Q1) You are given a project to implement @Test and other related annotations.

Step 1.1.1: Creating a simple Java project

- Open Eclipse.
- Go to the **File** menu and select **New->Java Project.**
- Enter the project name as **Annotations**. Click on **Next.**
- This will create the project files in the Project Explorer.

Step 1.1.2: Installing TestNG

- TestNG is already installed in your Practice labs. To learn about its directory details you can refer to the lab guide for Phase 2.

Step 1.1.3: Adding TestNG libraries to the Class Path

- In the **Project Explorer**, right-click on **Annotations.**
- Select **Properties**. Select **Java Build Path** from the list. Go to **Libraries.**
- Click on **Add Library.** Select **TestNG**. Click on **Next**. Now, click on **Finish.**
- Finally, click on **Apply and Close.**

Step 1.1.4: Creating a class file named TestAnnotations

- In the Project Explorer, expand **Annotations->Java Resources.**
- Right-click on **src** and choose **New->Class.**
- In **Class Name,** enter **TestAnnotations**. In **Package Name,** enter **com.testannotations** and click on **Finish.**
- Enter the following code:

```
package com.testannotations;
```

```
import org.testng.annotations.*;

public class TestAnnotations {

	@Test
	public void Test1() {
		System.out.println("Test1 Executed");
	}
	@Test
	public void Test2() {
		System.out.println("Test2 Executed");
	}

	@BeforeTest
	public void beforeTest() {
		System.out.println("BeforeTest Executed");
	}
	@AfterTest
	public void AfterTest() {
		System.out.println("AfterTest Executed");
	}

	@BeforeMethod
	public void beforeMethod() {
		System.out.println("BeforeMethod Executed");
	}
	@AfterMethod
	public void afterMethod() {
		System.out.println("AfterMethod Executed");
	}

	@BeforeClass
	public void beforeClass() {
		System.out.println("BeforeClass Executed");
	}
	@AfterClass
	public void afterClass() {
		System.out.println("AfterClass Executed");
	}

}
```

Step 1.1.5: Running the project as TestNG

- Right-click on **TestAnnotations** class. Click on **TestNG->Convert to TestNG.**
- Click on **Finish.** It will create a **TestNG.xml** file. Open that file.
- Right click on the screen. Select **Run As ->TestNG Suite.**

Q2) You are given a project to work with groups attribute of @Test and perform parallel execution.

Step 1.1.1: Creating a simple Java project

- Open Eclipse.
- Go to the **File** menu and select **New->Java Project.**
- Enter the project name as **Annotations**. Click on **Next.**
- This will create the project files in the Project Explorer.

Step 1.1.2: Installing TestNG

- TestNG is already installed in your Practice labs. To learn about its directory details you can refer to the lab guide for Phase 2.

Step 1.1.3: Adding TestNG libraries to the Class Path

- In the **Project Explorer**, right-click on **Annotations.**
- Select **Properties**. Select **Java Build Path** from the list. Go to **Libraries.**
- Click on **Add Library.** Select **TestNG**. Click on **Next**. Now, click on **Finish.**
- Finally, click on **Apply and Close.**

Step 1.1.4: Creating a class file named TestAnnotations

- In the Project Explorer, expand **Annotations->Java Resources.**
- Right-click on **src** and choose **New->Class.**
- In **Class Name,** enter **TestAnnotations**. In **Package Name,** enter **com.testannotations** and click on **Finish.**
- Enter the following code:

```
package com.testannotations;
```

```
import org.testng.annotations.*;

public class TestAnnotations {

	@Test
	public void Test1() {
		System.out.println("Test1 Executed");
	}
	@Test
	public void Test2() {
		System.out.println("Test2 Executed");
	}

	@BeforeTest
	public void beforeTest() {
		System.out.println("BeforeTest Executed");
	}
	@AfterTest
	public void AfterTest() {
		System.out.println("AfterTest Executed");
	}

	@BeforeMethod
	public void beforeMethod() {
		System.out.println("BeforeMethod Executed");
	}
	@AfterMethod
	public void afterMethod() {
		System.out.println("AfterMethod Executed");
	}

	@BeforeClass
	public void beforeClass() {
		System.out.println("BeforeClass Executed");
	}
	@AfterClass
	public void afterClass() {
		System.out.println("AfterClass Executed");
	}

}
```

Step 1.1.5: Running the project as TestNG

- Right-click on **TestAnnotations** class. Click on **TestNG->Convert to TestNG.**
- Click on **Finish.** It will create a **TestNG.xml** file. Open that file.
- Right click on the screen. Select **Run As ->TestNG Suite.**

Step 1.2.1: Creating a simple Java project

- Open Eclipse.
- Go to the **File** menu and select **New->Java Project.**
- Enter the project name as **Parallel Tests**. Click on **Next.**
- This will create the project files in the Project Explorer.

Step 1.2.2: Downloading Selenium WebDriver jar, chromdriver.exe, and firefoxdriver.exe

- Selenium WebDriver is already installed in your Practice lab. Refer QA to QE lab guide for Phase 2 for more information.
- Go to https://selenium.dev/thirdparty/
- To download chromedriver.exe, click on Google ChromeDriver and select the appropriate version as per your Chrome version.
- To download firefoxdriver.exe, click on Microsoft GeckoDriver

Step 1.2.3: Adding the Web Driver dependency in the project

- In the Project Explorer, right-click on **Parallel Tests.**
- Select **Properties**. Select **Java Build Path** from the list. Go to **Libraries**.
- Click on **Add External JARs** and browse the location where you have downloaded the JAR files.
- Select JARs from the **root** folder and the **libs** folder.
- Click on **Apply and Close.**
- Copy the **chromedriver.exe** and **geckodriver.exe**, and paste it your project creating a resource folder.

Step 1.2.4: Installing TestNG

- TestNG is already installed in your Practice labs. To learn about its directory details you can refer to the lab guide for Phase 2.

Step 1.2.5: Adding TestNG libraries to the Class Path

- In the Project Explorer, right-click on **Parallel Tests.**
- Select **Properties**. Select **Java Build Path** from the list. Go to **Libraries.**
- Click on **Add Library.** Select **TestNG**. Click on **Next**. Now, click on **Finish.**
- Click on **Apply and Close.**

Step 1.2.6: Creating a Java class named ParallelTest.java

- In the Project Explorer, expand **Parallel Tests->Java Resources.**
- Right-click on **src** and select **New->Class.**
- In **Class Name,** enter **ParallelTests** and click on **Finish.** In **Package Name,** enter **com.parallel** and click on **Finish.**
- Enter the following code:

```
package com.parallel;

import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.chrome.ChromeDriver;
import org.openqa.selenium.firefox.FirefoxDriver;
import org.testng.annotations.Test;

public class ParallelTests {

	WebDriver driver;
	@Test(groups="Chrome")
	public void LaunchChrome() {
		System.setProperty("webdriver.chrome.driver", "./Resources/chromedriver.exe");
		driver = new ChromeDriver();
		driver.get("https://www.facebook.com");
		try {
			Thread.sleep(2000);
		} catch (Exception e) {
			e.printStackTrace();
		}
	}
	@Test(groups="Chrome", dependsOnMethods="LaunchChrome")
	public void TryFacebook1() {
		System.out.println(Thread.currentThread().getId());
		driver.findElement(By.id("email")).sendKeys("ravi10thstudent@gmail.com");
		driver.findElement(By.id("pass")).sendKeys("12345");
		driver.findElement(By.id("loginbutton")).click();
	}

	@Test(groups="Firefox")
	public void LaunchFirefox() {
		System.setProperty("webdriver.gecko.driver", "./Resources/geckodriver.exe");
		driver = new FirefoxDriver();
		driver.get("https://www.facebook.com");
		try {
```

```
                Thread.sleep(4000);
            } catch (Exception e) {
                e.printStackTrace();
            }

        }

        @Test(groups="Firefox", dependsOnMethods="LaunchFirefox")
        public void TryFacebook2() {
            System.out.println(Thread.currentThread().getId());
            driver.findElement(By.id("email")).sendKeys("ravi10thstudent@gmail.com");
            driver.findElement(By.id("pass")).sendKeys("ravi28394");
            driver.findElement(By.id("loginbutton")).click();
            System.out.println(Thread.currentThread().getId());
        }
}
```

Step 1.2.7 Running the project

- Right-click on **ParallelTests** class. Click on **TestNG->Convert to TestNG.**
- Click on **Finish.** It will create a **TestNG.xml** file. Open that file.
- Right click on the screen. Select **Run As ->TestNG Suite.**

Q3) You are given a project to implement soft and hard assertions on your test cases.

Step 1.3.1: Creating a simple Java project

- Open Eclipse.
- Go to the **File** menu. Select **New->Java Project.**
- Enter the project name as **Test Assertions**. Click on **Next.**
- This will create the project files in the Project Explorer.

Step 1.3.2: Downloading Selenium WebDriver jar, chromdriver.exe, and forefoxdriver.exe

- Selenium WebDriver is already installed in your Practice lab. Refer QA to QE lab guide for Phase 2 for more information.
- Go to https://selenium.dev/thirdparty/

- To download chromedriver.exe, click on Google ChromeDriver and select the appropriate version as per your Chrome version.
- To download firefoxdriver.exe, click on Microsoft GeckoDriver.

