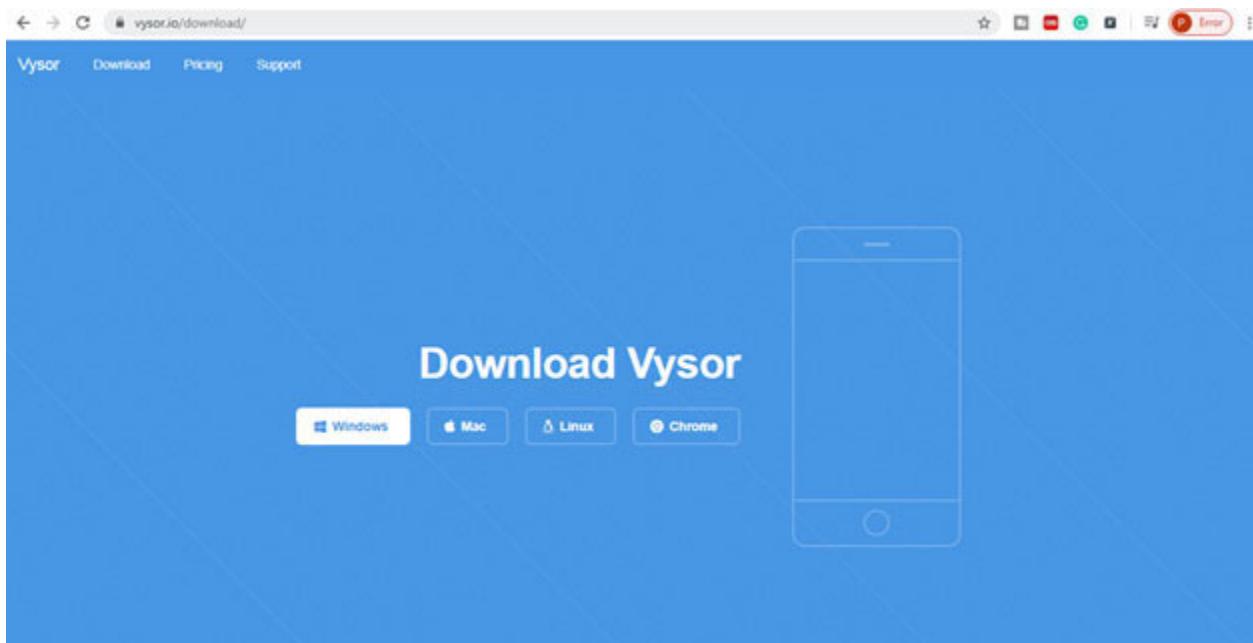
The Appium server can also be started by typing Appium at the command prompt.

Let's see how we can connect a real Android device on Windows

## Connecting a real Android device on Windows

To connect a real Android device on Windows, follow the steps below:

Install a software called Vysor, which will help cast the mobile screen on the desktop or laptop. The installation instructions can be found at [vysor.io/download](https://vysor.io/download). Vysor download page looks as shown below.

