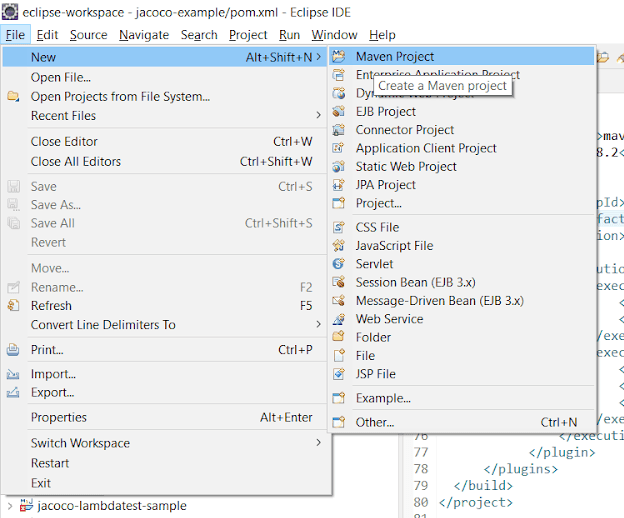
**Pre-requisites:**

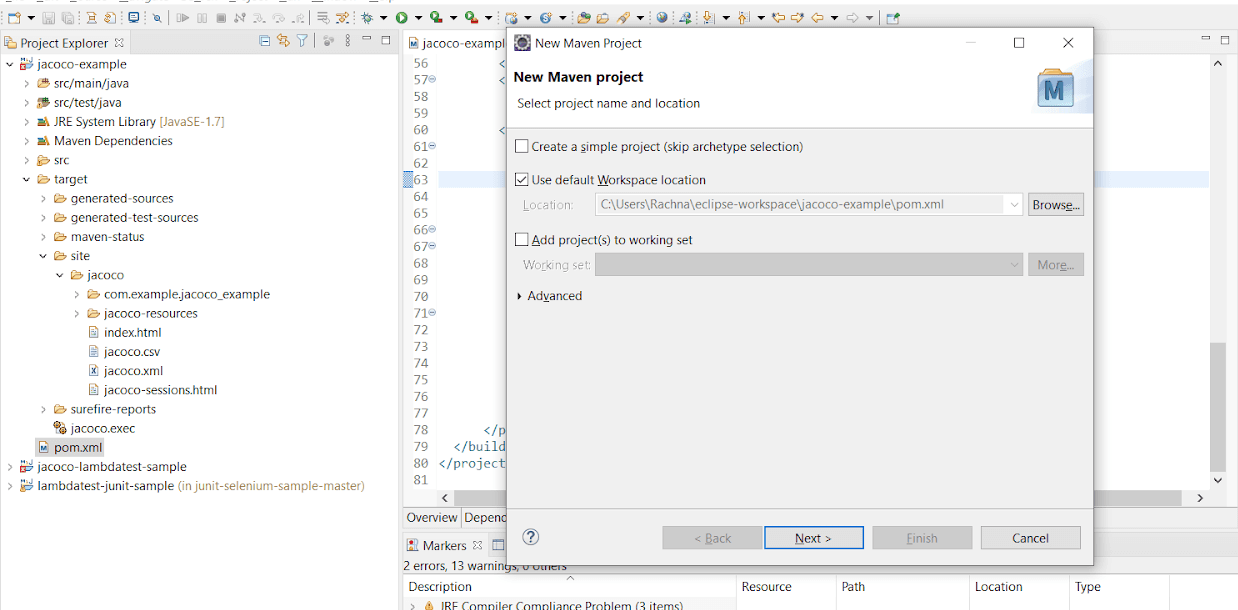
1. **Maven 3.0 or higher**: Maven is a project development management and comprehension tool. You can install the latest version from the official Apache Maven website.
2. **Java 1.5 or higher**: You need to ensure that Java is installed on your machine. In case you do not have Java installed, please download the latest version of Java from the official Java website.
3. **Eclipse IDE for Java Developers**: Though you can use the IDE of your choice, for this demo we have used the Eclipse IDE.

**Steps to create a simple Maven Project:**

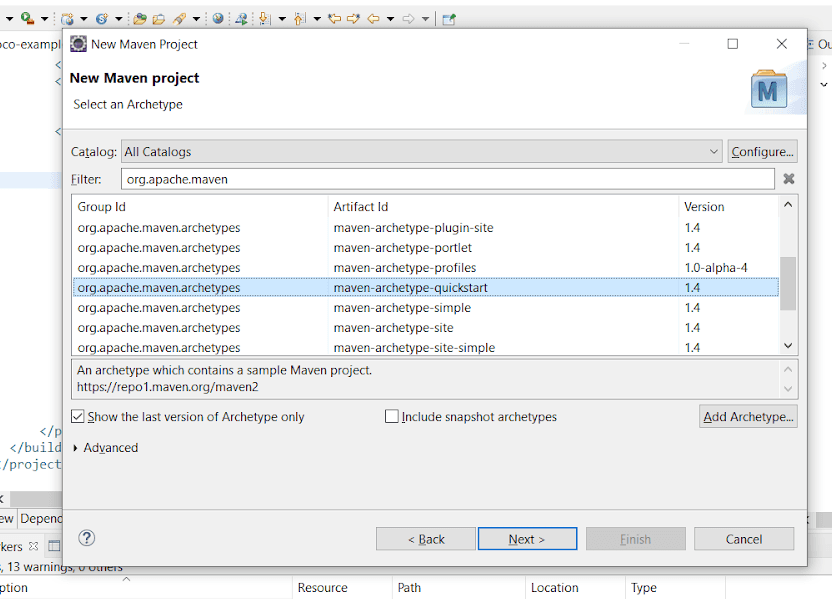
* 1. In Eclipse IDE, Go to File>New>Maven Project.



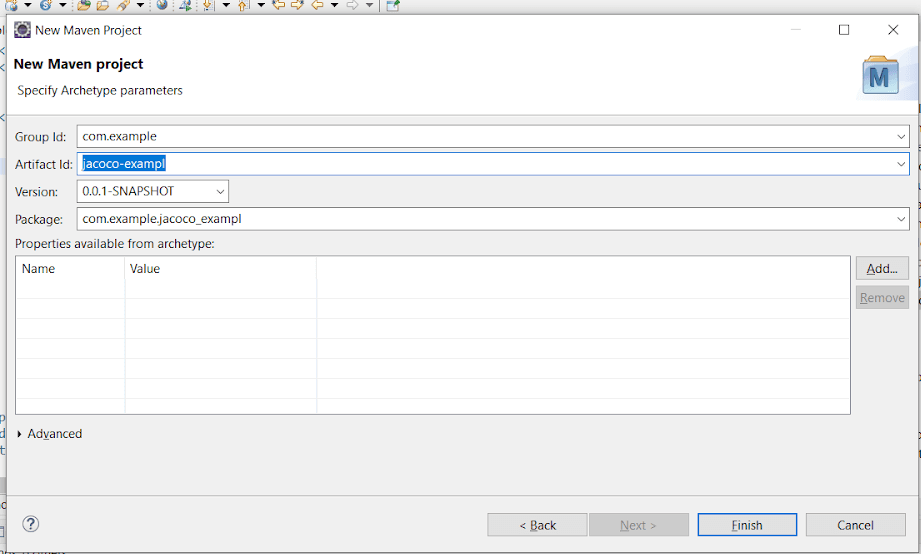
* 1. A new dialog box appears. Make sure that the ‘Use default Workspace location’ checkbox is ticked. Click Next.