Step 1.3.3: Adding the WebDriver dependency in the project

- In the Project Explorer, right-click on **Test Assertions.**
- Select **Properties**. Select **Java Build Path** from the list. Go to **Libraries.**
- Click on **Add External JARs** and browse the location where you have downloaded the JAR files.
- Select JARs from the **root** folder and the **libs** folder.
- Click on **Apply and Close.**
- Copy the chromedriver.exe and geckodriver.exe, and paste it to your project creating a resource folder.

Step 1.3.4: Installing TestNG

- TestNG is already installed in your Practice labs. To learn about its directory details you can refer to the lab guide for Phase 2.

Step 1.3.5: Adding TestNG libraries to the Class Path

- In the Project Explorer, right-click on **Test Assertions.**
- Select **Properties**. Select **Java Build Path** from the list. Go to **Libraries.**
- Click on **Add Library.** Select **TestNG**. Click on **Next**. Click on **Finish.**
- Click on **Apply and Close.**

Step 1.3.6: Creating a Java class named ParallelTest.java

- In the Project Explorer, expand **Test Assertions->Java Resources**
- Right-click on **src** and select **New->Class**

- In **Class Name,** enter **Assertions** and click on **Finish.** In **Package Name,** enter **com.assert** and click on **Finish.**
- Enter the following code:

```
package com.asserts;

import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.chrome.ChromeDriver;
import org.testng.Assert;
import org.testng.annotations.Test;
import org.testng.asserts.SoftAssert;

public class Assertions {

	SoftAssert soft = new SoftAssert();
	WebDriver driver;
	@Test
	public void Launch() {
		System.setProperty("webdriver.chrome.driver", "./Resources/chromedriver.exe");
		driver = new ChromeDriver();
		try {
			Thread.sleep(3000);
		} catch (Exception e) {
			e.printStackTrace();
		}
	}

	@Test(dependsOnMethods = { "Launch" })
	public void Facebook() {
		driver.get("https://www.facebook.com");
		soft.assertEquals("FB Title", driver.getTitle());
		try {
			Thread.sleep(2000);
		} catch (Exception e) {
			e.printStackTrace();
		}
	}

	@Test(dependsOnMethods = { "Facebook" })
	public void Login() {
		driver.findElement(By.id("email")).sendKeys("ravi10thstudent@gmail.com");
		driver.findElement(By.id("pass")).sendKeys("12345");
		driver.findElement(By.id("loginbutton")).click();
		soft.assertAll();

		try {
			Thread.sleep(3000);
		} catch (Exception e) {
			e.printStackTrace();
		}

	}
}
```

Step 1.3.7: Running the project

- Right-click on **Assertions** class. Click on **TestNG->Convert to TestNG.**
- Click on **Finish.** It will create a **TestNG.xml** file. Open that file.
- Right-click on the screen. Select **Run As ->TestNG Suite.**

Q4) Demonstrate how extent reports are generated.

Step 1.4.1: Generating Extent Reports

- Open Eclipse.

- Create a TestNG project in Eclipse.
- Extent Reports jar file is already present in your practice lab in /home/ubuntu/libs directory. Refer QA to QE lab guide for Phase 2 for more information.
- Add the Extent Reports jar file to your project.
- Create a java class, say **ExtentReportsClass** and add the following code to it.
- Sample code:

```
import java.io.File;

 import org.testng.Assert;

import org.testng.ITestResult;

import org.testng.SkipException;

import org.testng.annotations.AfterMethod;

import org.testng.annotations.AfterTest;

import org.testng.annotations.BeforeTest;

import org.testng.annotations.Test;



import com.relevantcodes.extentreports.ExtentReports;
```

```
import com.relevantcodes.extentreports.ExtentTest;

import com.relevantcodes.extentreports.LogStatus;



public class ExtentReportsClass{

 ExtentReports extent;

 ExtentTest logger;

 @BeforeTest

 public void startReport(){

 //ExtentReports(String filePath,Boolean replaceExisting)

 //filepath - path of the file, in .htm or .html format - path where your report needs to generate.

//replaceExisting - Setting to overwrite (TRUE) the existing file or append to it

 //True (default): the file will be replaced with brand new markup, and all existing data will be lost. Use this option to create a brand new report

 //False: existing data will remain, new tests will be appended to the existing report. If the supplied path does not exist, a new file will be created.

 extent = new ExtentReports (System.getProperty("user.dir") +"/test-output/STMExtentReport.html", true);

 //extent.addSystemInfo("Environment","Environment Name")

 extent

 .addSystemInfo("Host Name", "SoftwareTesting")

 .addSystemInfo("Environment", "Automation Testing")
```

```
        .addSystemInfo("User Name", "TestEngineer");
        //loading the external xml file (i.e., extent-config.xml) that was placed under the base directory
        //You could find the xml file below. Create xml file in your project and copy paste the code mentioned below
        extent.loadConfig(new File(System.getProperty("user.dir")+"\\extent-config.xml"));
}
@Test
public void passTest(){
//extent.startTest("TestCaseName", "Description")
//TestCaseName – Name of the test
//Description – Description of the test
//Starting test
logger = extent.startTest("passTest");
Assert.assertTrue(true);
//To generate the log when the test case is passed
logger.log(LogStatus.PASS, "Test Case Passed is passTest"); }
@Test
public void failTest(){
logger = extent.startTest("failTest");
Assert.assertTrue(false);
```

```
logger.log(LogStatus.PASS, "Test Case (failTest) Status is passed");

}



@Test

public void skipTest(){

logger = extent.startTest("skipTest");

throw new SkipException("Skipping - This is not ready for testing ");

}

@AfterMethod

public void getResult(ITestResult result){

if(result.getStatus() == ITestResult.FAILURE){

logger.log(LogStatus.FAIL, "Test Case Failed is "+result.getName());

logger.log(LogStatus.FAIL, "Test Case Failed is "+result.getThrowable());

}else if(result.getStatus() == ITestResult.SKIP){

logger.log(LogStatus.SKIP, "Test Case Skipped is "+result.getName());

}

// ending test

//endTest(logger) : It ends the current test and prepares to create a HTML report

extent.endTest(logger);

}
```

```
@AfterTest
public void endReport(){
//writing everything to document
//flush() - to write or update test information to your report.
        extent.flush();
        //Call close() at the very end of your session to clear all resources.
        //If any of your tests ended abruptly causing any side-affects (not all logs sent to ExtentReports, information missing), this method will ensure that the test is still appended to the report with a warning message.
        //You should call close() only once, at the very end (in @AfterSuite for example) as it closes the underlying stream.
        //Once this method is called, calling any Extent method will throw an error.
        //close() - To close all the operations
        extent.close();
    }
}
```

- Code explanation:

i. Imported two classes *ExtentReports* and *ExtentTest*.

ExtentReports: By using this class, we set the path where our reports need to be generated.

ExtentTest: By using this class, we could generate the logs in the report.

ii. Took three methods with @Test annotation such as *passTest*, *failTest* and *skipTest* and a method *startTest* with @BeforeTest annotation and another method *endReport* with @AfterMethod annotation

iii. The used object of ***ExtentReports*** class (i.e., *extent*) in the *startReport* method which was assigned to @BeforeTest annotation to generate the HTML report in the required path.

iv. The used object of ***ExtentTest*** class (i.e., *logger*) in the remaining methods to write logs in the report.

v. Used ***ITestResult*** class in the @AfterMethod to describes the result of a test

Step 1.4.2: Describing Extent-config.xml

By using this external XML file (extent-config.xml), we could change the details, such as Report Theme, Report Title, and Document Title.