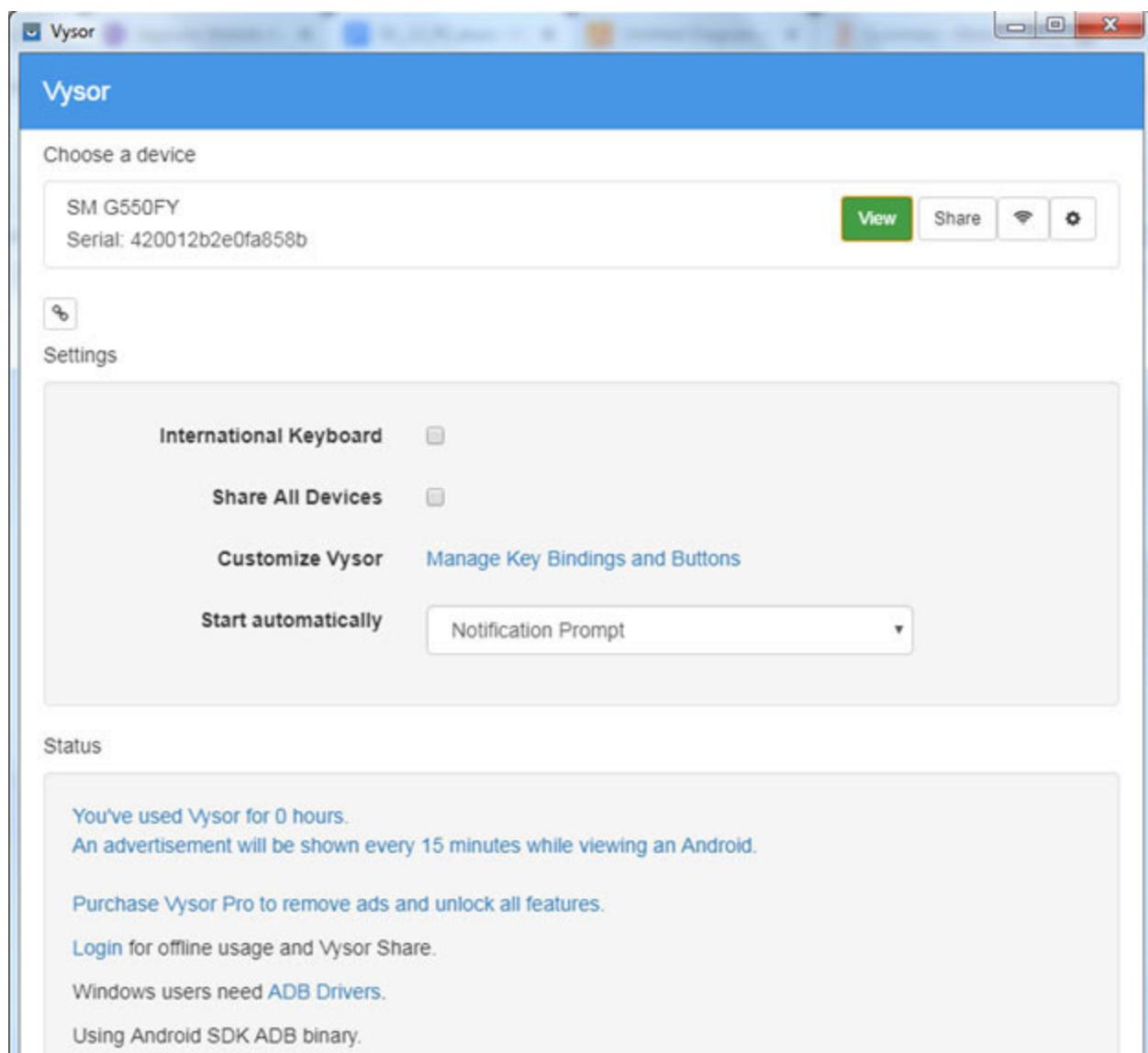


**Figure 13.6**

On the Android phone, go to **Settings | About Phone | Software Information** and click on **Build number** 7 times to enable **Developer options**

Navigate to **Developer options** and enable **USB debugging**

With this, your mobile phone is all set to work with Appium. Connect a USB cable and start Vysor. The Vysor screen should display as shown below



*Figure 13.7*

Click on and your mobile screen gets projected on your laptop/desktop, as shown below:



*Figure 13.8*

This completes Vysor setup

Next, download the SDK tools for windows. Goto <https://developer.android.com/studio> and download the SDK tools from the section shown below:

### Command line tools only

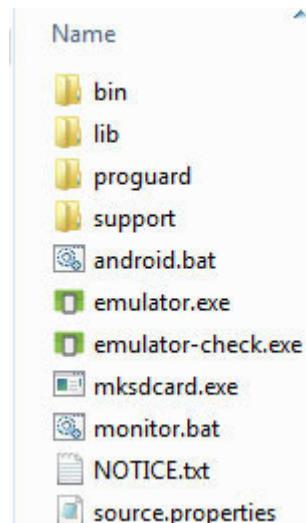
If you do not need Android Studio, you can download the basic Android command line tools below. You can use the included `sdkmanager` to download other SDK packages.  
These tools are included in Android Studio.

Platform	SDK tools package	Size	SHA-256 checksum
Windows	<a href="#">sdk-tools-windows-4333796.zip</a>	148 MB	7e81d69c303e47a4f0e748a6352d85cd0c8fd90a5a95ae4e07bb5e5f960d3c7a
Mac	<a href="#">sdk-tools-darwin-4333796.zip</a>	98 MB	ebc29358bc0f13d7c2fa0f9290135a5b608e38434aad9bf7067d0252c160853e
Linux	<a href="#">sdk-tools-linux-4333796.zip</a>	147 MB	92ffee5a1d98d856634e8b71132e8a95d96c83a63fde1099be3d86df3106def9

**Figure 13.9**

Extract the zip folder to an appropriate folder. The folder looks like below

It contains a tools folder which has the contents shown below



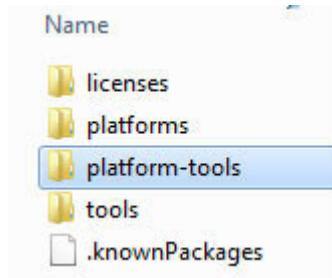
**Figure 13.10**

Goto the command prompt and type the command `SDKManager "platform-tools"`. The Android version for this example is 7.1.1. Notice that the number 28 indicates the API level, which can be found from the Android version, as shown below. Notice that the API level is 25 for version 7.1.1, but since downward compatibility API level, 25 will work fine here.

