# Why use Maven To Manage Selenium Dependency?

It is basically used to manage the life cycle of a project. Maven makes the build management process much easier, as you’ll only need to specify the dependencies in the pom.xml files and Maven will take care of the rest

# What Is Maven?

Maven is a software project management tool that uses the concepts of the project object model (POM). It enables the user to create an initial folder structure, perform compilation and testing and then package and deploy the final product.

# Install Maven

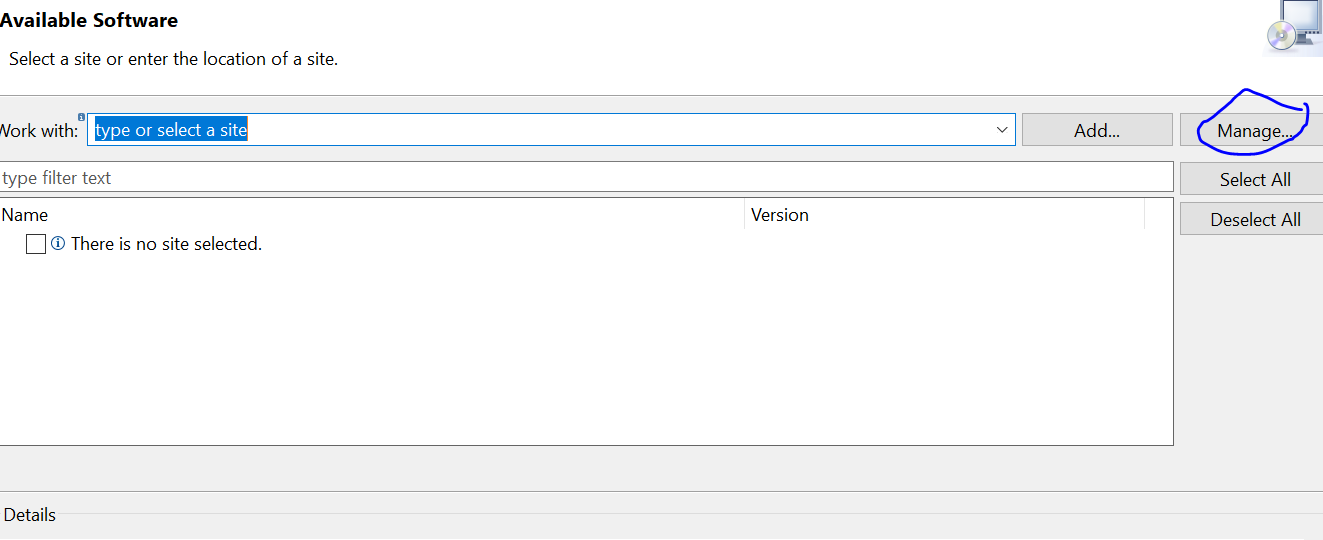
Let us use Eclipse IDE for installing Maven. If you don’t have it already use below URL