We use the extent object and use loadConfig() method to load this XML file.

```
<?xml version="1.0" encoding="UTF-8"?>

<extentreports>

  <configuration>

    <!-- report theme -->

    <!-- standard, dark -->

    <theme>standard</theme>

    <!-- document encoding -->

    <!-- defaults to UTF-8 -->
```

```
<encoding>UTF-8</encoding>

<!-- protocol for script and stylesheets -->

<!-- defaults to https -->

<protocol>https</protocol>

  <!-- title of the document -->

<documentTitle>ExtentReports 2.0</documentTitle>

 <!-- report name - displayed at top-nav -->

<reportName></reportName>

<!-- report headline - displayed at top-nav, after reportHeadline -->

<reportHeadline>Automation Report</reportHeadline>

<!-- global date format override -->

<!-- defaults to yyyy-MM-dd -->

<dateFormat>yyyy-MM-dd</dateFormat>

<!-- global time format override -->

<!-- defaults to HH:mm:ss -->

<timeFormat>HH:mm:ss</timeFormat>

<!-- custom javascript -->

<scripts>

  <![CDATA[

    $(document).ready(function() {
```

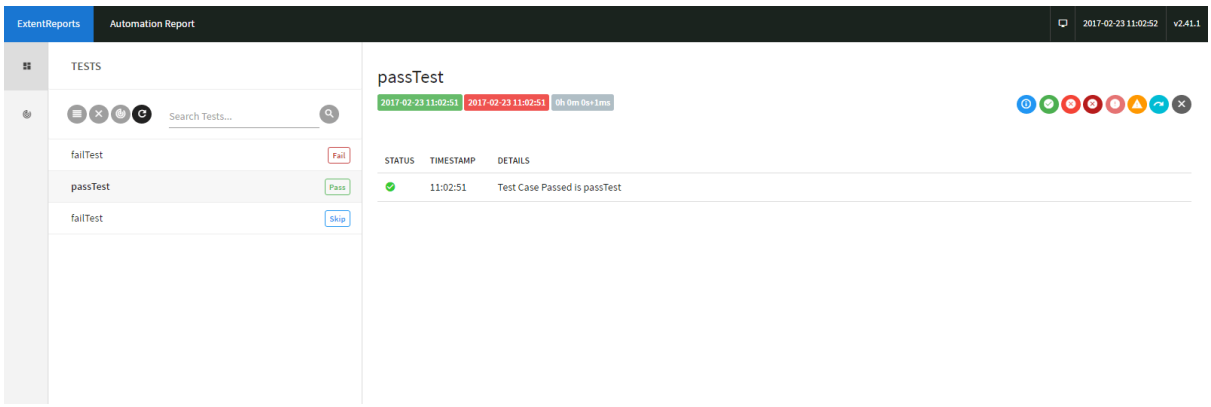
```
    });

    ]]>

  </scripts>

  <!-- custom styles -->

  <styles>

    <![CDATA[

      ]]>

  </styles>

 </configuration>

</extentreports>
```

- Console Output:

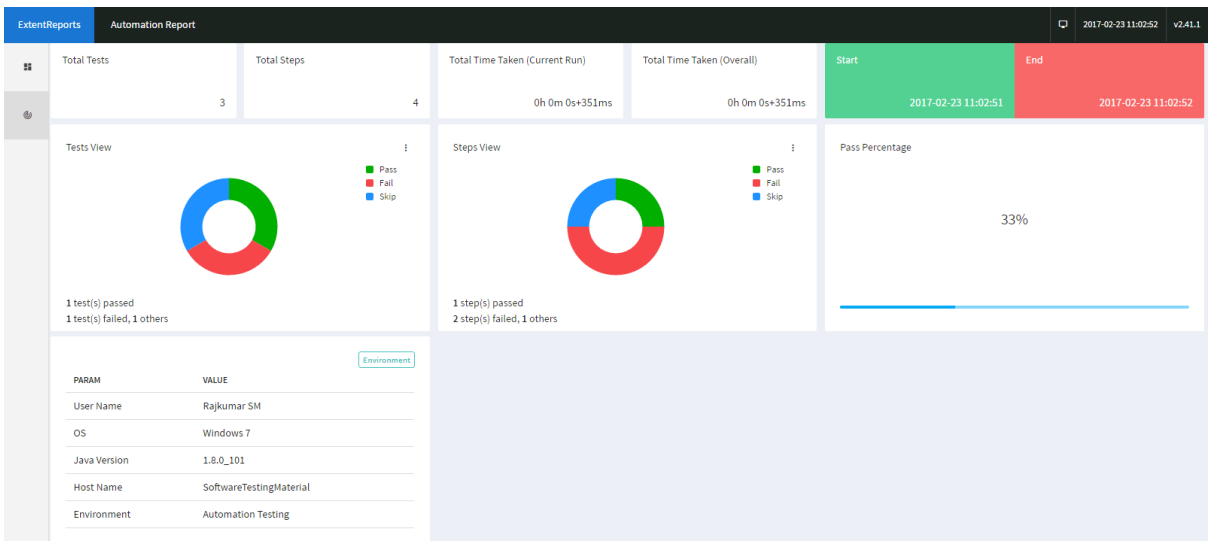
```
1 ===============================================

2 Default suite

3 Total tests run: 3, Failures: 1, Skips: 1

4 ===============================================
```

Refresh the project after execution of the above ExtentReportsClass.java file. You can find an HTML file named **STMExtentReport.html** in your test-output folder. Copy the location of the STMExtentReport.html file and open it by using any browser. Once you open the report, you will see rich HTML test results as shown below.

Step 1.4.3: Fetching Test Summary Report



Graphical Report with PIE Charts:



Q5) Demonstrate how test reports are exported to Excel.

Step 1.5.1 Creating a project with test cases with multiple annotations

- Open the Eclipse and create a Java project.
- Create multiple test case classes(Say Test_01, Test02).
- Create a Base class to extend the test cases.

Step 1.5.2 Adding AT Excel report jars

- Extent Reports jar file is already present in your practice lab in /home/ubuntu/libs directory.
- Add the Extent Reports jar file to your project: Right-click on project->Build path->Configure build path->Add external Jars.
- Click on Apply and then click OK.

Step 1.5.3 Executing the test suites to see the generated report in Excel sheet

- Write the test script in the Test_01 class.

```
package Testcases;
import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.chrome.ChromeDriver;
import org.testng.annotations.AfterMethod;
import org.testng.annotations.BeforeMethod;
import org.testng.annotations.Test;

import base.Baseclass;

public class TEST_01 extends Baseclass {

	@Test
	public void t_001()
	{
		driver.findElement(By.xpath("(//*[contains(text(),'Categories')])[1]")).click();
		driver.findElement(By.xpath("//*[contains(text(),'Central')]")).click();
		System.out.println("Test_01 executed successfully");
	}


}
```

- Write the test script in the Test_02 class.

```
package Testcases;

import org.openqa.selenium.By;
import org.testng.annotations.Test;

import base.Baseclass;

public class TEST_02 extends Baseclass {
	@Test
	public void t_002() {

			driver.findElement(By.xpath("(//*[contains(text(),'Popular')])[1]")).click();
```

```
                System.out.println("Test_02 executed successfully");
        }

}
```

- Write the test script for the extended Base class, where all annotations are declared here.

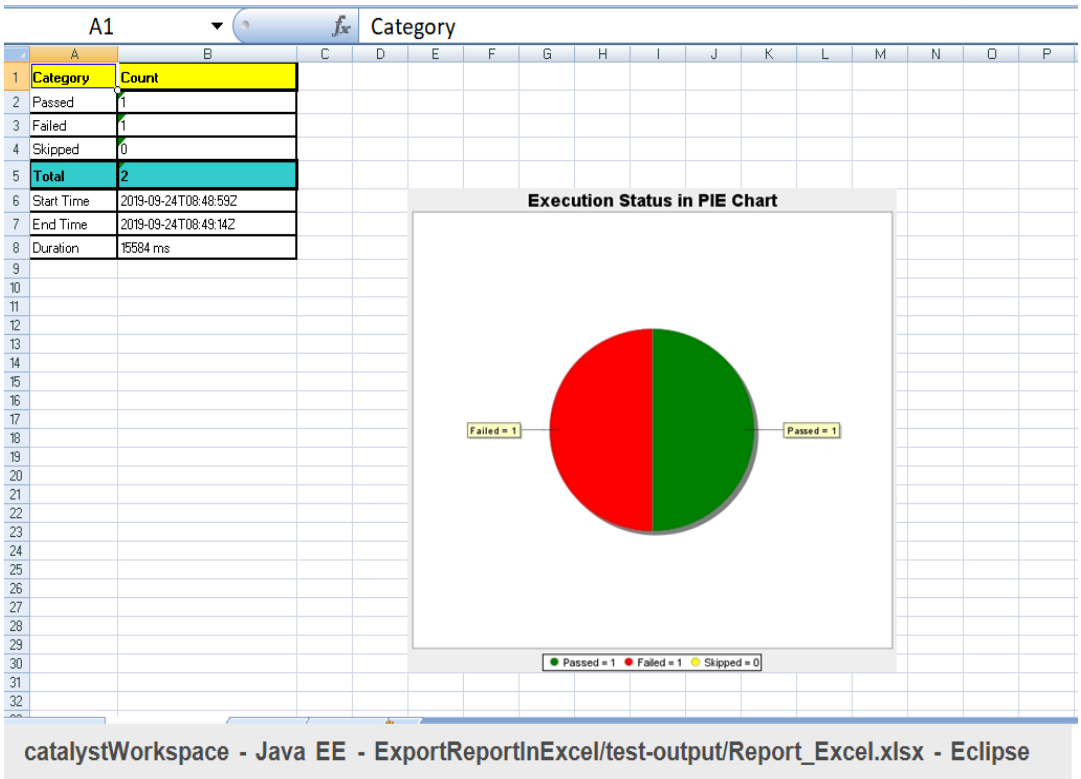
```
package base;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.chrome.ChromeDriver;
import org.testng.annotations.AfterMethod;
import org.testng.annotations.AfterSuite;
import org.testng.annotations.BeforeMethod;
import org.automationtesting.excelreport.Xl;

public class Baseclass {

        public WebDriver driver;

        @BeforeMethod
        public void baseclass1()
        {
                System.setProperty("webdriver.gecko.driver",
"/home/ubuntu/Downloads/gechodriver");
            driver = new geckodriver();
            driver.get("https://mvnrepository.com/");
        }

        @AfterMethod
        public void quitDriver() {
                driver.close();
        }

        @AfterSuite
        public void generateReport() throws Exception {
                Xl.generateReport("Report_Excel.xlsx");
        }

}
```

- Execute the test suite with multiple test cases and the testng.xml file will look like :

```
<!DOCTYPE suite SYSTEM "http://testng.org/testng-1.0.dtd">
<suite name="ExportReport">


        <test name="TEST1">

                <classes>
```

```
                        <class name="Testcases.TEST_01"></class>
                    </classes>
</test>

 <test name="TEST2">

                    <classes>
                        <class name="Testcases.TEST_02"></class>
                    </classes>
            </test>
</suite>
```

Finally, the executed script can generate the report in Excel and the graph will look like :



Q6) Demonstrate how test reports are published on Tomcat.