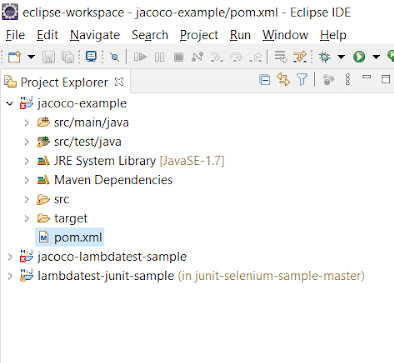
* 1. For selecting Archetype in the project, type org.apache.maven in the textbox that is located next to the Filter. Select maven-archetype-quickstart and click Next.



* 1. Now, specify the Group Id as com.example and Artifact Id as jacoco-example. The artifact Id is our project name. Finally, click on the Finish button.



* 1. You can see the project file and folder hierarchy in Project Explorer.



### How to specify JaCoCo Maven plugin in POM.xml

* 1. Open POM.xml, scroll to tag. We will specify the JaCoCo-Maven plugin in the section that lists the Maven plugins. Just copy the below code and paste it above the tag.

| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | org.jacoco  jacoco-maven-plugin  0.8.6    <!--first execution : for preparing JaCoCo runtime agent-->    prepare-agent    prepare-agent      <!--second execution : for creating code coverage reports-->    report  test    report |
| --- | --- |

* 1. Since we are demonstrating report generation for automated web testing with JUnit, we will also declare the JUnit dependency in POM.xml.Here is the entire content of POM.xml:

| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78 | <!--?xml version="1.0" encoding="UTF-8"?-->        4.0.0      com.example  jacoco-example  0.0.1-SNAPSHOT    jacoco-example        UTF-8  1.7  1.7        <!-- JUnit dependencies added to run test cases -->      junit  junit  4.11  test              <!-- Maven plugin for Project Management -->    maven-clean-plugin  3.0.0    <!-- see http://maven.apache.org/ref/current/maven-core/default-bindings.html#Plugin\_bindings\_for\_jar\_packaging -->    maven-resources-plugin  3.0.2      maven-compiler-plugin  3.7.0      maven-surefire-plugin  2.19.1      maven-jar-plugin  3.0.2      maven-install-plugin  2.5.2      maven-deploy-plugin  2.8.2      org.jacoco  jacoco-maven-plugin  0.8.6        prepare-agent  prepare-agent      report  test  report |
| --- | --- |

1. From the project directory, traverse to com.example.jacoco\_lambdatest package existing in src/main/java. Create a new Java class named LambdaTest.java. We will write a simple setUp() function in it that provides the desired capabilities of Selenium Grid.

| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29 | package com.example.Jacoco\_lambdatest;    import java.net.MalformedURLException;  import java.net.URL;  import org.junit.Before;  import org.junit.Test;  import org.openqa.selenium.By;  import org.openqa.selenium.remote.DesiredCapabilities;  import org.openqa.selenium.remote.RemoteWebDriver;    public class LambdaTest {  public static String username = "";  public static String accessKey = "";  public static DesiredCapabilities setUp() throws Exception {  DesiredCapabilities capabilities = new DesiredCapabilities();  capabilities.setCapability("platform", "Windows 10");  capabilities.setCapability("browserName", "Chrome");  capabilities.setCapability("version", "87.0"); // If this cap isn't specified, it will just get the any available one  capabilities.setCapability("resolution","1024x768");  capabilities.setCapability("build", "First Test");  capabilities.setCapability("name", "Sample Test");  capabilities.setCapability("network", true); // To enable network logs  capabilities.setCapability("visual", true); // To enable step by step screenshot  capabilities.setCapability("video", true); // To enable video recording  capabilities.setCapability("console", true); // To capture console logs  return capabilities;  }  } |
| --- | --- |

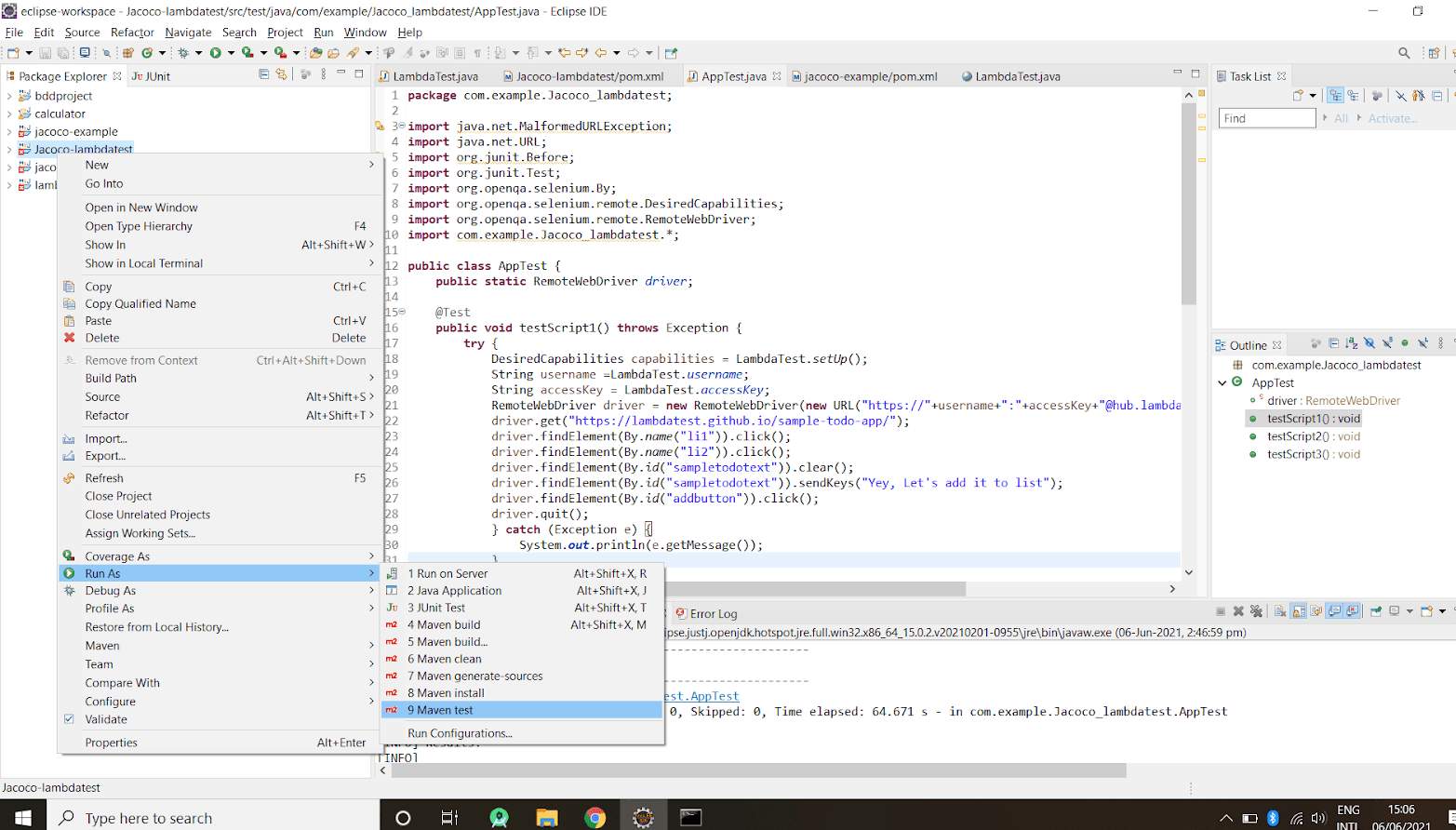
**Adding JUnit test Cases in the project:**