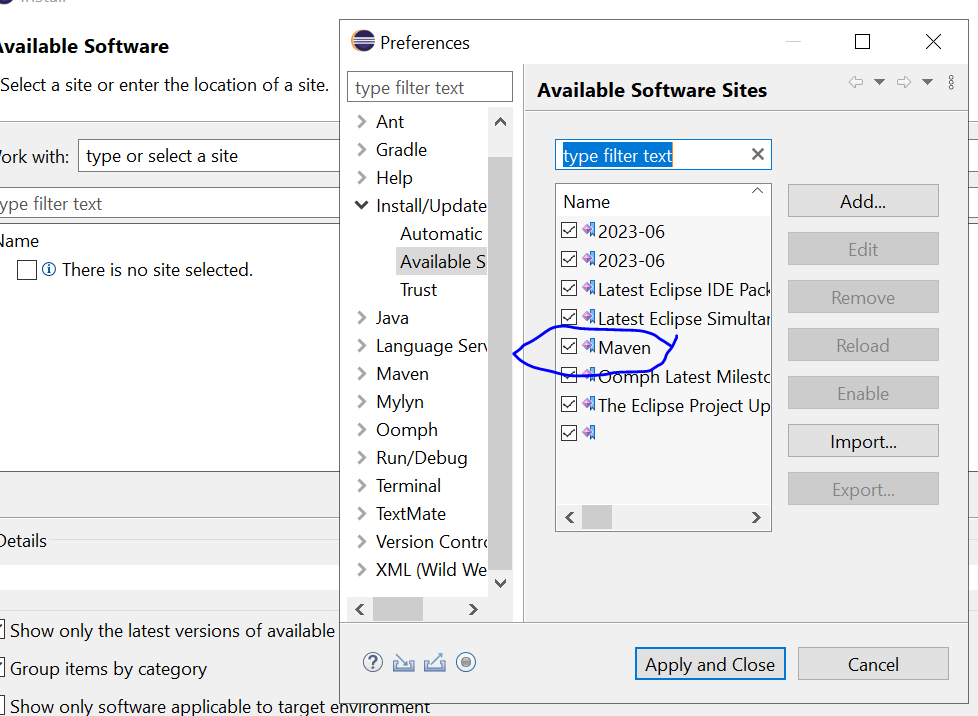
And Follow the steps mentioned in the URL

[https://www.eclipse.org/downloads/packages/installer](https://www.eclipse.org/downloads/packages/installer )

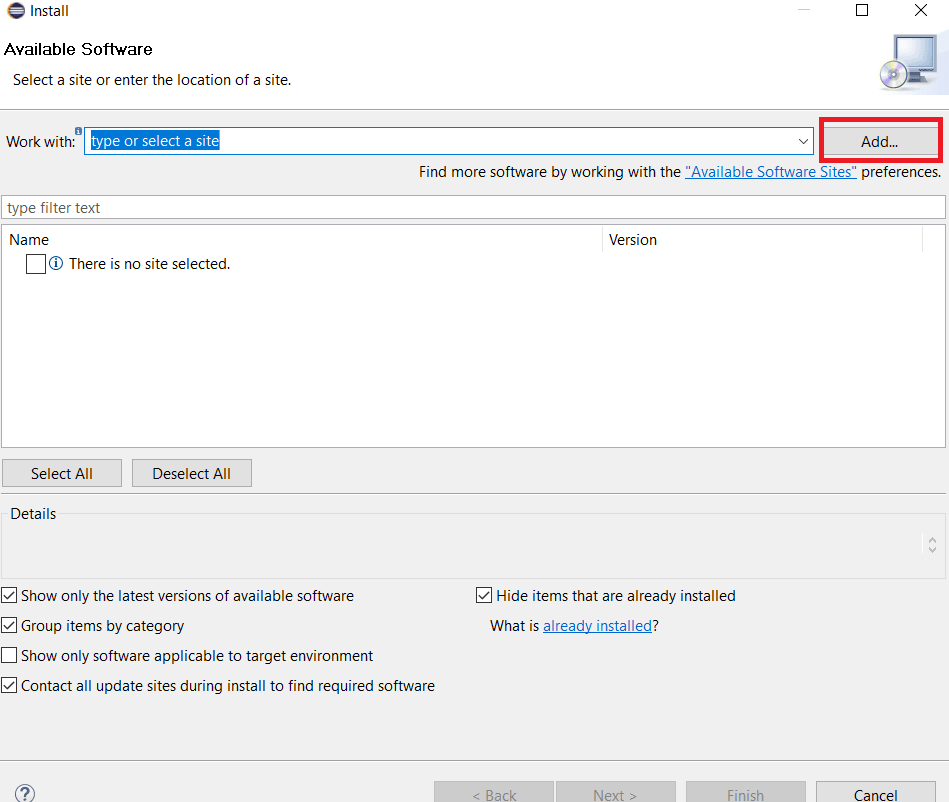
**Step 1:** Click on Help from the top menu in Eclipse and select ‘Install New Software’.