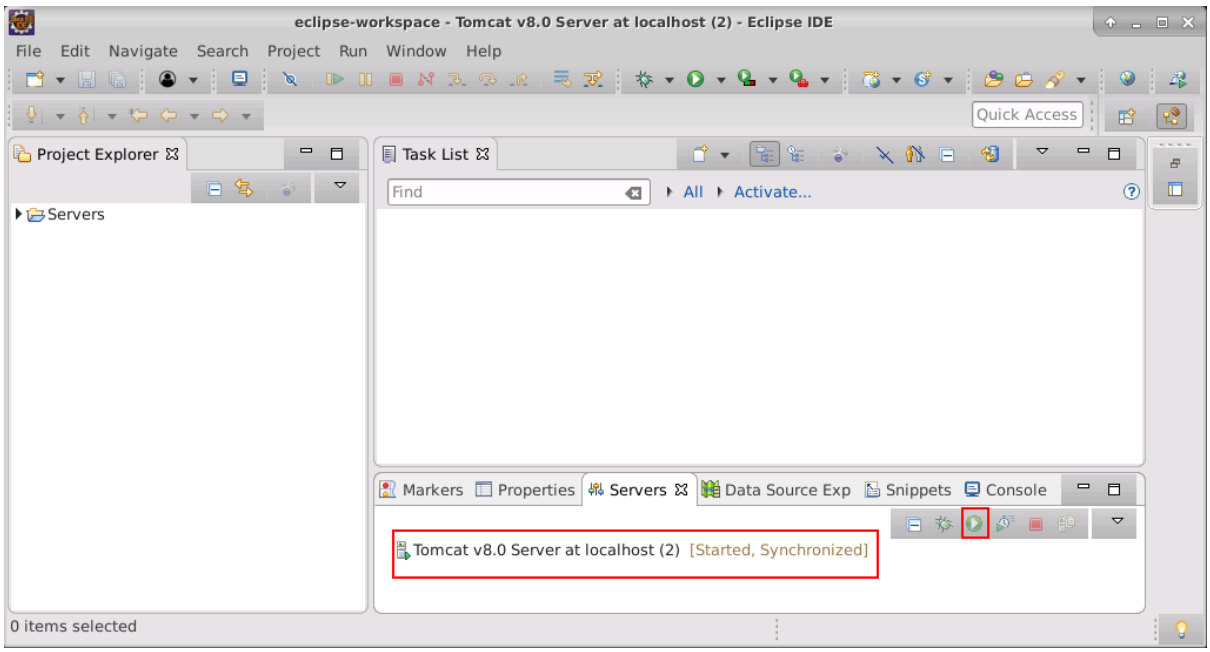
Step 1.6.1: Starting Tomcat server

- Download Tomcat within Eclipse using the following steps:
    - Open your eclipse environment from the desktop.
    - Select File tab, click on New>Other.
    - In the next page, click on Server from the server folder drop down. Click on Next.
    - Type tomcat in the filter text field. Select the tomcat version that you

would like to install and click on Next.

- Click on Download and Install. Select the directory where you want to download tomcat. Select the installed JRE and click Finish.

- Start the tomcat server in Eclipse

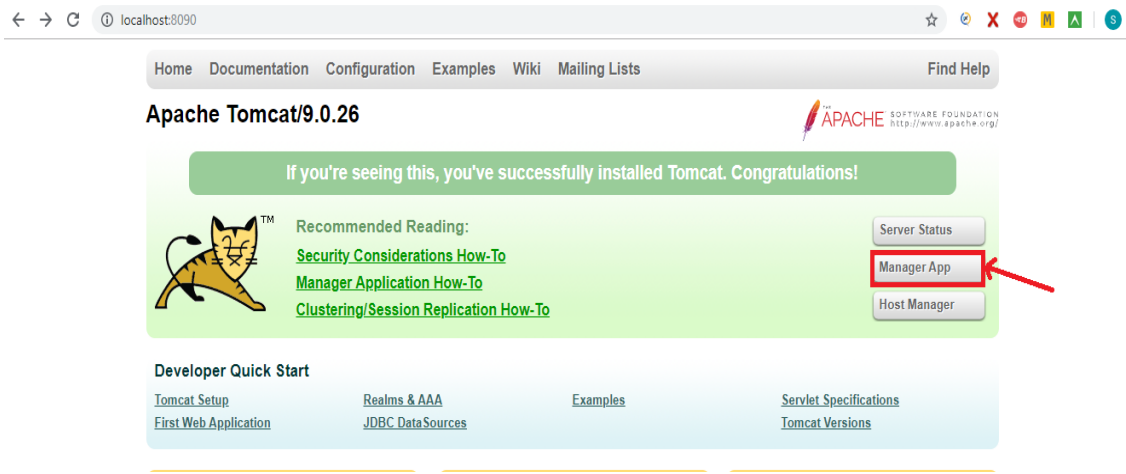


Step 1.6.2: Creating WAR file from Eclipse.

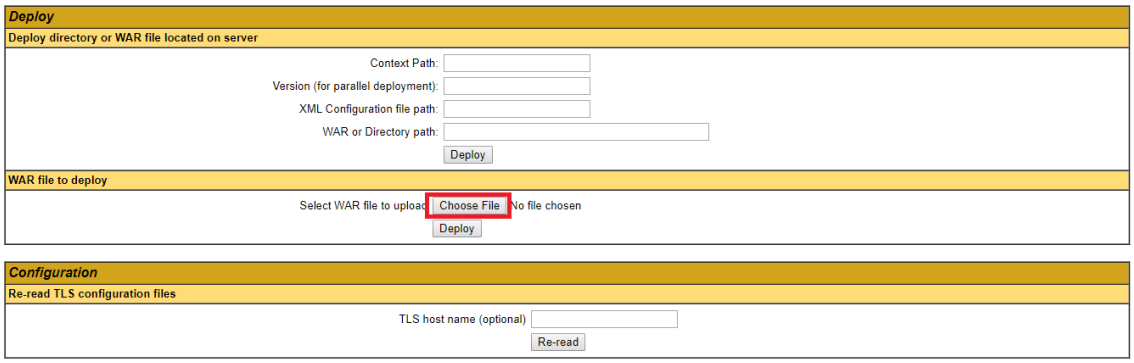
- Open Eclipse.
- Right-click on Project.
- Click on Export.
- Click on the Web in the Export window.
- Select WAR file and click on Next button.
- Enter the destination where the WAR file has to be saved.
- Check the ‘Export source file’ and ‘Overwrite Existing file’ checkboxes.

Step 1.6.3: Publishing Report on Tomcat

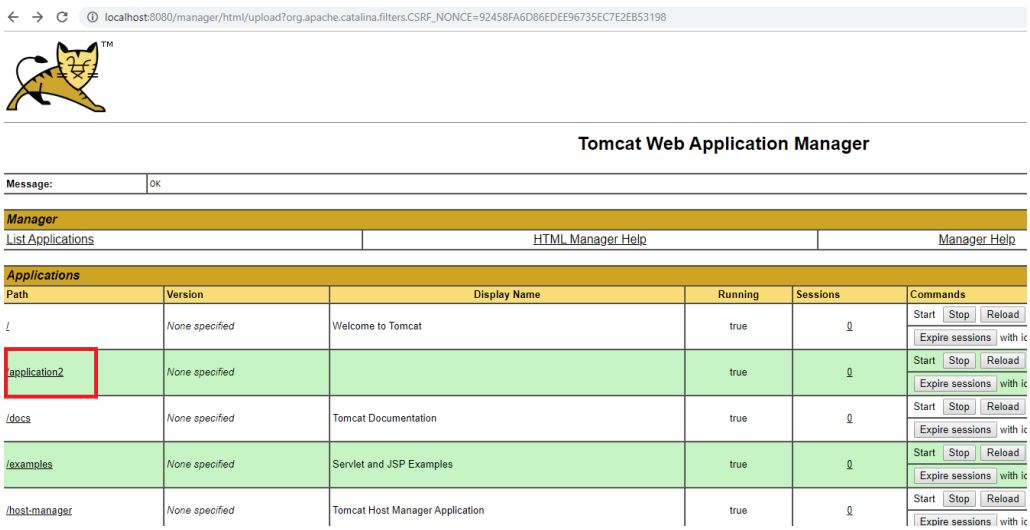
- Open the browser.
- Enter the url: https://localhost:8090 and click on Enter.
- Click on the Manager App button.