* 1. We will create a simple JUnit test case in AppTest.java. This is provided by default, in the src/test/java under the package name com.example.jacoco\_lambdatest.

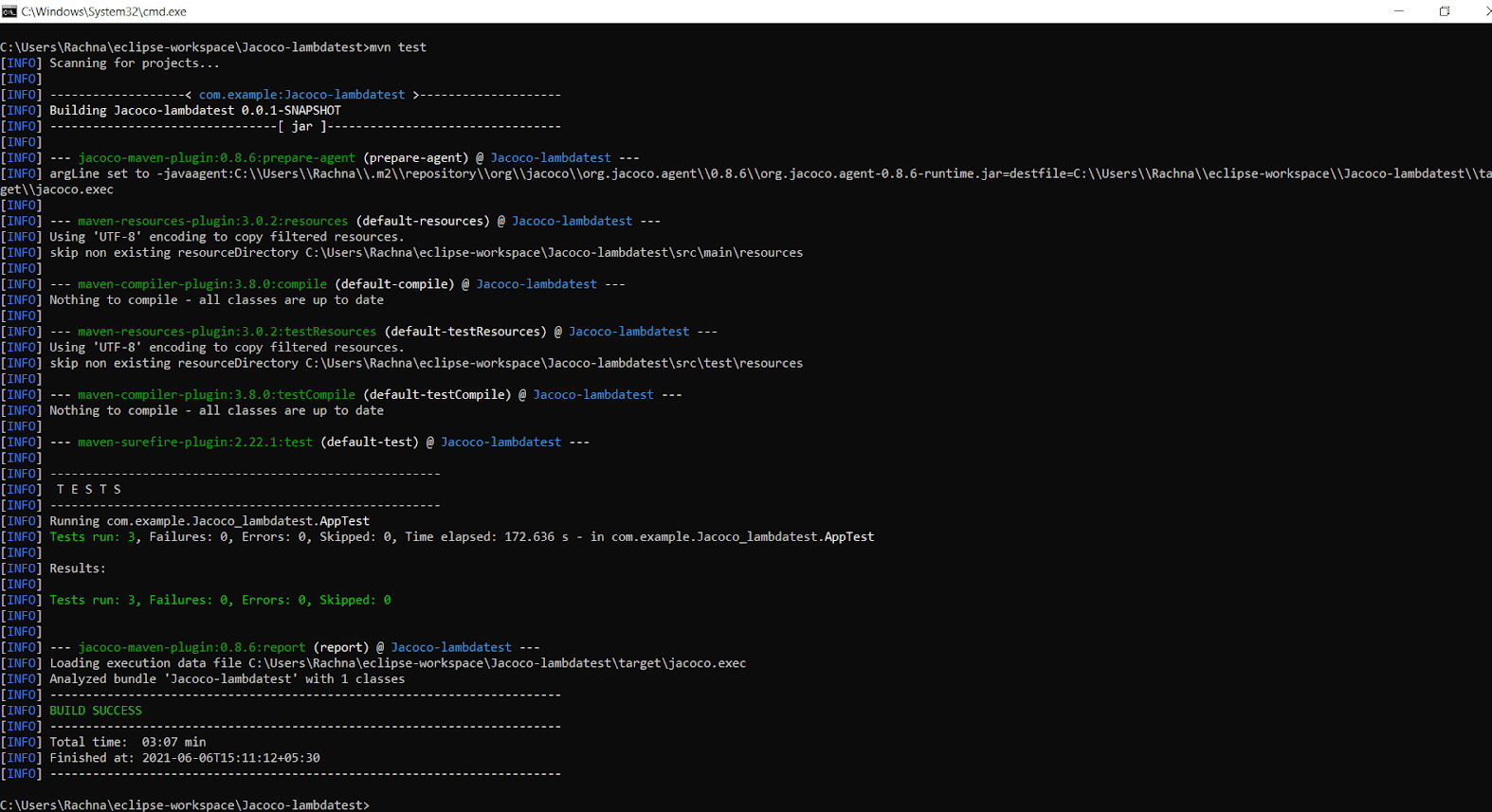
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30 | package com.example.Jacoco\_lambdatest;    import java.net.MalformedURLException;  import java.net.URL;  import org.junit.Before;  import org.junit.Test;  import org.openqa.selenium.By;  import org.openqa.selenium.remote.DesiredCapabilities;  import org.openqa.selenium.remote.RemoteWebDriver;  import com.example.Jacoco\_lambdatest.\*;    public class AppTest {  public static RemoteWebDriver driver;  @Test  public void testScript1() throws Exception {  try {  DesiredCapabilities capabilities = LambdaTest.setUp();  String username =LambdaTest.username;  String accessKey = LambdaTest.accessKey;  RemoteWebDriver driver = new RemoteWebDriver(new URL("https://"+username+":"+accessKey+"@hub.lambdatest.com/wd/hub"),capabilities); driver.get("https://lambdatest.github.io/sample-todo-app/");  driver.findElement(By.name("li1")).click();  driver.findElement(By.name("li2")).click(); driver.findElement(By.id("sampletodotext")).clear(); driver.findElement(By.id("sampletodotext")).sendKeys("Yey, Let's add it to list");  driver.findElement(By.id("addbutton")).click();  driver.quit();  } catch (Exception e) {  System.out.println(e.getMessage());  }  }  } |
| --- | --- |

### Generating code coverage reports:

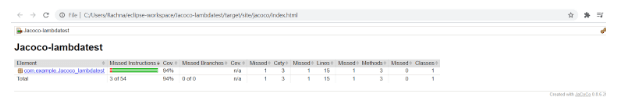
* 1. Just click on the Run As button and set the configuration as Maven Test.



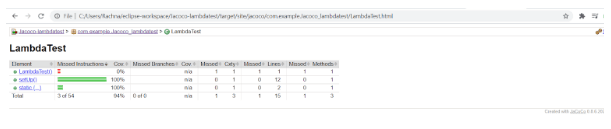
* 1. Instead, you can open cmd(Command Line), traverse to the project folder, and run the maven command, “mvn test.”