Code name	Version numbers	Initial release date	API level	References
No codename	1.0	September 23, 2008	1	[9]
	1.1	February 9, 2009	2	[9][11]
Cupcake	1.5	April 27, 2009	3	
Donut	1.6	September 15, 2009	4	[12]
Eclair	2.0 – 2.1	October 26, 2009	5 – 7	[13]
Froyo	2.2 – 2.2.3	May 20, 2010	8	[14]
Gingerbread	2.3 – 2.3.7	December 6, 2010	9 – 10	[15]
Honeycomb	3.0 – 3.2.6	February 22, 2011	11 – 13	[16]
Ice Cream Sandwich	4.0 – 4.0.4	October 18, 2011	14 – 15	[17]
Jelly Bean	4.1 – 4.3.1	July 9, 2012	16 – 18	[18]
KitKat	4.4 – 4.4.4	October 31, 2013	19 – 20	[19]
Lollipop	5.0 – 5.1.1	November 12, 2014	21 – 22	[20]
Marshmallow	6.0 – 6.0.1	October 5, 2015	23	[21]
Nougat	7.0	August 22, 2016	24	[22]
	7.1.0 – 7.1.2	October 4, 2016	25	[23][24][25]

**Figure 13.11**

The above command will extract the platform-tools folder, as shown below.



**Figure 13.12**

Create an environment variable ANDROID\_HOME and set it's value to the SDK tools folder. Append the location of the platform-tools folder to the Path system variable.

Make sure the mobile device is connected to the machine with a USB cable and open a command prompt and type adb devices [adb stands for Android Debug Bridge]. The list of mobile devices attached to the machine gets displayed as shown below

```
C:\>adb devices
List of devices attached
420012b2e0fa858b    device
```

A screenshot of a Windows Command Prompt window titled 'Administrator: Command Prompt'. The window shows the command 'adb devices' being run, which lists a single device with the identifier '420012b2e0fa858b' and the status 'device'. The window has a standard Windows title bar and scroll bars.

### ***Figure 13.13***

Let's create a simple test to be run using Appium.

## [Creating a simple test](#)

Follow the steps to create a test using Appium:

Add the Appium Java client to as shown below. Right-click on project and click **Maven | Update Project**

io.appium  
java-client  
7.3.0

Create a test class in IDE:

```
public class TestClass4 {  
    AppiumDriver driver=null;  
    @Test  
    public void test1() {  
        DesiredCapabilities caps = new DesiredCapabilities();  
        caps.setCapability("deviceName", "Galaxy On5 Pro");  
        caps.setCapability("udid", "420012b2e0fa858b");  
        caps.setCapability("platformName", "Android");  
        caps.setCapability("platformVersion", "7.1.1");  
        caps.setCapability("appPackage",  
            "com.cogmento.app");  
        caps.setCapability("appActivity",  
            "com.cogmento.app.MainActivity");
```

```

URL url = null;
try {
url = new URL("http://127.0.0.1:4723/wd/hub");

} catch (MalformedURLException e) {
// TODO Auto-generated catch block
e.printStackTrace();
}
driver = new AppiumDriver(url,caps);
System.out.println("Application has started...");
```

}

}

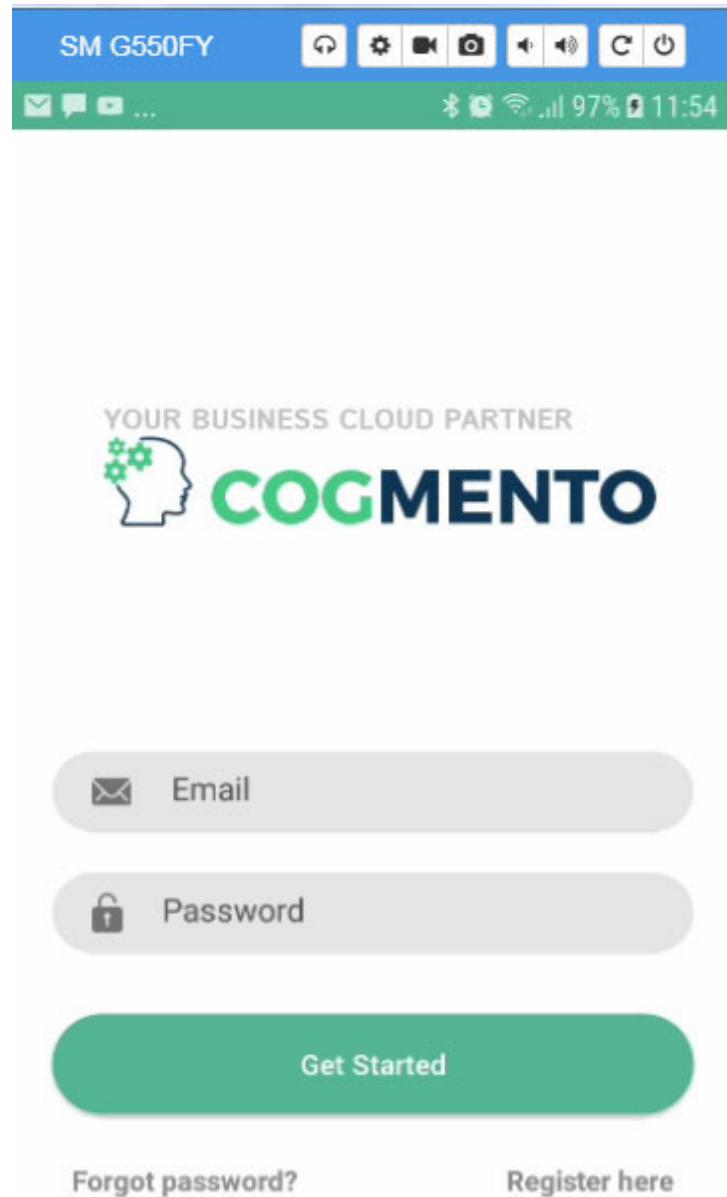
Here we are taking example of the freecrm application. We will now create a TestNG XML file as shown below:

```

version="1.0" encoding="UTF-8"?>
suite SYSTEM "http://testng.org/testng-1.0.dtd">
name="Suite">
name="Test">

name="keywordframework.TestClass4"/>
```