- Move to 'WAR file to deploy' block and click on "Choose" file.

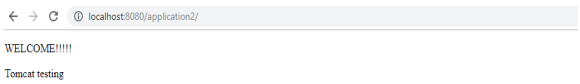


- Select the WAR file which we have created from Eclipse.
- Click on the 'Deploy' button.
- The filename will be displayed in applications list in Tomcat home page and will look like



- Click on the file name which we have deployed.

- The report is displayed in the Tomcat(output), which will look like



Q7) Demonstrate TestNG XML Parser.

Step 1.7.1: Explaining types of XML parsers

There are mainly three types of XML parsers:

1.7.1.1 SAX

1.7.1.2 DOM

1.7.1.3 Pull parser

Step 1.7.1.1: SAX

SAX stands for 'Simple API for XML'. It does not create any internal structure. Clients do not know what methods to call. They just override the methods of the API and place his own code inside the method. It is an event-based parser, it works as an event handler in Java.

- Advantages
  - Since it reads each unit of XML, it creates an event so that the calling program can use it.
  - SAX uses what it likes to, by ignoring the bits which it doesn't care about.
  - It is memory efficient.
  - It's very fast and works for huge documents.
- Disadvantages
  - The main disadvantage of SAX is that the Calling program must keep track of everything it might ever need.
  - Since its Event-based, its API is less Intuitive.

Step 1.7.1.2: DOM

DOM stands for ‘Document Object Model’. A DOM Parser creates an internal structure in memory which is a DOM document object and the client applications get information of the original XML document by invoking methods on this document object. DOM Parser has a tree-based structure.

- Advantages
    - It supports both Read and Write operations.
    - It is preferred when there is random access to widely separated parts of the documents required.
    - It builds the entire XML document representation in memory and then hands the calling program the whole chunk of memory.
- Disadvantages
    - It consumes more memory since the whole XML document will be loaded into the memory.

Step 1.7.1.3: Pull Parser

Pull parser waits for the application to come calling. That is, they ask for the next available event, and the application basically loops until it runs out of XML.

- Advantages
    - It is designed to be used with large data sources.
    - Pull parser chooses to skip the events (whole section of the document) which it is not interested in.

Q8) Configure Selenium Grid using JSON.

Step 1.9.1: Configuring the grid hub using JSON

a. Create JSON file for the hub which will look like:

```
{
    "port": 4444,
    "newSessionWaitTimeout": -1,
    "servlets" : [],
    "withoutServlets": [],
    "custom": {},
    "capabilityMatcher": "org.openqa.grid.internal.utils.DefaultCapabilityMatcher",
    "throwOnCapabilityNotPresent": true,
    "cleanUpCycle": 5000,
    "role": "hub",
    "debug": false,
    "browserTimeout": 0,
    "timeout": 1800
}
```

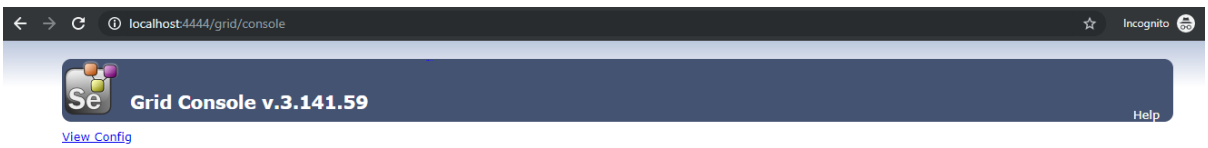
b. Save it in a folder with a valid name (example: myhub) in which we have saved Selenium standalone Server jar file.
c. Go to the command prompt.
d. Navigate to the folder structure where you have saved the Selenium standalone Server jar file.
e. Type the below command in the command prompt

Java -jar selenium-server-standalone-3.141.59.jar -role hub -hubConfig myhub.json

and click on **Enter**. It will look like:

```
15:44:15.021 INFO [GridLauncherV3.parse] - Selenium server version: 3.141.59, revision: e82be7d358
15:44:15.157 INFO [GridLauncherV3.lambda$buildLaunchers$5] - Launching Selenium Grid hub on port 4444
2019-08-20 15:44:15.581:INFO::main: Logging initialized @929ms to org.seleniumhq.jetty9.util.log.StdErrLog
15:44:16.217 INFO [Hub.start] - Selenium Grid hub is up and running
15:44:16.218 INFO [Hub.start] - Nodes should register to http://192.168.1.248:4444/grid/register/
15:44:16.218 INFO [Hub.start] - Clients should connect to http://192.168.1.248:4444/wd/hub
15:49:28.797 INFO [DefaultGridRegistry.add] - Registered a node http://192.168.1.248:5555
```

f. Open the Chrome browser.
g. Enter URL as 'http://localhost:4444/grid/console' and click on **Enter.**
h. Grid console page is loaded as below.



Step 1.9.2: Configuring the grid nodes using JSON

a. Once the Selenium Grid Hub using JSON is configured, the next step is to configure Selenium Grid nodes using JSON.

b. Create a JSON file for node, which will look like:

```
{
  "capabilities":
  [
    {
      "browserName": "firefox",
      "maxInstances": 5,
      "seleniumProtocol": "WebDriver"
    },
    {
      "browserName": "chrome",
      "maxInstances": 5,
      "seleniumProtocol": "WebDriver"
    }
  ],
  "proxy": "org.openqa.grid.selenium.proxy.DefaultRemoteProxy",
  "maxSession": 5,
  "port": 5555,
  "register": true,
  "registerCycle": 5000,
  "hub": "http://localhost:4444",
  "nodeStatusCheckTimeout": 5000,
  "nodePolling": 5000,
  "role": "node",
  "unregisterIfStillDownAfter": 60000,
  "downPollingLimit": 2,
  "debug": false,
  "servlets" : [],
  "withoutServlets": [],
  "custom": {}
}
```

c. Save it in a folder with a valid name (example: mynode) in which we have saved Selenium standalone Server jar file.

d. Open the new command prompt.

e. Navigate to the folder structure where you have saved the Selenium standalone Server jar file.

f. Type the below command in the command prompt

java -Dwebdriver.gecko.driver="geckodriver.exe" -Dwebdriver.chrome.driver="chromedriver.exe" -jar selenium-server-standalone-3.141.59.jar -role node -nodeConfig mynodes.json and click on **Enter** button, which will look like:

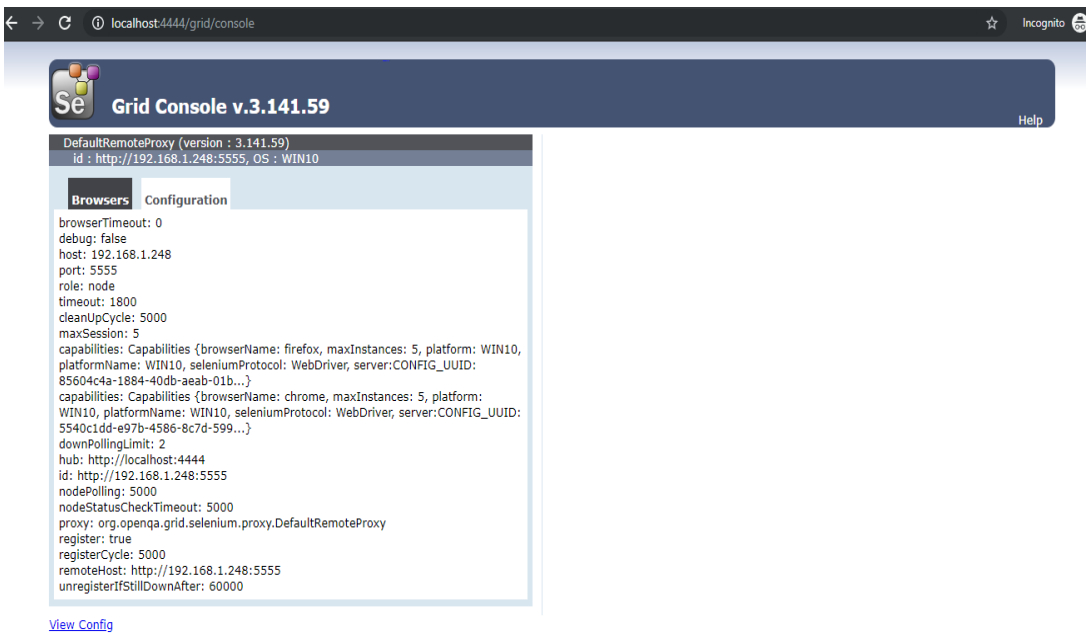
```
16:05:36.650 INFO [GridLauncherV3.parse] - Selenium server version: 3.141.59, revision: e82be7d358
16:05:36.809 INFO [GridLauncherV3.lambda$buildLaunchers$7] - Launching a Selenium Grid node on port 5555
2019-08-20 16:05:37.511:INFO::main: Logging initialized @1177ms to org.seleniumhq.jetty9.util.log.StdErrLog
16:05:37.856 INFO [WebDriverServlet.<init>] - Initialising WebDriverServlet
16:05:37.959 INFO [SeleniumServer.boot] - Selenium Server is up and running on port 5555
16:05:37.959 INFO [GridLauncherV3.lambda$buildLaunchers$7] - Selenium Grid node is up and ready to register to the hub
16:05:38.225 INFO [SelfRegisteringRemote$1.run] - Starting auto registration thread. Will try to register every 5000 ms.

16:05:38.769 INFO [SelfRegisteringRemote.registerToHub] - Registering the node to the hub: http://localhost:4444/grid/re
gister
16:05:38.992 INFO [SelfRegisteringRemote.registerToHub] - The node is registered to the hub and ready to use
```

g. Open the browser.

h. Enter URL as **http://localhost:4444/grid/console** and click on **Enter.**

i. The Grid console page will get loaded, which shows **Browsers** by default.

j. Click on **Configuration** which shows the configuration details.