Choose Manage and verify if Maven is already installed



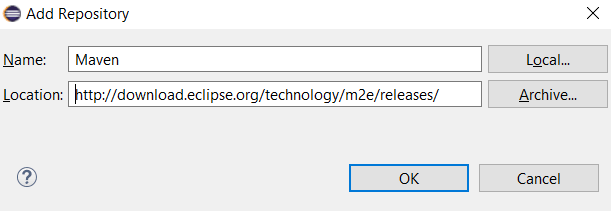


**Step 2:** On the newly opened window, click on the Add button.



**Step 3:** If it is already installed, then you can skip all the below steps. If it is not installed follow the below

In the name text box type “Maven”, and in the location text box type <http://download.eclipse.org/technology/m2e/releases/> This URL is the location from where Maven can be downloaded.



**Step 4:** A check-box will appear in the pop window, Check the check-box and click on the Next button.

**Step 5:** Wait for a few minutes for the window to complete its process.

**Step 6:** Keep the default settings and click on the Next button.

**Step 7:** Accept the Terms & Conditions and click on Finish.

**Step 8:** Wait for the installation to finish.

**Step 9:** Once the installation is finished, it will ask you to restart your eclipse. Click on Yes so as to see the changes being reflected.

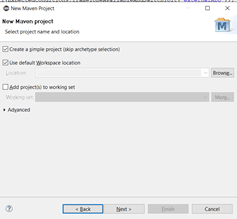
# Creating Maven Project with Eclipse IDE

Below are the steps to create a Maven Project with Eclipse IDE:

**Step 1:**Create a new project from the Eclipse IDE.

**Step 2:**From the new project window expand Maven and select Maven Project and then click on Next.

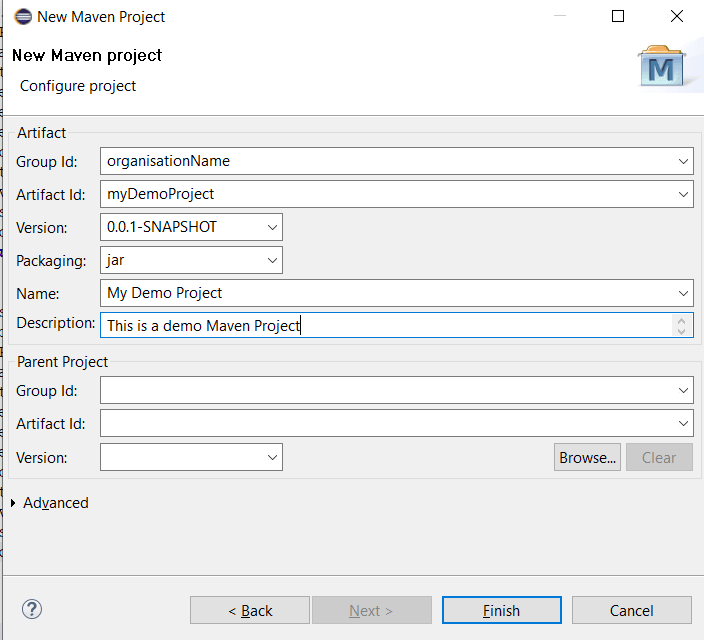
**Step 3:**You may create a simple project or just let go of this option. For now, we’ll use a simple project which would create a simple Maven-enabled Java project.



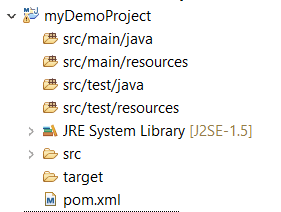
**Step 4:** Now upon clicking Next you’ll need to type information of the Maven project is created. You may refer below descriptions to fill in the values:

Group Id- corresponds to your organization’s name.  
Artifact Id- refers to the project name.

The version can be chosen flexibly. If your project does not have any parent dependencies, you don’t need to fill in the project dependencies. Just fill in the appropriate information and click on ‘Finish’.



**Step 5:** Your Maven project has now been created!



**Note:** Java code is placed in /src/main/java, resources are kept in /src/main/resources, testing code is placed in /src/test/java and the testing resources are placed in /src/test/resources.