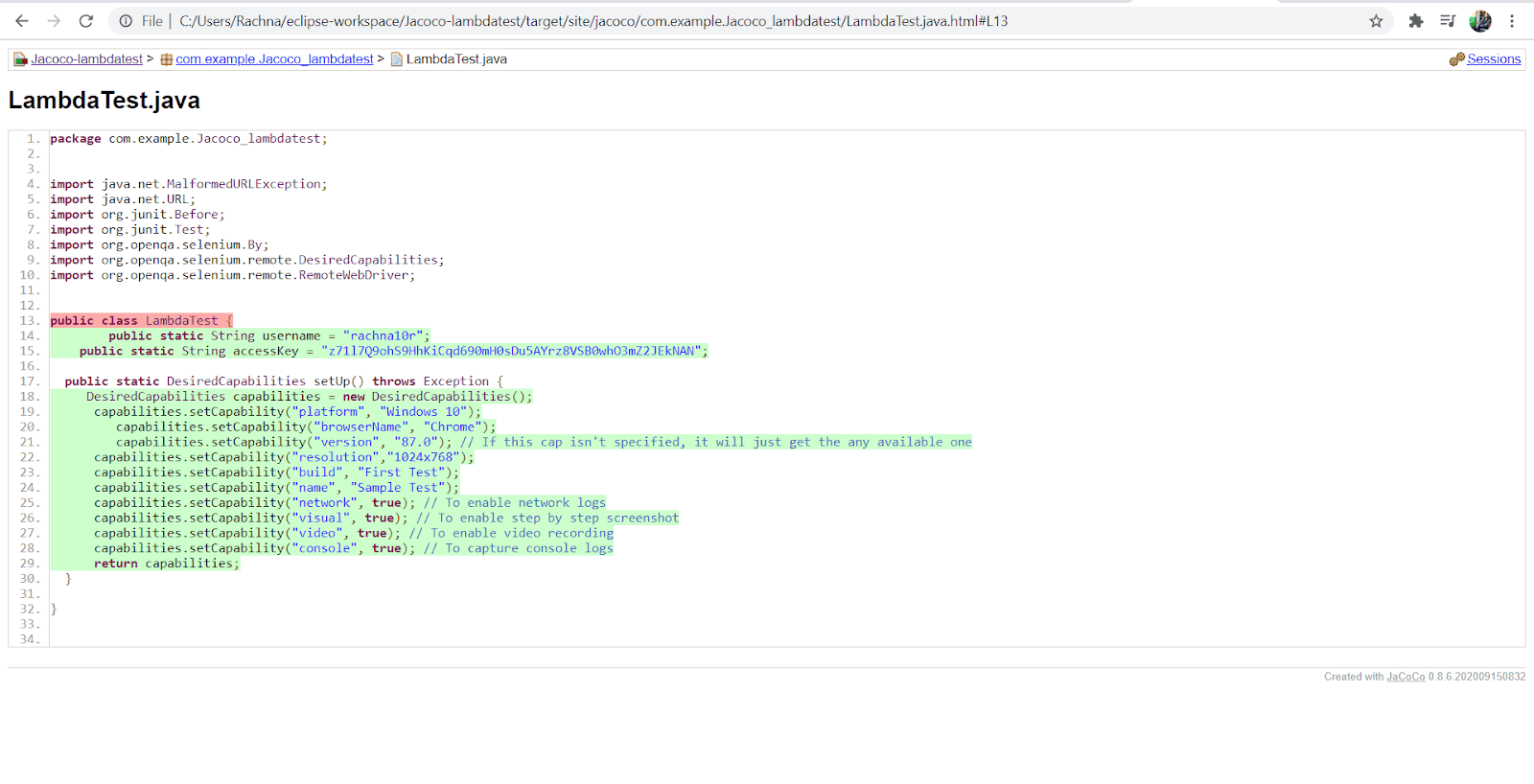
* 1. Running the JUnit tests will automatically set the JaCoCo agent in motion. It will create a report in binary format in the target directory, with path target/jacoco.exec. The output of jacoco.exec cannot be interpreted single-handedly but other tools like SonarQube and plugins can interpret it. As we have earlier specified the jacoco:report goal, it will generate readable code coverage reports in popular formats like HTML, CSV, and XML.
  2. As the build is successful, go to the target folder, then to the site>jacoco folder. The code coverage report (i.e. index.html) is located in target/site/jacoco/index.html. The report looks like this:



* 1. You can drill down at a micro level by clicking on com.example.jacoco\_lambdatest>LambdaTest in the report.



* 1. By clicking on specific functions, you will have a more detailed view in LambdaTest.java.



1. Here, you would see a lot of diamonds of different colors like green, yellow and red. These are the specifications used in the report to symbolize which line of code was executed and when it was executed. We will learn more about it in the next section of the report analysis. With this, you have successfully generated a code coverage report via the Jacoco Maven plugin.

### Analysis of the Code Coverage Report

Our code coverage report shows 94% instruction coverage, 100% branch coverage, which is a great code coverage score. Later, we will try to achieve a 100% code coverage score by adding more test cases.

The 38 instructions shown by JaCoCo in the report refer to the bytecode instructions instead of Java code instructions. The JaCoCo reports help you visually analyze  
code coverage by using diamonds with colors for branches and background highlight colors for lines. A brief explanation of the diamonds seen in the code coverage report is below:

* **Red diamond** indicates that no branches have been exercised during the testing phase
* **Yellow diamond** indicates that the code is partially covered (i.e. some branches are not exercised)
* **Green diamond** indicates that all branches are exercised during the test

The same color code applies to the background highlight color for the line coverage.  
The report mainly provides three crucial metrics :

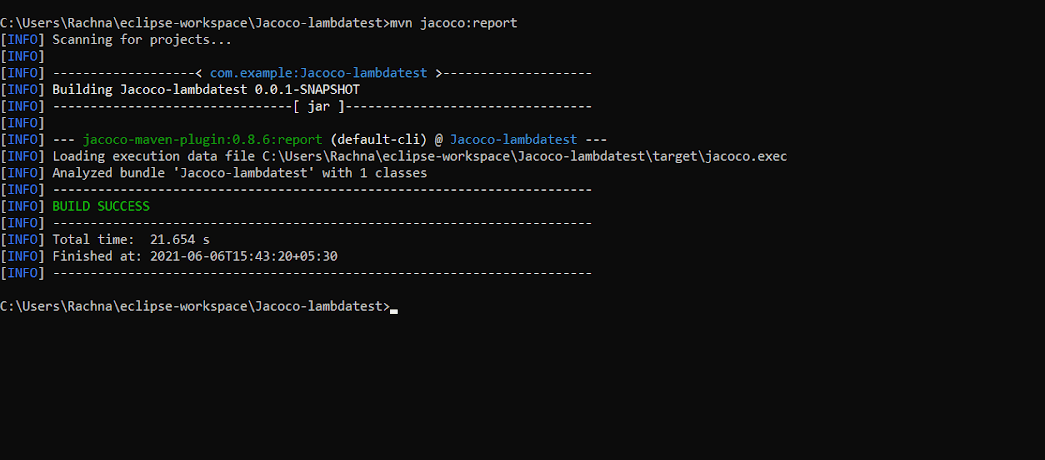
1. **Line coverage**: This reflects the amount of code exercised based on the number of Java byte code instructions called by the tests.
2. **Branch coverage**: This shows the percentage of exercised branches in the source code. These are typical if/else or switch statements.
3. **Cyclomatic complexity**: This reflects code complexity via the number of paths needed to cover all possible paths in a code. It also refers to the number of test cases needed to implement to cover the entire code. As there is no switch or statement in the code, the cyclomatic complexity will be 1; only one execution path is sufficient to cover the entire code.