When we run the TestNG xml by **Run As | TestNG** the freecrm application starts on the mobile device as shown below:



**Figure 13.14**

Now that we were able to load the app through Appium, we will see how to locate elements using the Appium Inspector. Follow the steps below to locate elements using Appium Inspector:

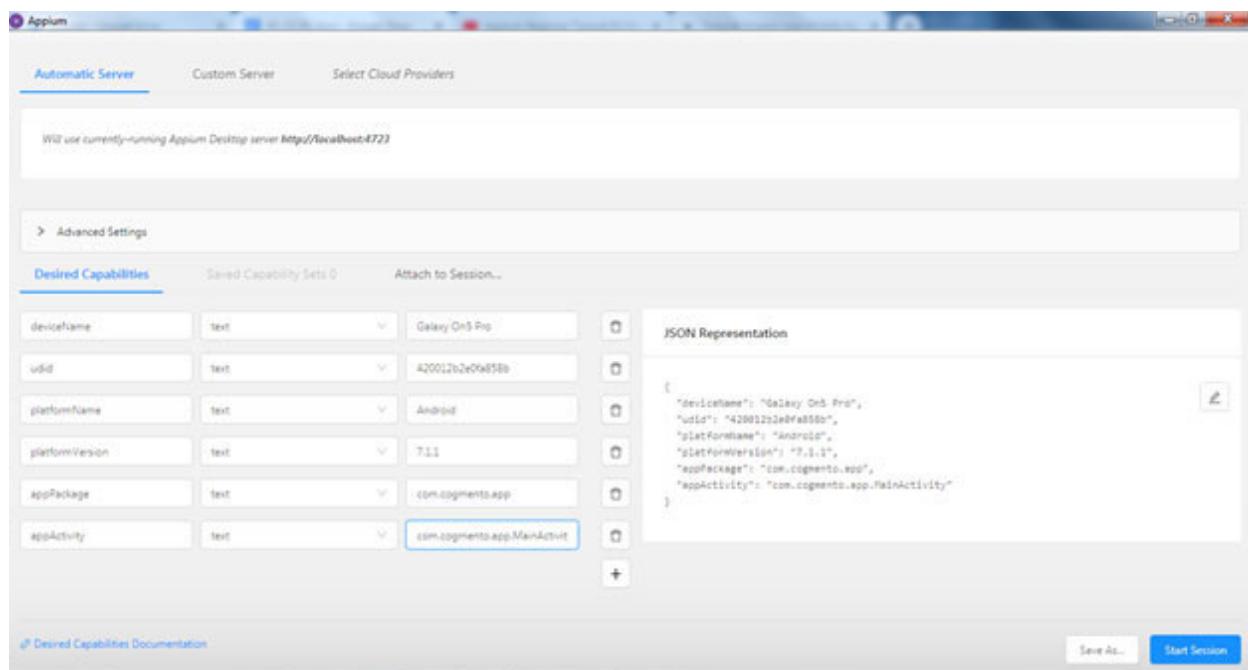
From the Appium Desktop client, open Appium Inspector by clicking the highlighted button shown below:

The screenshot shows the Appium Inspector window. At the top, there's a status message: "The server is running". To the right of the status are three icons: a magnifying glass (highlighted with a yellow box), a downward arrow, and a double vertical bar. Below the status, the title "session list" is followed by a detailed log of session operations:

```
session list
• [UiAutomator2] Deleting UiAutomator2 session
• [UiAutomator2] Deleting UiAutomator2 server session
• [WD Proxy] Matched '/' to command name 'deleteSession'
• [WD Proxy] Proxying [DELETE /] to [DELETE
http://127.0.0.1:8200/wd/hub/session/ab6bf9b0-d12d-4dc1-bca7-99c3c0b3e695] with no
body
• [WD Proxy] Got response with status 200: {"sessionId":"ab6bf9b0-d12d-4dc1-bca7-
99c3c0b3e695","value":null}
• [ADB] Running 'C:\android-sdk\platform-tools\adb.exe -P 5037 -s 420012b2e0fa858b
shell am force-stop com.cogmento.app'
• [Instrumentation] .
• [Instrumentation] Time: 68.053
• [Instrumentation]
• [Instrumentation] OK (1 test)
• [Instrumentation] The process has exited with code 0
• [Logcat] Stopping logcat capture
• [ADB] Removing forwarded port socket connection: 8200
• [ADB] Running 'C:\android-sdk\platform-tools\adb.exe -P 5037 -s 420012b2e0fa858b
forward --remove tcp\:8200'
• [HTTP] --> GET /wd/hub/sessions
• [HTTP] {}
• [GENERIC] Calling AppiumDriver.getSessions() with args: []
• [GENERIC] Responding to client with driver.getSessions() result: []
• [HTTP] <-- GET /wd/hub/sessions 200 8 ms - 40
• [HTTP]
```

*Figure 13.15*

The Appium inspector opens, as shown below:



**Figure 13.16**

Click **Start** Appium inspector opens up. You can use Appium Inspector to capture Element Locators

Let's finally see the changes to be done for the framework.