Q9) Demonstrate Running Tests on Selenium Grid on Multiple Browsers .

- Open Eclipse.

- Click on **File>New>Other> Class.**
- Give a valid Class name (example: GridTest).
- Check the **public static void main** checkbox and click on **finish** , which will then create a blank Java class.
- Write the desired capabilities in the class, which will look like:

```
package testing.sidTesting;

import org.openqa.selenium.Platform;
import org.openqa.selenium.remote.DesiredCapabilities;

public class GridTest {

  public static void main(String[] args) {
    DesiredCapabilities cap = new DesiredCapabilities();
    cap.setBrowserName("chrome");
    cap.setPlatform(Platform.WIN10);
  }

}
```

- Start the selenium grid hub in the command prompt using **java -jar selenium-server-standalone-3.141.59.jar -role hub** command.
- Start the Selenium grid node in the command prompt using **java -Dwebdriver.chrome.driver="chromedriver.exe -jar selenium-server-standalone-3.141.59.jar -role node -hub** http://localhost:4444/grid/register command.
- Go to Eclipse and add a statement for remoteWebdriver, which has an implementation of WebDriver, to pass the hub port (http://192.168.1.248:4444/wd/hub), and DesiredCapabilities object as parameters.
- Write Selenium code to open the browser and navigate to any web page (example: Google page).

```java
import java.net.URL;

import org.openqa.selenium.Platform;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.remote.DesiredCapabilities;
import org.openqa.selenium.remote.RemoteWebDriver;

public class GridTest {

  public static void main(String[] args) throws MalformedURLException {
    DesiredCapabilities cap = new DesiredCapabilities();
    cap.setBrowserName("chrome");
    cap.setPlatform(Platform.WIN10);


    URL url = new URL("http://192.168.1.248:4444/wd/hub");
    WebDriver driver = new RemoteWebDriver(url, cap);

    driver.get("https://www.google.com");
    System.out.println("Google Title: " + driver.getTitle());

    driver.close();
  }

}
```

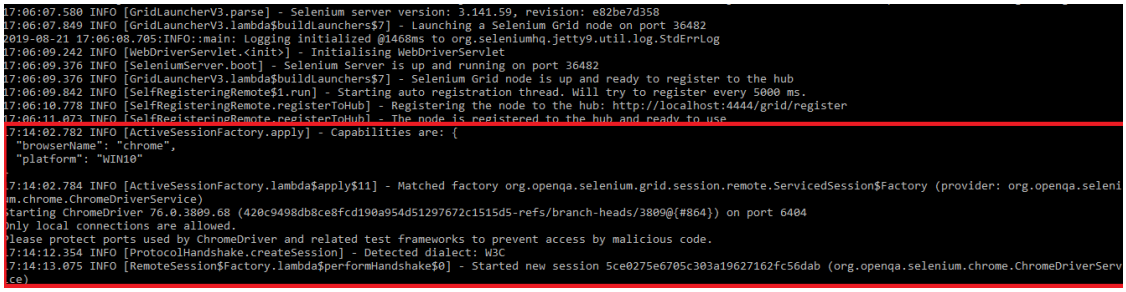
- Execute the Java program by right-clicking on the program and navigating to **Run As--> 1 Java Application.**
- This is how it looks like in the Eclipse console.

```
Aug 21, 2019 5:14:13 PM org.openqa.selenium.remote.ProtocolHandshake createSession
INFO: Detected dialect: W3C
Google Title: Google
```

- We can see that the capabilities passed through are displayed in both command prompts in the server (hub) as well as in clients (node).
- Selenium grid hub in command prompt with desired capabilities will look like:

```
17:01:00.989 INFO [GridLauncherV3.parse] - Selenium server version: 3.141.59, revision: e82be7d358
17:01:01.317 INFO [GridLauncherV3.lambda$buildLaunchers$5] - Launching Selenium Grid hub on port 4444
2019-08-21 17:01:02.084:INFO::main: Logging initialized @1821ms to org.seleniumhq.jetty9.util.log.StdErrLog
17:01:03.315 INFO [Hub.start] - Selenium Grid hub is up and running
17:01:03.317 INFO [Hub.start] - Nodes should register to http://192.168.1.248:4444/grid/register/
17:01:03.317 INFO [Hub.start] - Clients should connect to http://192.168.1.248:4444/wd/hub
17:06:11.072 INFO [DefaultGridRegistry.add] - Registered a node http://192.168.1.248:36482
17:14:02.672 INFO [RequestHandler.process] - Got a request to create a new session: Capabilities {browserName: chrome, platform: WIN10}
17:14:02.678 INFO [TestSlot.getNewSession] - Trying to create a new session on test slot {server:CONFIG_UUID=64178522-5111-44a4-b432-a610c8b45434, seleniumProtocol=WebDriver, browserName=chrome, maxInstances=5, platformName=WIN10, platform=WIN10}
```

- Selenium grid node in the command prompt with desired capabilities will look like:



Q10) Demonstrate page object design pattern in Selenium.

Step 1.11.1: Explaining why POM is used

- Consider this simple script to log in to the Rediff mail website.

```
public class Loginapplication {

   @Test
   public void Login()
   {
      System.setProperty("webdriver.chrome.driver","chromedriver.exe");
      WebDriver driver=new ChromeDriver();
      driver.get("https://mail.rediff.com/cgi-bin/login.cgi");
      //find user name and fill it
      driver.findElement(By.xpath(".//*[@id='login1']")).sendKeys("hello");
      //find password and fill it
      driver.findElement(By.name("passwd")).sendKeys("123");
      //click Go button
      driver.findElement(By.name("proceed")).click();
   }
}
```

- As you can see, all we are doing is finding elements and filling values for those elements.
- This is a small script. Script maintenance looks easy. But with time the test suite will grow. As you add more and more lines to your code, things become tough.
- The main problem with script maintenance is that if 10 different scripts are using the same page element, with any change in that element, you need to change all the 10 scripts. This is time-consuming and error-prone.
- A better approach to script maintenance is to create a separate class file,

which would find web elements, fill them, or verify them. This class can be reused in all the scripts using that element. In the future, if there is a change in the web element, we need to make the change in just 1 class file and not 10 different scripts.

Step 1.11.2: Implementing Page Object Model

- This is the basic structure of the Page Object Model (POM), where all Web Elements of the Application Under Test and the method that operate on these Web Elements, are maintained inside a class file.

```
public class RediffLoginPage {          //  Page class in object repository

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

  By username=By.xpath(".//*[@id='login1']");      // storing the web element
  By Password=By.name("passwd");
  By go=By.name("proceed");

  public WebElement Emaild()
  {
     return driver.findElement(username);      // finding the web element
  }
```

Step 1.11.3: Writing code to implement the Page Object Model using a test case

- Test Case: Log in to Rediff mail. Here we will be dealing with Login page POM and Test script.
- Login Page:

```
public class RediffLoginPage {            //  Page class in object repository

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

   By username=By.xpath(".//*[@id='login1']");      // storing the web element
   By Password=By.name("passwd");
   By go=By.name("proceed");

   public WebElement Emaild()
   {
       return driver.findElement(username);      // finding the web element
   }

   public WebElement Password()
   {
       return driver.findElement(Password);
   }

   public WebElement submit()
   {
       return driver.findElement(go);
   }

}
```