### Introducing more test cases for improving code coverage

* 1. To achieve better code coverage, more tests need to be introduced that test the code that was not covered earlier via the test implementation.
  2. Go to AppTest.java in src/test/java to add more test cases.
  3. The new test cases added to AppTest.java will look as follows:

| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30 | @Test  public void testScript2() throws Exception {  try {  DesiredCapabilities capabilities = LambdaTest.setUp();  String username = LambdaTest.username;  String accessKey = LambdaTest.accessKey;  RemoteWebDriver driver = new RemoteWebDriver(new URL("https://"+username+":"+accessKey+"@hub.lambdatest.com/wd/hub"),capabilities); driver.get("https://lambdatest.github.io/sample-todo-app/"); driver.findElement(By.name("li2")).click();  driver.findElement(By.name("li3")).click();  driver.findElement(By.id("sampletodotext")).clear(); driver.findElement(By.id("sampletodotext")).sendKeys("Yes, Let's add it to list"); driver.findElement(By.id("addbutton")).click();  driver.quit();  } catch (Exception e) {  System.out.println(e.getMessage());  }  }    @Test  public void testScript3() throws Exception {  try {  DesiredCapabilities capabilities = LambdaTest.setUp();  String username = LambdaTest.username;  String accessKey = LambdaTest.accessKey;  RemoteWebDriver driver = new RemoteWebDriver(new URL("https://"+username+":"+accessKey+"@hub.lambdatest.com/wd/hub"),capabilities); driver.get("https://lambdatest.github.io/sample-todo-app/"); driver.findElement(By.name("li4")).click();  driver.findElement(By.id("sampletodotext")).clear(); driver.findElement(By.id("sampletodotext")).sendKeys("Yes, Let's add it!");  driver.findElement(By.id("addbutton")).click();  driver.quit();  } catch (Exception e) {  System.out.println(e.getMessage());  }  } |
| --- | --- |

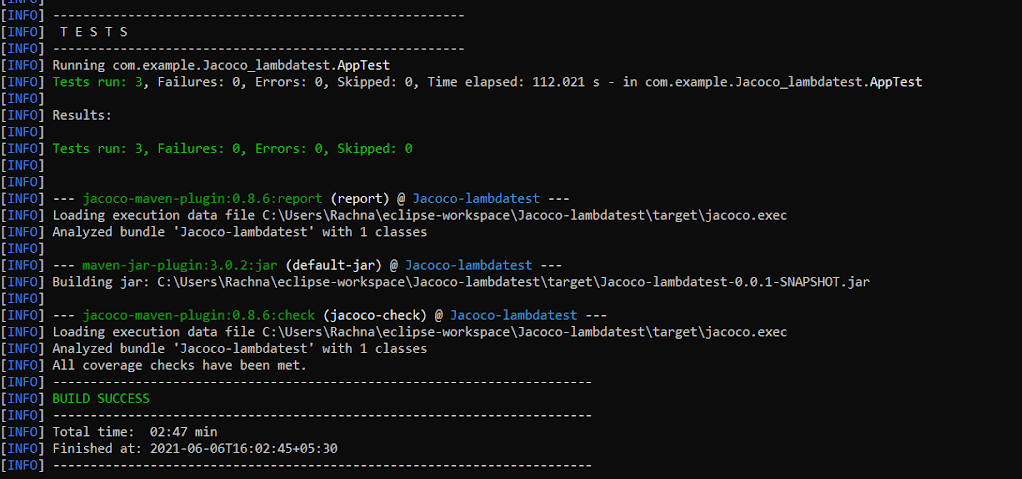
* 1. Let’s run the maven jacoco:report to publish a new coverage report.



* 1. JaCoCo offers a simple and easy way to track code coverage score by declaring minimum requirements. Build fails if these requirements are not met else the build is successful.
  2. These requirements can be specified as rules in POM.xml. Just specify the new execution specifying ‘check’ goal in POM.xml. Add the below code after the second <execution> tag in POM.xml.

| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | <!--Third execution : used to put a check on the entire package-->    jacoco-check    check          PACKAGE      LINE  COVEREDRATIO  0.50 |
| --- | --- |

* 1. With this, we are limiting our coverage ratio to 50%. This signifies a minimum of 50% of the code should be covered during the test phase.
  2. You can run maven clean verify to check whether the rules set in jacoco:check goal are met or not.



1. The log shows “All coverage checks have been met.” as our code coverage score is 94% which is greater than our minimum 50%.

## **Automation Testing On LambdaTest Selenium Grid using Maven Project with Jacoco Plugin**

Selenium testing on the cloud helps you attain better browser coverage, increased test coverage, and accelerated time to market. [Parallel testing in Selenium](https://www.lambdatest.com/blog/what-is-parallel-testing-and-why-to-adopt-it/) helps you achieve the above mentioned requirements.

LambdaTest [Cloud Selenium Grid](https://www.lambdatest.com/selenium-automation) is a cloud-based scalable Selenium testing platform that enables you to run your automation scripts on 2000+ different browsers and operating systems.

**Pre-requisites:**

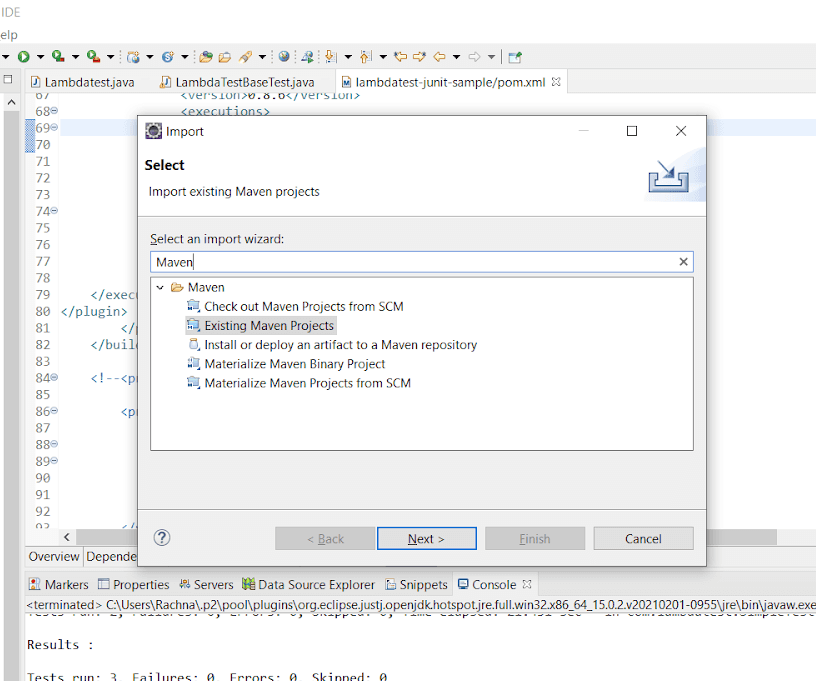
To run the test script using JUnit with Selenium, first, we need to set up an environment. You would first need to create an account on LambdaTest. Do make a note of the username and access-key that is available in [LambdaTest profile section](https://accounts.lambdatest.com/detail/profile).