## Changes for the framework

Let's see the changes that are required for the framework.

## Changes to the SingletonDriver class

Add the declarations shown below:

```
private ThreadLocal>mobileDriver =  
new ThreadLocal>();  
String localHub = "http://127.0.0.1:4723/wd/hub";
```

The setDriver method will have the case statements shown below:

```
case "iphone":  
case "ipad":  
if (browser.equalsIgnoreCase("ipad")) {  
caps = DesiredCapabilities.ipad();  
}
```

```
else {  
caps = DesiredCapabilities.iphone();  
}
```

```
mobileDriver.set(new IOSDriver(  
new URL(localHub), caps));
```

```
break;  
case "android":  
caps = DesiredCapabilities.android();
```

```
mobileDriver.set(new  
AndroidDriver(  
new URL(localHub), caps));  
  
break;  
}
```

Now when we have to access the WebDriver and MobileDriver together in a single test or two different browsers in a single test, use the code shown below

The first one is the overloaded setDriver for browsers:

```
public void setDriver(WebDriver driver) {  
webDriver.set(driver);  
sessionId.set(((RemoteWebDriver) webDriver.get())  
.getSessionId().toString());  
sessionBrowser.set(((RemoteWebDriver) webDriver.get())  
.getCapabilities().getBrowserName());  
sessionPlatform.set(((RemoteWebDriver) webDriver.get())  
.getCapabilities().getPlatform().toString());  
setBrowserHandle(getDriver().getWindowHandle());  
}
```

The second one is the overloaded setDriver for mobile:

```
public void setDriver(AppiumDriver driver) {  
mobileDriver.set(driver);  
sessionId.set(mobileDriver.get())
```

```
.getSessionId().toString());  
sessionBrowser.set(mobileDriver.get())  
.getCapabilities().getBrowserName());  
sessionPlatform.set(mobileDriver.get())  
.getCapabilities().getPlatform().toString());  
}
```

In order to get the current mobile driver instance, we write the code shown below:

```
public AppiumDriverfetchDriver(boolean mobile) {  
return mobileDriver.get();  
}
```

The code shown below will fetch the currently active webDriver or

```
/**  
* fetchCurrentDriver method will retrieve the active WebDriver  
* or AppiumDriver  
*  
* @return WebDriver  
*/  
public WebDriver fetchCurrentDriver() {  
if (getInstance().getSessionBrowser().contains("iphone") ||  
getInstance().getSessionBrowser().contains("ipad") ||  
getInstance().getSessionBrowser().contains("android") ) {  
  
return getInstance().getDriver(true);  
}
```

```
else {  
    return getInstance().getDriver();  
}  
}  
}
```

Next we set few IOS and Android preferences.

This is how we set IOS preferences:

```
switch(browser) {  
case "iphone": case "ipad":  
if (browser.equalsIgnoreCase("ipad") ) {  
cap = DesiredCapabilities.ipad();  
}  
  
else {  
cap = DesiredCapabilities.iphone();  
}  
  
cap.setCapability("appName",  
"https://myapp.com/myApp.zip");  
  
cap.setCapability("udid",  
"12345678"); // physical device  
  
cap.setCapability("device",  
"iPhone"); // or iPad
```

```
mobileDriver.set(new IOSDriver  
(new URL("http://127.0.0.1:4723/wd/hub"), caps));  
  
break;  
}
```

This is how we set Android preferences:

```
switch(browser) {  
case "android":  
cap = DesiredCapabilities.android();  
  
cap.setCapability("appName",  
"https://myapp.com/myApp.apk");  
  
cap.setCapability("udid",  
"12345678"); // physical device  
  
cap.setCapability("device", "Android");  
mobileDriver.set(new AndroidDriver  
(new URL("http://127.0.0.1:4723/wd/hub"), caps));  
break;  
}
```

We will next see the changes required for mobile for Selenium Grid:

```
// for each mobile device instance
```

```
if (environment.equalsIgnoreCase("remote") ) {  
    // setup the Selenium Grid capabilities...  
    String remoteHubURL = "http://localhost:4444/wd/hub";  
  
    cap.setCapability("browserName", browser);  
    cap.setCapability("platform", platform);  
  
    // unique user-specified name  
  
    cap.setCapability("applicationName",  
        platform + "-" + browser);  
  
    if (browser.contains("iphone") ) {  
        mobileDriver.set(new IOSDriver  
(new URL(remoteHubURL),  
            cap));  
    }  
  
    else {  
        mobileDriver.set(new AndroidDriver  
(new URL(remoteHubURL),  
            cap));  
    }  
}
```

That's all as far as the code for mobile automation goes. We have covered quite a lot about mobile automation in this chapter.