**Step 6:** You may now open the pom.xml to view the structure set up by Maven. You’ll see all the information that we entered in ‘step 4’ here. You can use the tabs at the bottom to change the view. The pom.xml tab has the pom XML code for the Maven project.

# Selenium Automation

## Adding Dependencies to POM.xml

 We’ll write the dependencies for the project in pom.xml, and Maven will directly download those from its repository. This saves the trouble of doing it manually and reducing the chance of missing out on adding some of the jars

## Selenium Maven Dependency For Your Automation Project

All the external libraries that are used in a project are called **dependencies**. Maven has an excellent feature that automatically downloads required libraries from its central repository, which makes it easy as you don’t have to store them locally. Below is an example of writing a Selenium Maven dependency in your pom.xml:

<dependency>

<groupId>org.seleniumhq.selenium</groupId>

<artifactId>selenium-java</artifactId>

<version>4.10.0</version>

</dependency>

## TestNG Selenium Maven Dependency:

This would import the testing framework dependency for Java.

<dependency>

<groupId>org.testng</groupId>

<artifactId>testng</artifactId>

<version>6.9.10</version>

<scope>test</scope>

</dependency>

## Create TestClass

create the sample test class under the src/test/java folder of our project structure.

import org.openqa.selenium.WebDriver;

import org.testng.annotations.Test;

public class Test1 {

WebDriver mydriver;

@Test

public void Setup()

{

System.out.println("first maven execution");

}

}

# Maven Lifecycle In Selenium

## Goals

Goals required for the Selenium project.

1. validate: would check if our project is correct and all the required information available
2. compile: would compile the project source code
3. test: would unit test the compiled source code of our project
4. package: would package the compiled code into the distributable formats, like JAR
5. integration-test: would deploy the package into an environment where we would run the integration tests
6. verify: would verify if the package is valid
7. install: would locally install the package
8. deploy: would be used in integration or release environment by copying the final project into a remote repository where it can be accessed by other projects or developers
9. clean: cleans up previous build artifacts
10. site: creates site documentation for the project

Of the above-said default goals, three are crucial for Selenium test automation. There are clean, install, and tests.

**Clean-** It would clean the target folder, i.e. the folder where the previous build’s libraries, build files(war, tar or jar files), reports, output files, etc are saved. On executing mvn -clean this target folder will be deleted.