We will use this [sample project](https://github.com/rachnaagrawal/junit-selenium-sample.git) for Java Selenium testing.

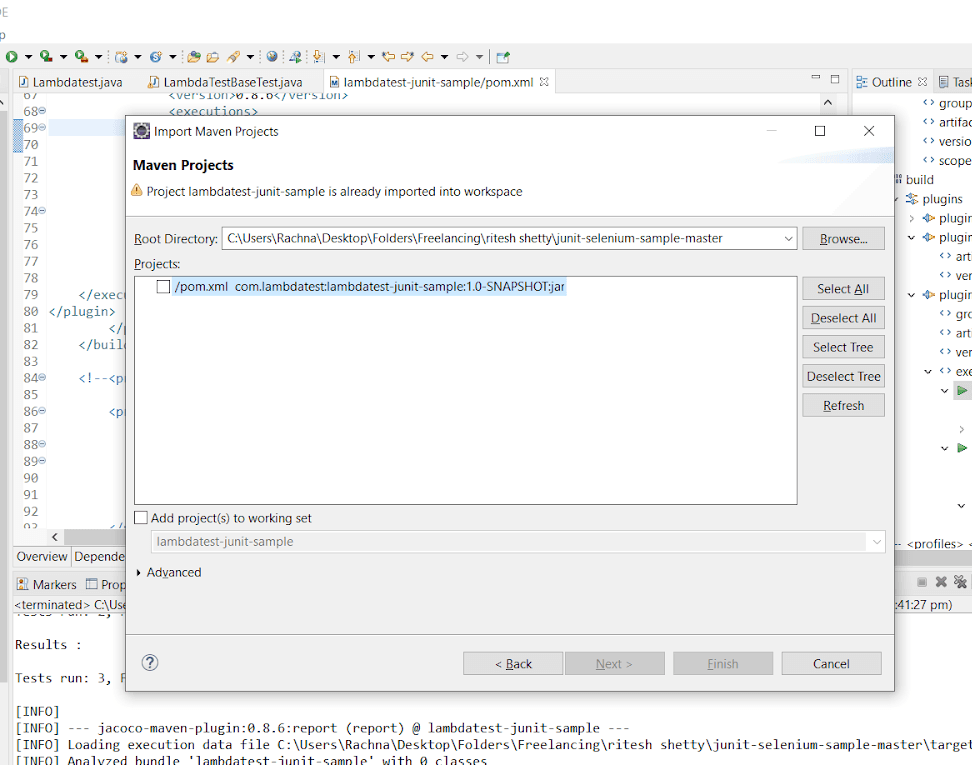
### Importing project to Eclipse IDE:

After downloading a zip file of the project: junit-selenium-sample from GitHub, we import it to Eclipse IDE by following the below mentioned steps:

* 1. Go to your Eclipse IDE, click on the File menu and select Import. A new dialog box appears.
  2. Type Maven in the textbox below and select Existing Maven Projects, and then click Next.



* 1. In the next dialog box, click on Browse and traverse to the project folder. Also, tick the checkbox giving the path to the POM.xml file, and click Finish.



1. Your project will be loaded in Eclipse IDE successfully.

### Adding dependencies in the POM.xml file:

* 1. Open the POM.xml, now add the dependencies of JUnit, Selenium, and JaCoCo Maven Plugin. After adding the dependencies to the code of POM.xml should look like this:

| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70 | <!--?xml version="1.0" encoding="UTF-8"?-->    4.0.0    com.lambdatest  lambdatest-junit-sample  1.0-SNAPSHOT  UTF-8  2.19.1  default        <!--JUnit dependency-->    junit  junit  4.12  test      commons-io  commons-io  1.3.2  test    <!--Selenium dependency-->    org.seleniumhq.selenium  selenium-java  2.52.0  test      com.googlecode.json-simple  json-simple  1.1.1  test            <!--Apache Maven Plugins-->    maven-compiler-plugin  3.0    1.8  1.8        maven-surefire-plugin  2.19.1    <!--JaCoCo Maven Plugin-->    org.jacoco  jacoco-maven-plugin  0.8.6      prepare-agent  prepare-agent      report  test  report |
| --- | --- |

| 1 | <em><a href="https://github.com/rachnaagrawal/junit-selenium-sample/blob/master/pom.xml" target="\_blank" rel="nofollow noopener">Github</a></em> |
| --- | --- |

### Configuring the Desired Capabilities for JUnit Automation Testing:

* 1. To connect to LambdaTest Selenium Automation Grid, the first thing done is invoking a remote webdriver. This remote driver requires some capabilities like browser, browser versions, operating system, etc., to build an environment. The code to it looks as follows:

| 1  2 | WebDriver driver = new RemoteWebDriver(new URL("https://" + username + ":" + accesskey + "@hub.lambdatest.com/wd/hub"),  DesiredCapabilities.firefox()); //A class named DesiredCapabilities is used to create an environment as a Firefox browser. |
| --- | --- |

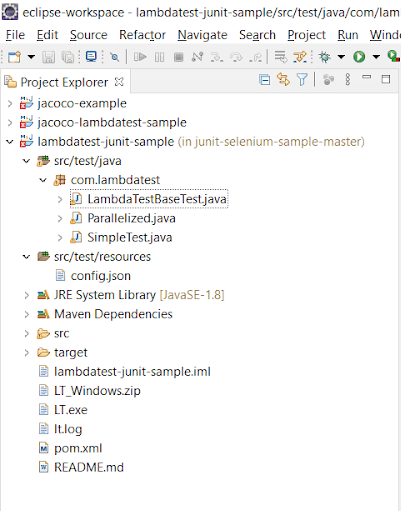
* 1. In the JUnit automation testing code, the capabilities like browser, browser versions, operating system information, etc., can be customized and are passed via Capabilities Object.
  2. LambdaTest has made this process very easy by providing an inbuilt Capabilities Generator. The [Capabilities Generator](https://www.lambdatest.com/capabilities-generator/) will automatically generate the code for desired capabilities based on your input. Such as, our configurations are:

| Fields | Selected Values |
| --- | --- |
| Operating Systems | Windows 10 |
| Browser | Chrome |
| Browser Version | 62.0 |
| Resolution | 1024×768 |
| Selenium Version | 3.13.0 |

1. Selecting the above-specified configuration in the Capabilities generator and paste it to LambdaTestBseTest.java.

### Specifying LambdaTest Username and Access Key in required Java Class:

* 1. In the Project Explorer, you would see 3 Java classes: LambdaTestBaseTest.java (contains the setup required for Java Selenium testing), Parallelized.java (contains Junit tests for parallel testing on LambdaTest Selenium grid), and SimpleTest.java (contains simple unit tests).