## Conclusion

This chapter covered several concepts related to Appium. We saw how mobile Automation works with regards to Appium. We saw what Appium Inspector and UIAutomator are and also understood the concept of Appium Server. Finally, we saw the changes related to Selenium Grid and integration with the framework.

The next chapter will be the final chapter, which will cover the important features in Selenium 4. So stay tuned.

## Questions

What is the use of AppiumDriver?

Take a sample website and identify elements using Appium Inspector

Try identifying elements with UIAutomator2

Try writing a simple program for freecrm website

List the steps in mobile automation

A Look at Selenium-4

We are in the final chapter in which we will learn about the nuances in Selenium 4. Selenium 4 is the next major release after Selenium 3 and brings to the plate, many new features apart from aligning with the W3C WebDriver specification (<https://w3c.github.io/webdriver/>). These features include Artificial Intelligence features that assist in visually validating a page as far as Visual features are concerned. We will look at some newly added features related to locating elements on a web page. Switching to a parent frame is no longer a tedious job. Switching to a parent frame can be accomplished with just a single command, which we will see in this chapter. There are enhancements related to taking screenshots, which we will explore.

This will be a fun chapter to read and digest. Once we get a stable version for Selenium-4, you will be able to migrate your framework to Selenium-4. There will be very little code in this chapter. This will be an informative chapter.

After reading this chapter, the reader will understand the new features in Selenium-4.

## Structure

Here's what we will learn in this chapter:

Changes to Screenshot functionality

Switching to the parent frame directly

Changes to Selenium grid

New Locators

AI with Selenium

Visual validation

Applitools Eyes

Self-correcting tests

Auto coder

Improved Docker support

## Objectives

Understanding the new features in Selenium-4

Learn the AI features introduced in Selenium-4

Learn the changes related to screenshots, frame switching etc

Learn the new locators in Selenium-4

We begin with an introduction to the changes brought in for the Screenshot functionality.

### [\*Changes related to taking screenshots\*](#)

In Selenium-3, we have the TakesScreenshot interface and getScreenshotAs() method to capture screenshots. But we only get the screenshot of the currently visible page as a whole. Selenium 4 lets you take the screenshot of a section of a page (Div or Form) or a single element. Let's see how to take the screenshot of a single

### Taking a screenshot of a single web element or a section

To take the screenshot of a single web element follow the steps below

Create a WebElement for the element for which a screenshot has to be captured

```
WebElement sampleButton = driver.findElement(By.xpath(Button  
xpath));
```

Invoke the getScreenshotAs() method on this web element as shown below

```
File srcFile = sampleButton.getScreenshotAs(OutputType.FILE);
```

Copy the screenshot to the desired location using the FileUtils class from the Apache CommonsIO package. Note the ./ which is used to go to the parent folder.

```
FileUtils.copyFile(srcFile,new File("./path of the screenshot  
folder/name of file.png"));
```

Follow steps 1 to 3 to take the screenshot of a section of a page, which can be a form or div. You need to put the locator of the div or form, and then a screenshot of that section will be taken



## **Switching to the parent frame directly**

Up until Selenium-3, if we have frames inside frames, then switching from the child frame to the parent frame was a two-step process as shown below:

Switch from the child frame to the main window using  
`driver.switchTo().defaultContent();`

Switch from the main window to the parent frame

Now with Selenium-4 in the picture, you need to give one command once you are in the child frame, as shown below:

`driver.switchTo().parentFrame();`

The command shown above will take you to the parent frame directly without the need to navigate to the main window.

## Selenium Grid-4

The new Selenium Server JAR contains everything that is needed to run a grid. It's a single jar that contains every dependency, so can be run by issuing the commands shown below

```
java -jar selenium-server-4.0.0-alpha-1.jar
```

Selenium Grid-4 can be run in three modes, as shown below:

Standalone mode

Hub and node

Fully distributed

## [Standalone mode](#)

The simplest mode to run the server in is the standalone mode.

Just run:

```
java -jar selenium-server-4.0.0-alpha-1.jar standalone
```

The server will listen on and that's the URL one should point remote WebDrivers to (that is, one no longer needs to use By default, the server will detect available drivers it can use (GeckoDriver, etc.) by looking at the PATH one can use this mode to make sure that things are working as expected.

## Hub and node

This is the traditional mode in which the Grid has been run as two pieces: the hub and one or more nodes. To start the hub, use the command below:

```
java -jar selenium-server-4.0.0-alpha-1.jar hub
```

And start the node with:

```
java -jar selenium-server-4.0.0-alpha-1.jar node --detect-drivers
```

If you wish to run the hub and the node on separate machines, start the node with:

```
java -jar selenium-server-4.0.0-alpha-1.jar node --detect-drivers --  
publish-events tcp://hub:4442 --subscribe-events tcp://hub:4443
```

Let's look at the final one, which is fully distributed.