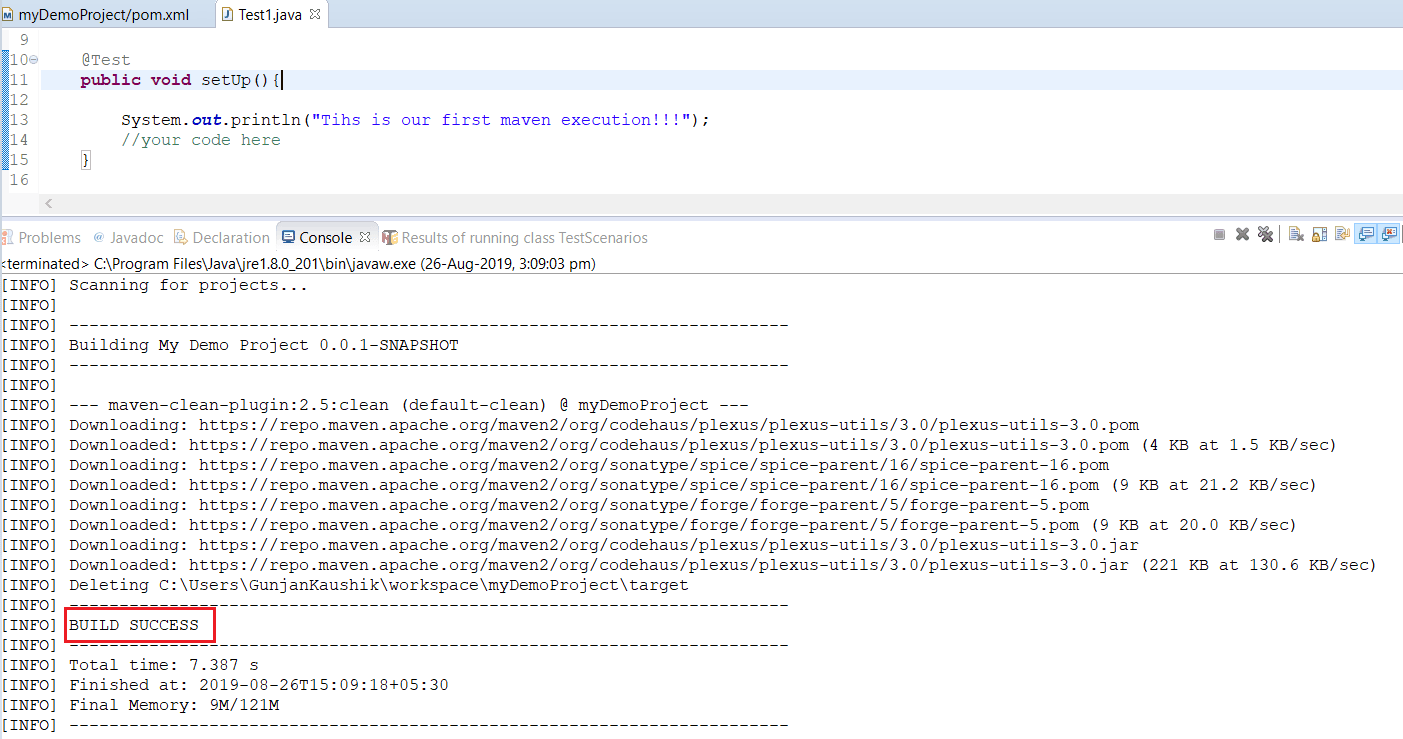
**Install-** It would install all the dependencies, in case of missing jars, and create the deployment file(war/jar files in case of JAVA) and then it’ll run the test.

**Test-** It will simply run the test without creating any deployment file.

When using Eclipse IDE, you can directly use any of these three goals by **right-clicking on your pom.xml, then Run As and selecting any of the options.**

## Maven Clean

We’ll start with selecting Maven **clean** in this Selenium Maven tutorial, you can see the output below:



So here the task of clean, which is to delete the target folder has been successfully completed and hence our Build is successful.

## Maven Install

Before going to the second task of **install**, you need to add a plugin called Maven Compiler plugin.Without it the test automation build will fail. This plugin is used to identify the specific location of the compiler. You can just add the below plugin in your pom.xml under build/pluginManagement/plugins and refresh the project before executing Maven install.

<build>

<pluginManagement>

<plugins>

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-compiler-plugin</artifactId>