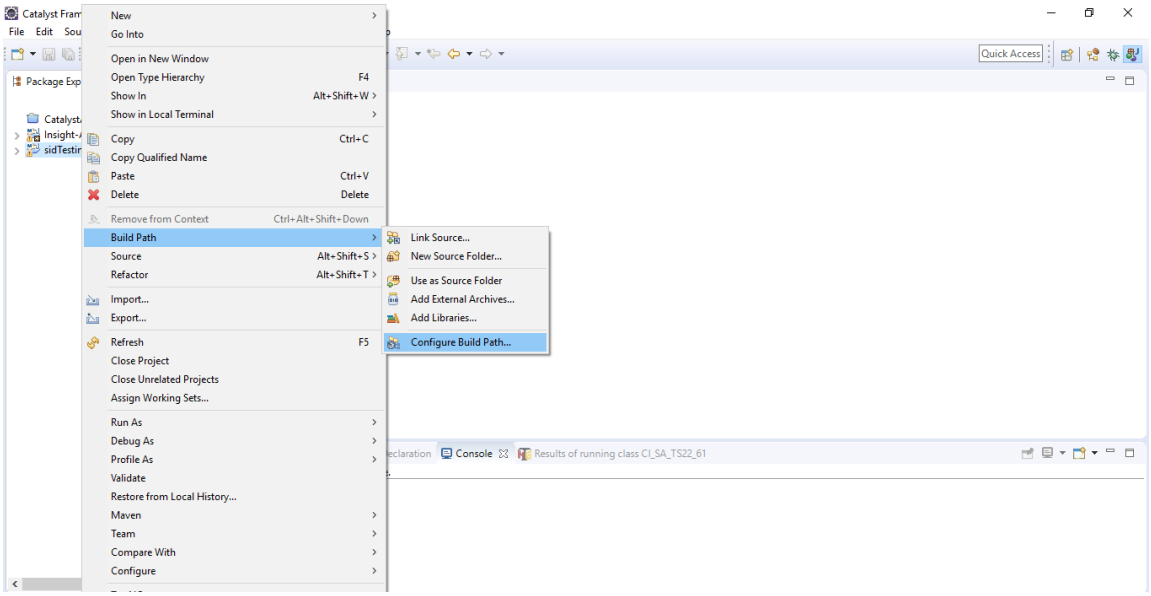
- Test Case:

```
public class LoginApplication {
/*
    * This test case will get  https://mail.rediff.com/cgi-bin/login.cgi
    * And Login to application
*/
   @Test
   public void Login()
   {
      System.setProperty("webdriver.chrome.driver", "chromedriver.exe");
      WebDriver driver=new ChromeDriver();
      driver.get("https://mail.rediff.com/cgi-bin/login.cgi");
      //create Login Page object
      RediffLoginpage rfLogin=new RediffLoginpage(driver);
      //enter Emaild
      rfLogin.Emaild().sendKeys("hello");
      //enter password
      rfLogin.Password().sendKeys("123");
      //click on Go button
      rfLogin.submit().click();
   }

}
```

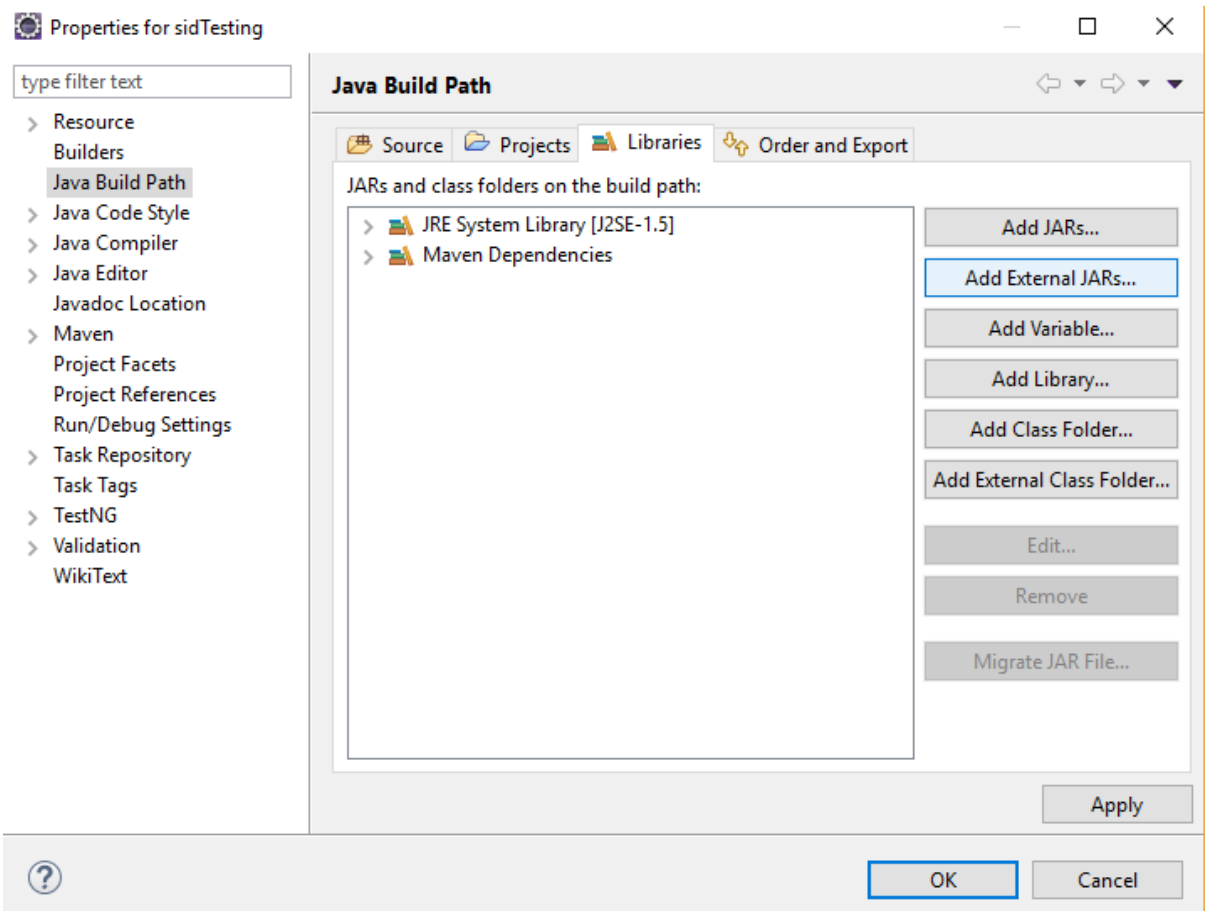
Q11) Demonstrate how Apache POI is configured in Selenium.

Step 1.12.1: Adding external jar files to the Selenium Project in Eclipse

a. Apache Poi is already installed in your Practice lab. Refer to QA to QE lab guide for Phase 2 for more information about its directory details.
b. Go to Eclipse.
c. Right-click on your project.
d. Navigate to **Build Path** **Configure Build Path.**



e. Click on **the Libraries** tab and click on **Add External JARs**.

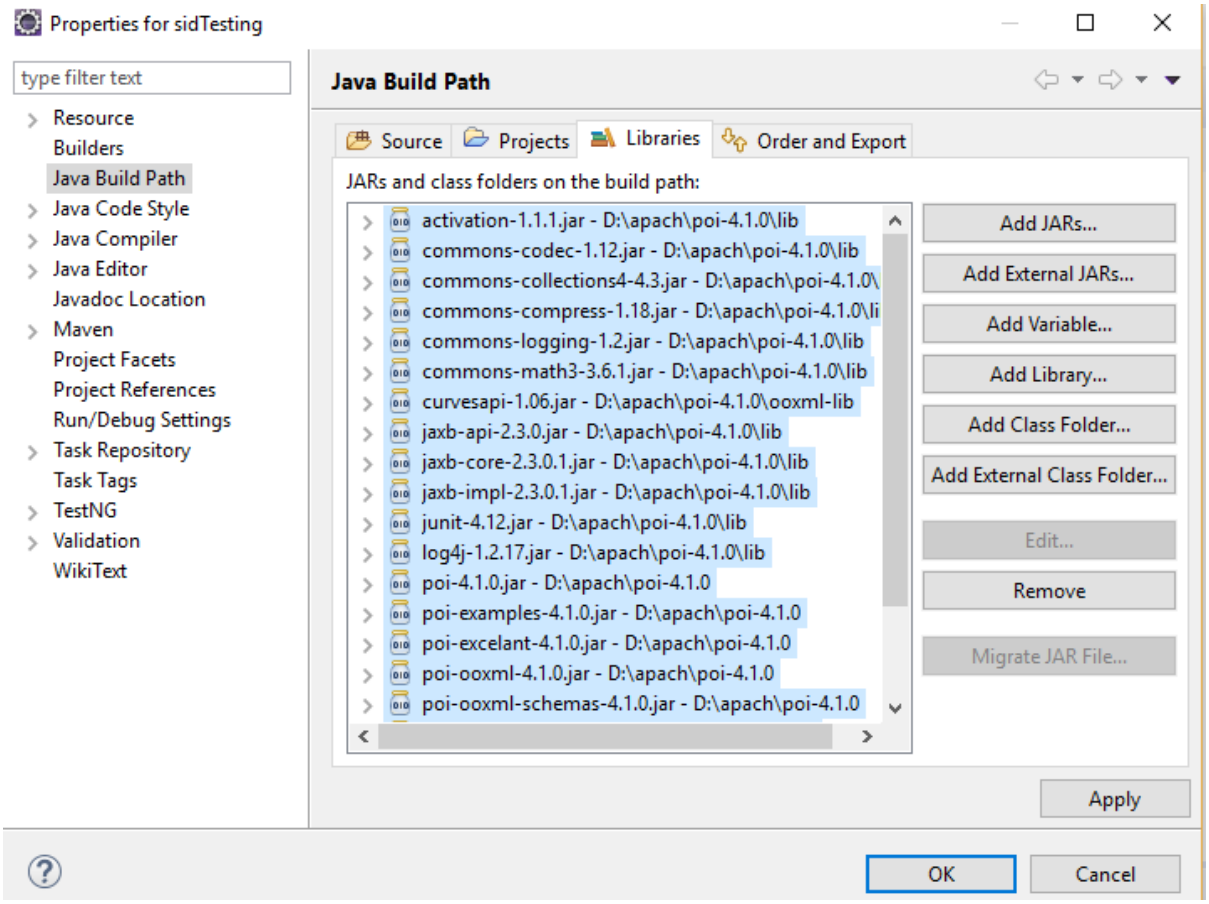


f. Navigate to the folder where Apache POI jar file is stored.

g. Select the executable JAR files from the lib folder and click on **Open**.

h. Repeat steps e.
i. Select the executable JAR files from the ooxml-lib folder and click on **Open**.
j. Repeat the steps e.
k. Select the remaining executable JAR files from the folder and click on **Open**.

l. The following snapshot shows that all the JARs are included in the Library list.
m. Click on **Apply** and then on the **OK** button.