## Fully distributed

One can start Grid 4 in a fully distributed manner, with each conceptual piece in its process. First of all, start the session map (which is responsible for mapping session IDs to the node the session is running on):

```
java -jar selenium-server-4.0.0-alpha-1.jar sessions
```

Then start the distributor:

```
java -jar selenium-server-4.0.0-alpha-1.jar distributor --sessions  
http://localhost:5556
```

And augment the system with a router:

```
java -jar selenium-server-4.0.0-alpha-1.jar router --sessions  
http://localhost:5556 --distributor http://localhost:5553
```

The router will now listen to new session requests on

Add a node to it:

```
java -jar selenium-server-4.0.0-alpha-1.jar node --detect-drivers
```

Finally, confirm everything by navigating to

Let's move on and see some of the new locators introduced in Selenium 4.

## New locators

We know about the existing locators in Selenium-3. Let's now look at what new locators have been introduced in Selenium 4

`toLeftOf()`: Returns the element located to the left of specified element.

`toRightOf()`: Returns the element located to the right of the specified element.

`above()`: Returns the element located above the specified element.

`below()`: Returns the element located below the specified element.

`near()`: Returns the element that is at most 50 pixels far away from the specified element. The pixel value can be modified.

## Using artificial intelligence with Selenium

Machine learning, which is the subset of artificial intelligence, has gained a lot of popularity nowadays. Machine learning is an area where machines are taught, and they learn using data samples to perform a particular task. A very famous example of machine learning is self-driving cars or driverless cars. While machine learning has become popular in other areas, it is finding its way into the world of test automation as well. Machines can be made to take over a few of the mundane tasks of testers, but we still require manual testing, which cannot be wiped off completely.

Machine learning can be used to help in the areas mentioned below:

Machine learning can examine the results of your tests and help to correct problems

Machine learning can be used to automatically correct errors with your tests

Machine learning that can write tests for you

Of course, the above areas have a lot to be worked on, but still, we have a few APIs that can help us with the usage of AI in test automation. Let's look at the first area in which AI can be used, which is visual validation.



## Visual validation

Automation code can be written to take screenshots and then make a pixel-by-pixel comparison to see if the two match. The problem is that they rarely do: variables encountered can be different machines which run at different resolutions, different rendering engines that could result in color changes that were invisible to the human eye, but different to a computer for eg. Attributes such as brightness and contrast levels. One has to accept the small percentage of error that happens

Using a percentage of a difference to work out if the correct thing was displayed on-screen turned out to be very difficult for visual testing, a percentage difference on its own is just not noticeable by a human. Imagine that you have two screenshots of a screen: your expected image and the actual image. Both images are 80% similar to each other; however, they are very different concerning a color change that cannot be seen with the human eye or showing a different button.

Next, we have a look at an API called AppliTools Eyes, which helps in validating the visual contents of a web page. This tool works with Selenium-3 as well.

## Exploring Applitools Eyes

Applitools Eyes can be set in the project using the entry shown below in

com.applitools  
eyes-common-java4  
4.3.0

The next thing to do is register on the AppliTools website  
<https://applitools.com/users/register>

Once the signup process gets completed, the API key will be made available. Using the sample code on their website, a visual test can be done. One can customize the code to add the script as per his/her requirement. Let's see a sample script where visual checkpoints are inserted before and after the click of a button.

```
public class TestClass5 {  
    WebDriver driver=null;  
    Eyes eyes = new Eyes();  
  
    @BeforeTest  
    public void setUp() {  
        try {  
            SingletonDriver.getInstance().setDriver("chrome", "local",  
                "windows");  
        } catch (Exception e) {  
            e.printStackTrace();  
        }  
    }  
}
```

```
} catch (Exception e) {  
    // TODO Auto-generated catch block  
    e.printStackTrace();  
  
}  
driver = SingletonDriver.getInstance().getDriver();  
}
```

```
@Test  
public void test1() {  
    eyes.setApiKey("Your API Key");  
    eyes.open(driver, "HelloWorld", "My first Eyes code");  
    driver.get("https://applitools.com/helloworld");  
    eyes.checkWindow("Hello!");  
    driver.findElement(By.tagName("button")).click();  
    eyes.checkWindow("Click!");  
    eyes.close();  
}
```

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

## OUTPUT:

The first time the test is run, a base image gets generated. After the first run, all subsequent runs will be compared with the base image. The Applitools dashboard is shown below

The screenshot shows the appliTools Eyes Test results interface. At the top, there's a header with the appliTools logo and a dropdown menu labeled "Test results". Below the header, a sidebar on the left lists "Last 1 batch runs" with one entry: "My first Eyes code" from Jan 18 at 5:24 PM, status "Passed". The main area is titled "Test results of batch: My first Eyes code" and shows a summary: Started: 18 Jan 2020 at 5:24 PM, Duration: 00:00:21, 1 test, 1 new test, 1 passed test. A "Passed" button is highlighted. Below this, there are sections for "SELECTED TESTS", "VIEW", "TEST RESULTS", and "AUTO MAINTENANCE". The "TEST RESULTS" section contains a table with columns for Status, Test, OS, Browser, Viewport, Device, Started, and Actions. It shows two rows: "My first Eyes code" (Passed) and "Windows 7 | Chrome 79.0 | 1034x572 | 18 Jan 2020 at 5:24 PM". Under the "TEST RESULTS" table, there are two screenshots labeled "HELLO WORLD". The first screenshot has a green checkmark icon and the text "1/2 Hello!". The second screenshot has a green checkmark icon and the text "2/2 Click!".