1. LambdaTestBaseTest.java fetches the required data like desired capabilities, username, and access key from a config file. This is provided in src/test/resources as config.json.
2. Specify the desired capabilities, Username, and Access key in config.json. This JSON file is used as you can provide multiple configurations in it for realizing [parallel testing with Selenium](https://www.lambdatest.com/blog/speed-up-automated-parallel-testing-in-selenium-with-testng/). to specify multiple configurations in config.json and then fetch them later.

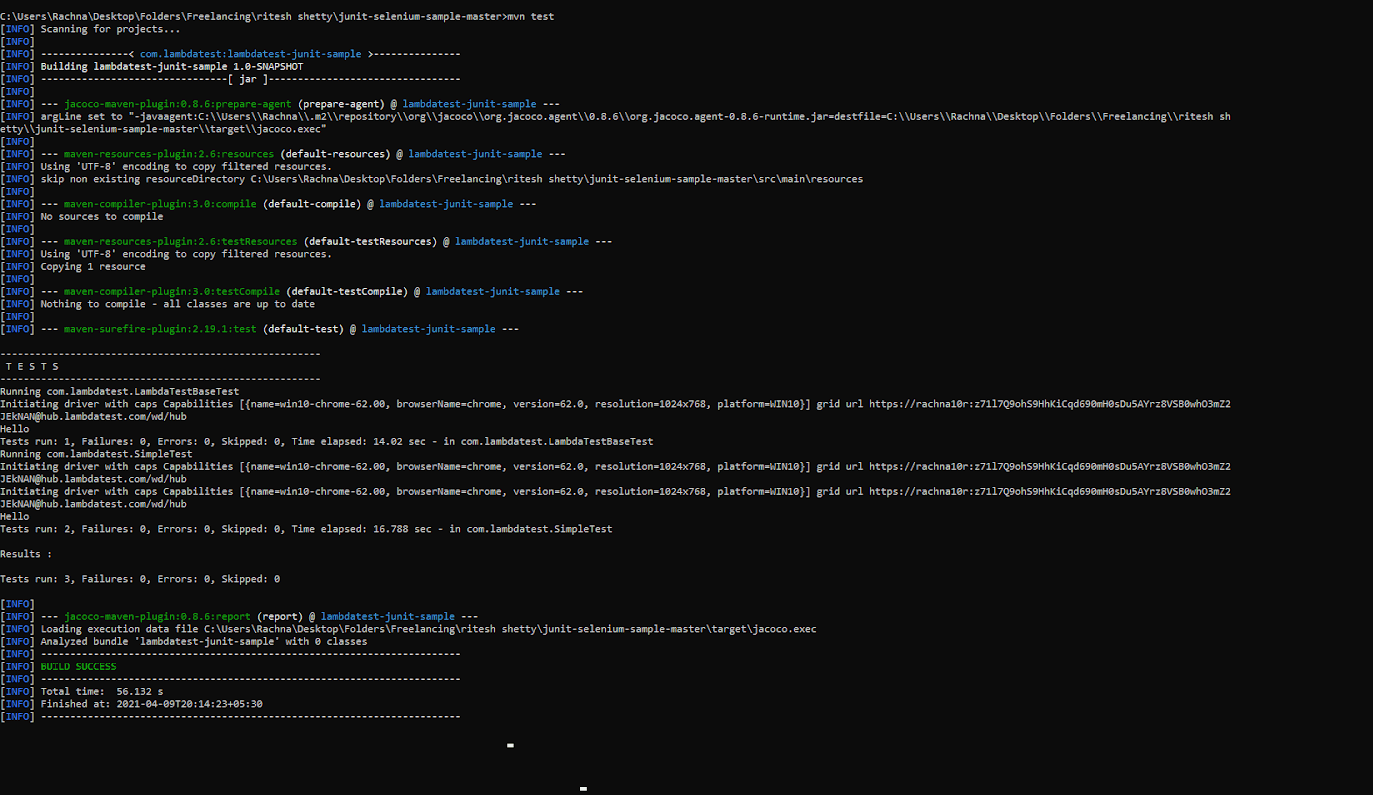
| 1  2  3  4  5  6  7  8  9 | [ { "tunnelName":"LambdaTest tunnel",  "buildName":"running Java Selenium Tests",  "testName":"Jacoco JUnit Test",  "username": "user-name",  "access\_key":"access-key",  "operatingSystem" : "win10",  "browserName" : "chrome",  "browserVersion" : "62.0",  "resolution" : "1024x768" }] |
| --- | --- |

### Unit Testing using JUnit with Selenium:

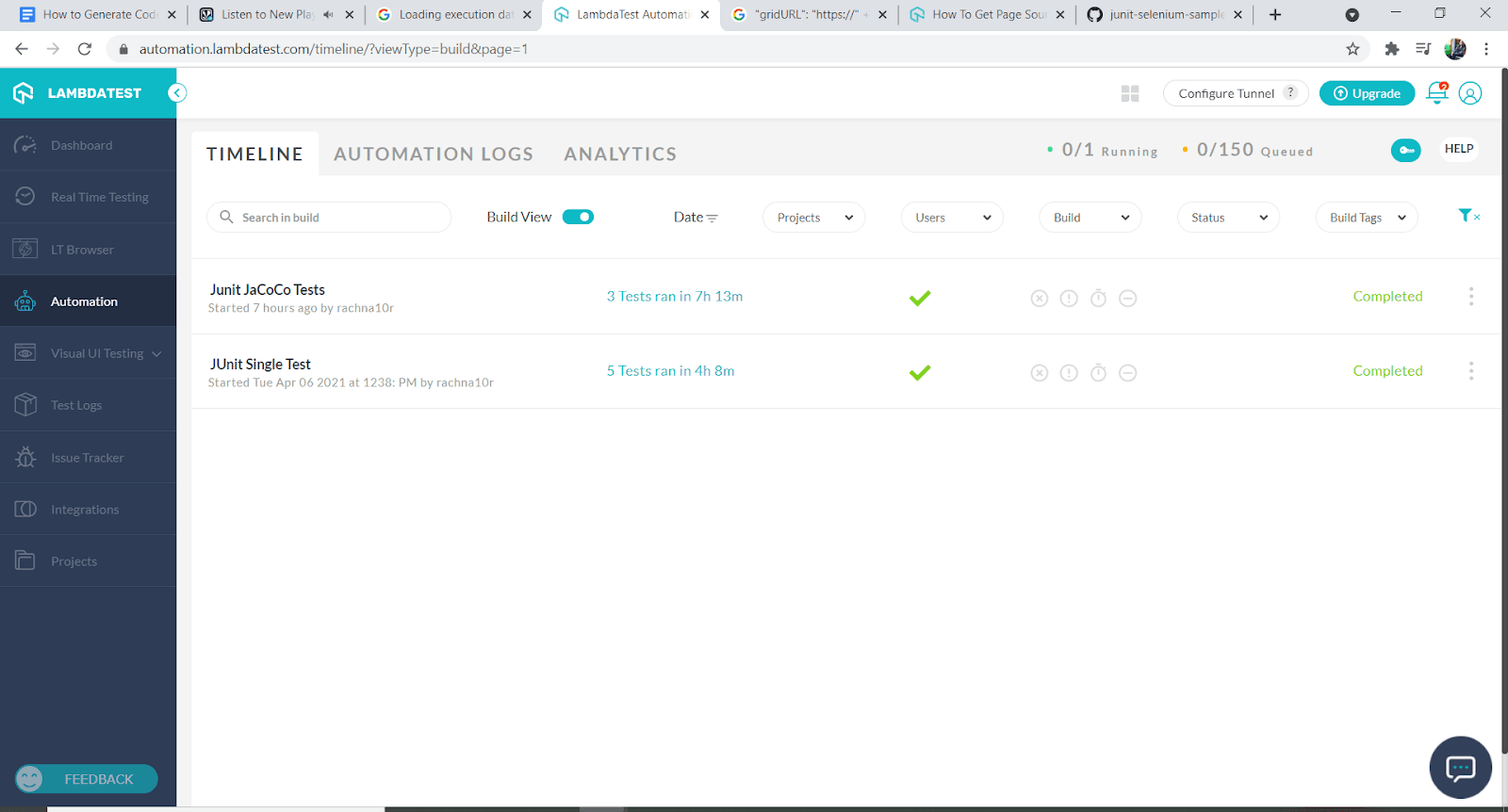
* 1. SimpleTest.java is the Java class for specifying a single unit test case for testing and performing code coverage using the JaCoCo Maven plugin.

| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24 | package com.lambdatest;    import com.lambdatest.LambdaTestBaseTest;  import org.junit.Test;  import org.openqa.selenium.By;  import static org.junit.Assert.assertEquals;    public class SimpleTest extends LambdaTestBaseTest {  /\*\*  \* Simple Test case annotation for JUnit Test  \* @throws Exception  \*/    @Test  public void validateUser() throws Exception {  driver.get("https://lambdatest.github.io/sample-todo-app/");  driver.findElement(By.name("li1")).click();  driver.findElement(By.name("li2")).click();  driver.findElement(By.id("sampletodotext")).clear();  driver.findElement(By.id("sampletodotext")).sendKeys("Yey, Let's add it to list");  driver.findElement(By.id("addbutton")).click();  }    } |
| --- | --- |

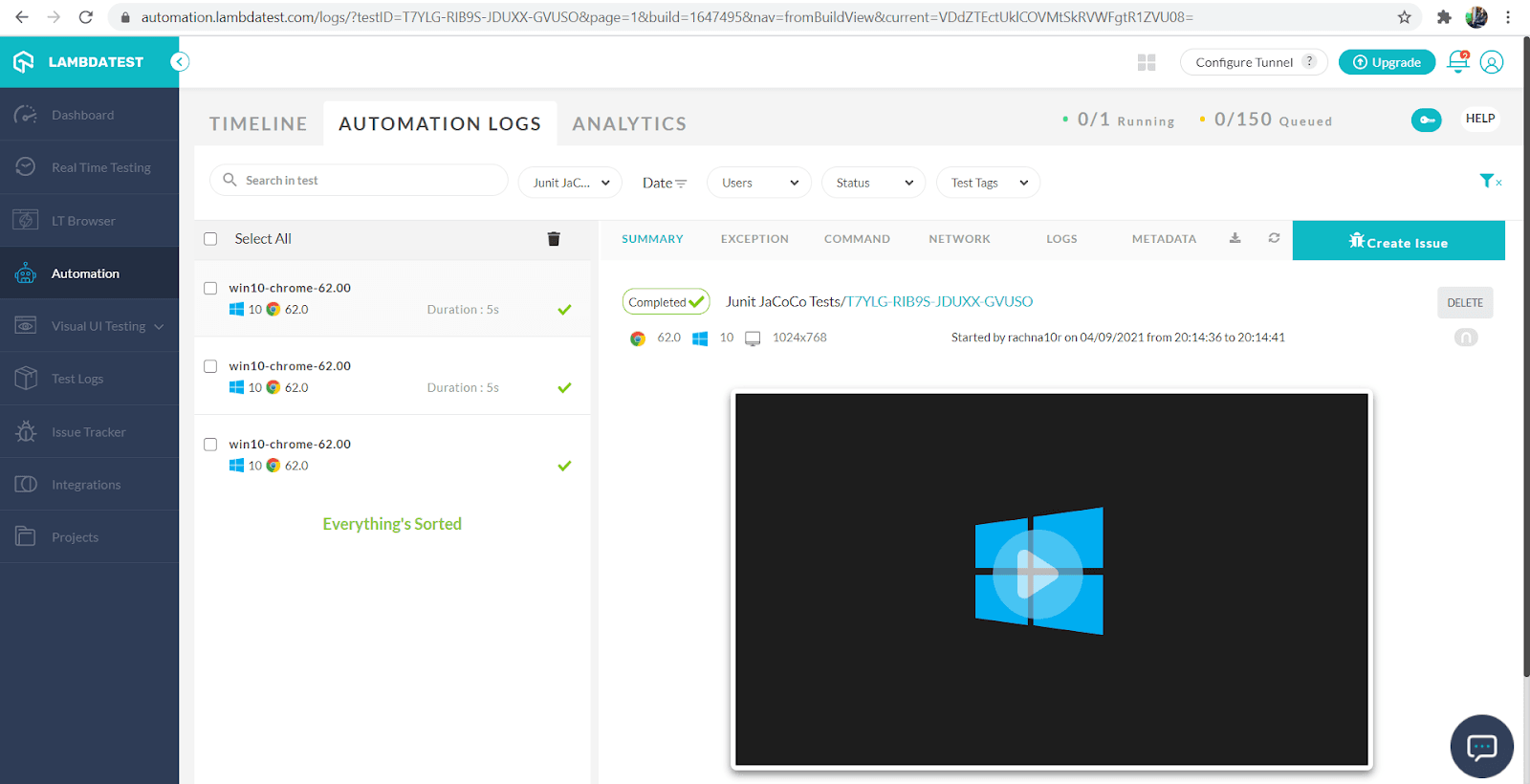
* 1. This is a simple Selenium WebDriver test that will open a sample to-do application that will do the following task:
     + Mark the first two items as mark done.
     + Add a new item to the list.
     + Return the added item.
  2. Trigger the command mvn test on the terminal to build and run the test case.



* 1. Now, login to your LambdaTest Account and go to Automation. You will find the tests you ran under the build name “JUnit JaCoCo Tests.”



* 1. Click on “JUnit JaCoCo Tests” and review them in detail. LambdaTest has recorded the video. So you can see the visuals too.



### Generating Code Coverage Report via JaCoCo Maven Plugin:

1. Now, as we have run the JUnit test cases on LambdaTest Selenium Grid, the code coverage report should be generated via the JaCoCo Maven plugin.
2. Just go to the target folder, you will find the binary format of the report as jacoco.exec.
3. You can view the HTML, CSV, and XML form of the code coverage report in the target/site/jacoco folder as index.html, jacoco.csv, and jacoco.xml, respectively.
4. Great! Now you can also try to improve the code coverage score and analyze the code coverage report generated.