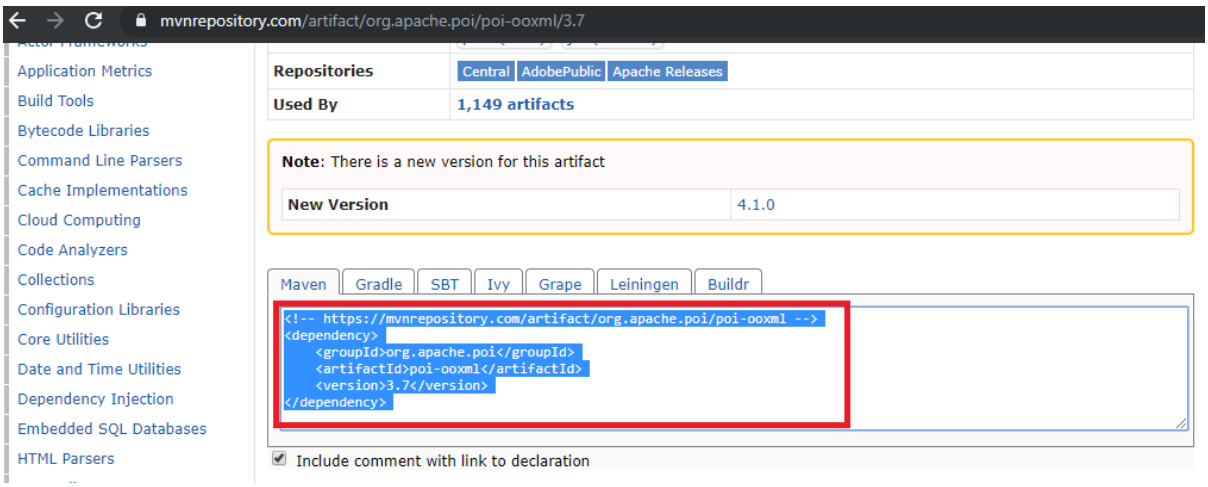
Step 1.12.2: Adding Maven Dependency

a. Go to the Maven Repository: https://mvnrepository.com/artifact/org.apache.poi/poi/3.7
b. Copy the dependency.

c. Go to Eclipse.
d. Open the Pom.xml file from the project.
e. Copy the dependency in the Pom.xml file:

```
<!-- https://mvnrepository.com/artifact/org.apache.poi/poi -->
<dependency>
    <groupId>org.apache.poi</groupId>
    <artifactId>poi</artifactId>
    <version>3.7</version>
</dependency>
```

f. Go to the Maven Repository: https://mvnrepository.com/artifact/org.apache.poi/poi-ooxml/3.7
g. Copy the dependency.

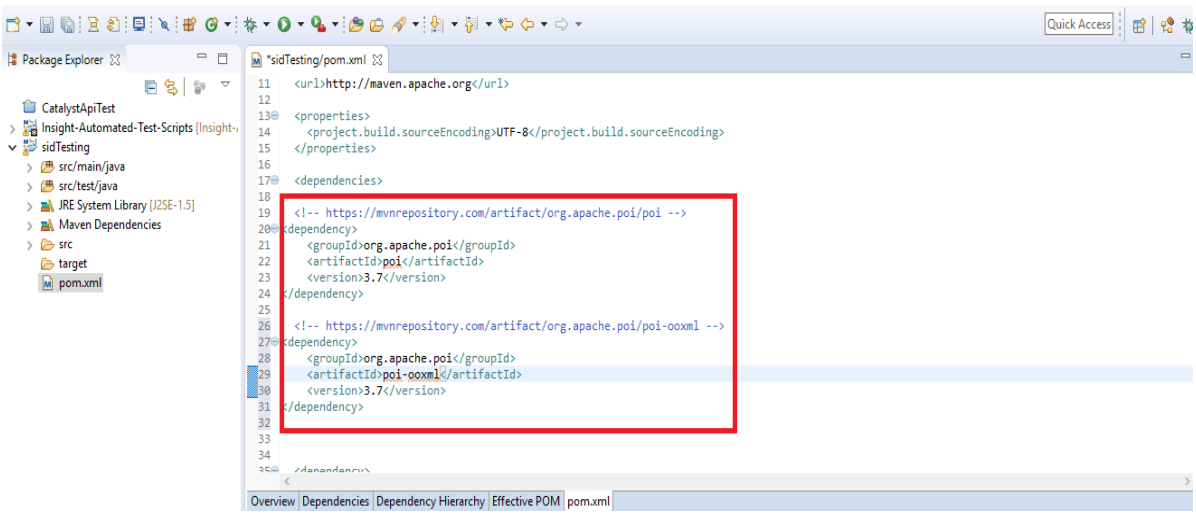


h. Go to Eclipse.

i. Open the Pom.xml file from the project.
j. Copy the dependency in the Pom.xml file.

```
 <!-- https://mvnrepository.com/artifact/org.apache.poi/poi-ooxml -->
<dependency>
    <groupId>org.apache.poi</groupId>
    <artifactId>poi-ooxml</artifactId>
    <version>3.7</version>
</dependency>
```

k. This is how the pom.xml file will look like in Eclipse:



Q12) Demonstrate how data is read from an Excel sheet in Selenium.

Reading the data from the Excel sheet:

- Create an Excel, enter the data, and Save it in a particular location.
- Create a Project.
- Open Eclipse -> File -> New -> Java Project.
- Create a class.
- Right-click on src (default package) -> New -> Class.
- Load the file and specify the path in it:

FileInputStream obj = new FileInputStream (“Path of the excel sheet”)

- Load the Workbook:

  Workbook obj1 = WorkbookFactory.create(obj);

- Load the Sheet:

  Sheet obj2 = obj1.getSheet(“Sheet1”);

- Specify which row you want to read:

Row obj3 = obj2.getRow(1);

- Specify which column to read and the data type:

  Cell obj4 = obj3.getCell(0);

  String st = obj4.getStringCellValue();

- Code in Eclipse will look like this:

```
//Load the file
FileInputStream fis = new
FileInputStream("/home/ubuntu/Desktop/Testdata.xlsx");
//Load the Workbook
Workbook wb = WorkbookFactory.create(fis);
//Load Sheet
Sheet sh = wb.getSheet("Sheet1");
//Specify which row you want to read
Row rw = sh.getRow(1);
//Specify which cell want to read and which data type
Cell cel = rw.getCell(0);
  String st = cel.getStringCellValue();
  System.out.println("Username is"+ st);
```

Step 1.13.2: Write the data in an Excel sheet

- Create an Excel file, enter the data, and save it in a particular location.
- Create a Project.
- Open Eclipse -> File -> New -> Java Project.
- Create a class.
- Right-click on src (default package) -> New -> Class.

- Load the file and specify the path in it.

  FileInputStream obj = new FileInputStream (“Path of the excel sheet”)

- Load the Workbook.

  Workbook obj1 = WorkbookFactory.create(obj);

- Load the Sheet.

  Sheet obj2 = obj1.getSheet(“Sheet1”);

- Specify which row you want to read:

Row obj3 = obj2.getRow(1);

- Specify in which column you want to write:

  Cell obj4 = obj3.getCell(0);

  obj4.setCellValue(“Enter the value”);

- Write the output to a file.

  FileOutputStream obj5 = new FileOutputStream(“Path of the excel sheet”);

  Obj1.write(obj5);

- Code in Eclipse will look like this:

```
//Load the file
FileInputStream fis = new
FileInputStream("/home/ubuntu/Desktop/Testdata.xlsx");
//Load the Workbook
Workbook wb = WorkbookFactory.create(fis);
//Load Sheet
Sheet sh = wb.getSheet("Sheet1");
//Specify which row you want to read
Row rw = sh.getRow(1);
//Specify which row want to read and which data type
Cell cel = rw.getCell(0);
cel.setCellValue("John");
//Write the output to a file
FileOutputStream fout = new
FileOutputStream("/home/ubuntu/Desktop/Testdata.xlsx");
wb.write(fout);
```