**Figure 14.1**

Now run the test again and notice the output.

This screenshot is similar to Figure 14.1, showing the appliTools Eyes Test results interface. The sidebar shows "Last 2 batch runs" with entries for "My first Eyes code" from Jan 18 at 5:31 PM and "My first Eyes code" from Jan 18 at 5:24 PM, both marked as "Passed". The main area shows the same test results summary and table. In the "TEST RESULTS" table, the first row ("My first Eyes code") has a black square highlight over its "Status" column. The screenshots below show a visual comparison between the current run and a previous one. The first screenshot has a black square highlight over the "Hello!" text, indicating a difference. The second screenshot has a green checkmark icon and the text "2/2 Click!".

**Figure 14.2**

Notice the highlighted section has an equal to sign which means that the visual screen of the current run are matching with those

of the Base run

Now change the URL to <https://applitools.com/helloworld?diff2> and run the test again. This time the test fails. If you navigate to the Applitools dashboard, the changes are highlighted for you, and thus the two screen versions are visually compared.

The screenshot shows the Applitools Eyes dashboard with the following details:

- Test results of batch: My first Eyes code**
  - Started: 18 Jan 2020 at 5:40 PM | Duration: 00:00:10 | 1 test | 1 unresolved test
  - Unresolved: Resolved diffs 0% - 2 unresolved steps
- Selected Tests**: Shows three entries:
  - My first Eyes code (18 Jan 2020 at 5:40 PM) - Unresolved
  - My first Eyes code (18 Jan 2020 at 5:31 PM) - Passed
  - My first Eyes code (18 Jan 2020 at 5:24 PM) - Passed
- View Options**: Includes buttons for Selected tests, View, Test results, and Auto maintenance.
- Group by**: Options for Status, Test, OS, Browser, Viewport, Device, and Started.
- Test Results Grid**: Displays two rows of visual comparisons for the 'Hello' and 'Click' steps. Each row has two columns: 'Actual' and 'Expected'. The first row shows a 'Hello' step with a pink box labeled 'Hello' and a 'Click' step with a pink box labeled 'Click'. The second row shows a 'Hello' step with a pink box labeled 'Hello' and a 'Click' step with a pink box labeled 'Click'.

**Figure 14.3**

Next, let's see what self-correcting tests are.

## [Self-correcting tests](#)

One of the main areas of interest right now for artificial intelligence in the area of automation is the idea of self-healing tests. This is where artificial intelligence can learn whether a change to the website in question that has caused your test to fail is something that is expected. If it is an expected change, artificial intelligence automatically modifies your code to fix the problem.

One of the websites which use Selenium is Testcraft:

This tool generally works in two ways. On the one hand, they crawl your website to try and work out what pages are available and thus create an image of the website. On the other hand, they use to record and play for training purposes. At the time of writing, there is no way to pass your Selenium code over to these systems, which is kind of understandable.

Once these tools are trained, they will regularly check against your website and verify that everything still looks ok. If the tool detects changes, it can then either flag the changes as errors so that they can be fixed, or it can use its self-correcting algorithms to modify the tests so that they won't fail in the future.

## Auto coders

Sometime in the future, we might have tools that generate Selenium code for us using Artificial Intelligence. The tester needs to point the tool at the website you wish, and Selenium code will get generated for you. If this tool becomes a reality, then manual testing will get reduced to only validating certain functional aspects.

The automation tool will have to be trained by providing constant input so that it has a large amount of data to work with, which is required in a machine learning kind of scenario.

Auto coding robots will become a reality, but it will take time since the machine learning field is still evolving as far as Selenium goes.

### *Improved Docker support*

Last but not least, Selenium 4 comes with improved docker support, which makes it easier to work with Selenium-Grid 4. The official documentation has still not been released, but it is going to be a promising journey ahead with Selenium.

## Next steps

With this, we come to the end of this chapter and this book. I hope you enjoyed reading this book. I would encourage the readers of this book to practice Selenium WebDriver using examples in this book. In this book, I have used the MySQL database, but you can experiment with Microsoft Excel or a CSV file. The disadvantage of an Excel file is that it becomes difficult to use it on non-Windows platforms. You can also experiment with a JSON file for the test cases and test data. JSON works on non-Windows platforms as well.

## Conclusion

This chapter covered the much-hyped area of AI to be used with Selenium 4. We saw a few new locators in Selenium 4. We also saw the Grid changes for Grid 4. We had a look at a visual validation tool called Appli-Tools Eyes and also saw a sample program for validating a web page.

## Questions

Write a program to take the screenshot of a button

Write a program to take the screenshot of a div containing some web elements

Try changing the current code for Selenium Grid concerning Grid 4

Try writing a simple program for Applitools Eyes

Try exploring websites that cover Machine Learning for Selenium