<version>3.11.0</version>

<configuration>

<source>1.8</source>

<target>1.8</target>

</configuration>

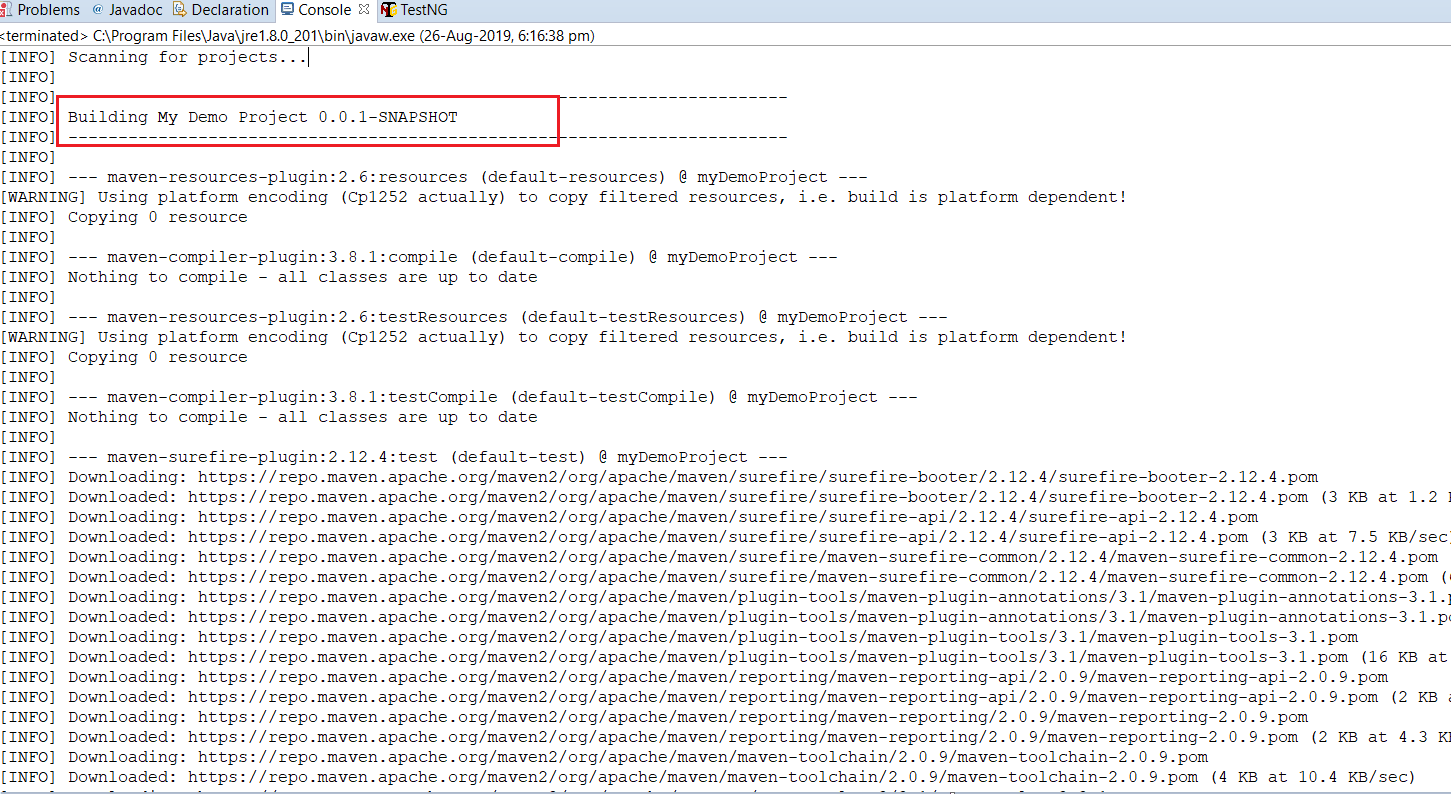
</plugin>

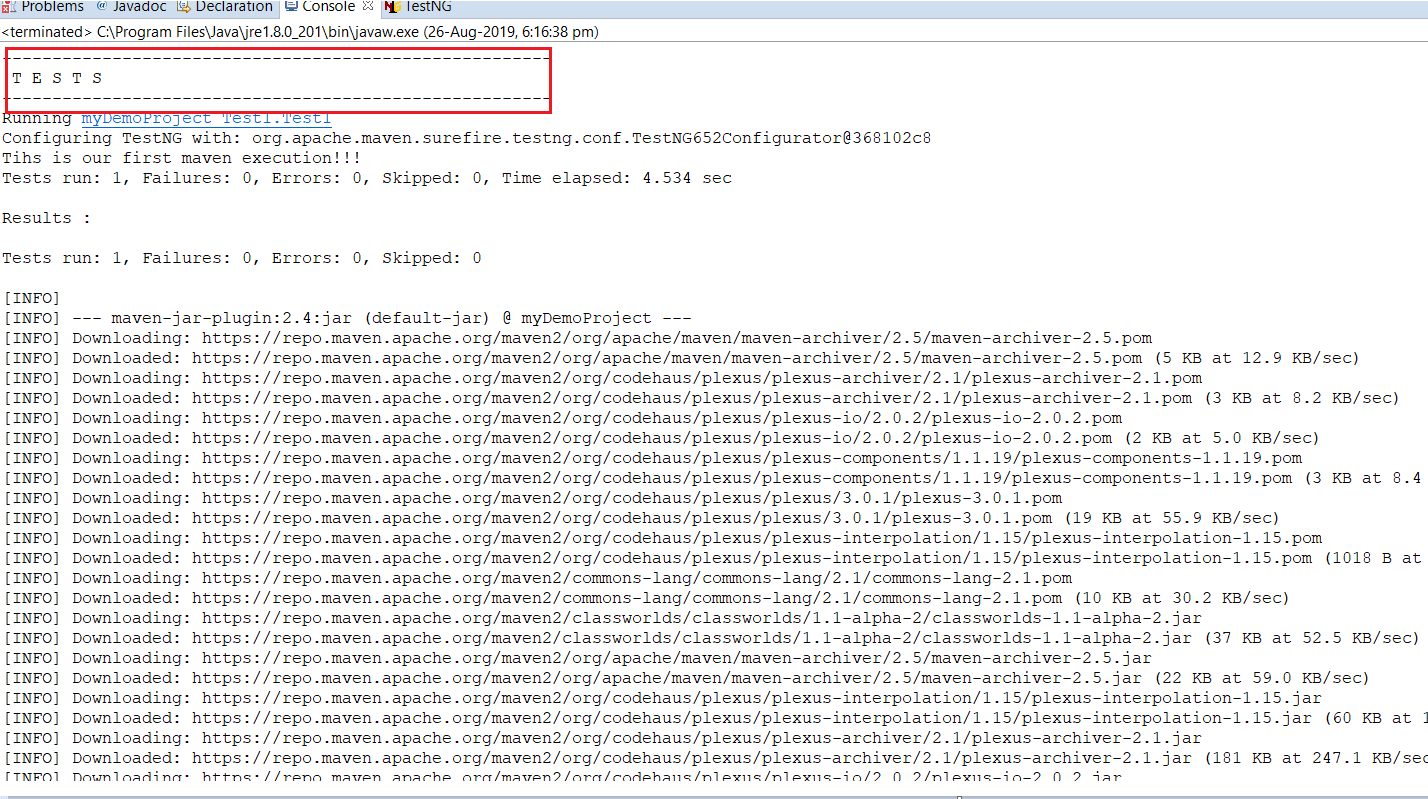
</plugins>

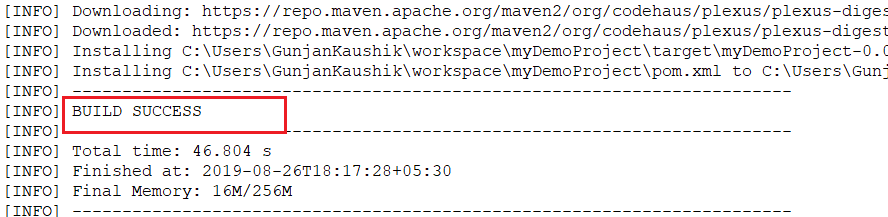
</pluginManagement>

</build>

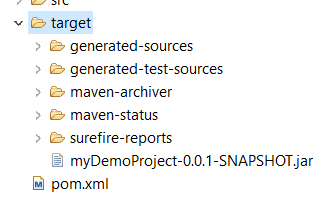
Now, go to Maven Install just like you did for Maven Clean(by **right-clicking on your pom.xml, then Run As and selecting Maven Install)** in this Selenium Maven tutorial and observe the console output for the build installation:





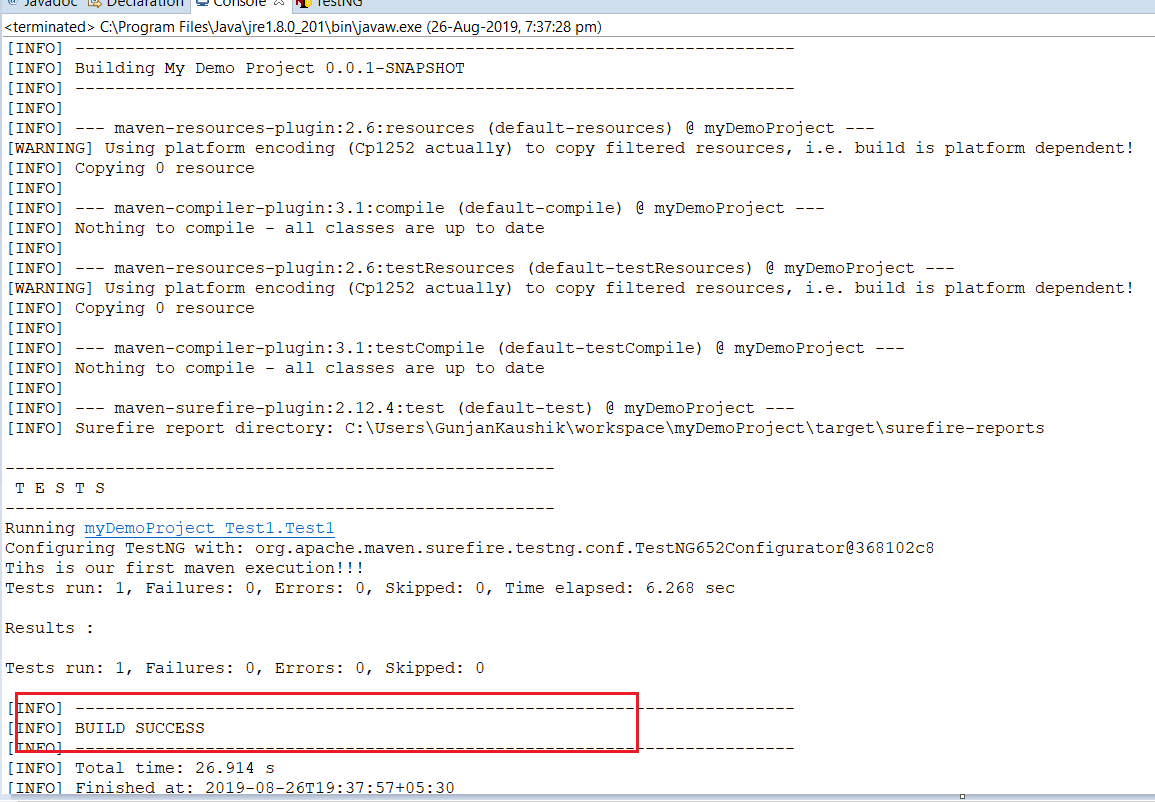


You can see in the console output that the Maven installer executed the tests as well. To see the installed directories in your local system you refresh your project and see the directories generated. In the below snapshot you can see all the files generated(a jar file as well since this is a simple JAVA program) as a result of Maven install.



## Maven Test

Similarly, we can do **Maven Test** (by **right-clicking on your pom.xml, then Run As and selecting Maven Test)**in this Selenium Maven tutorial and see the build results in the console:



# Maven Surefire Plugin

The surefire plugin helps Maven to identify the tests and is used with whichever framework your project is built on. To add Surefire plugin to your pom.xml use below code snippet:

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-surefire-plugin</artifactId>

<version>3.1.2</version>

<configuration>

<suiteXmlFiles>

<suiteXmlFile>src/test/resources/TestFile.xml</suiteXmlFile>

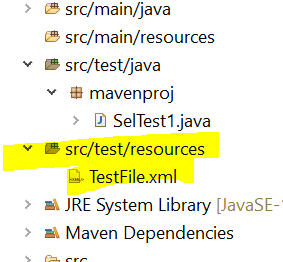
</suiteXmlFiles>

<testFailureIgnore>true</testFailureIgnore>

</configuration>

</plugin>

In this case we are specifying the TestFile.xml is placed at src/test/resources



TestFile.xml ( Here mavenproj is package name and Test1 is className)

<?xml version="1.0" encoding="UTF-8" standalone="no"?>

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

<suite name="myTestSuite">

<test name="myTest">

<classes>

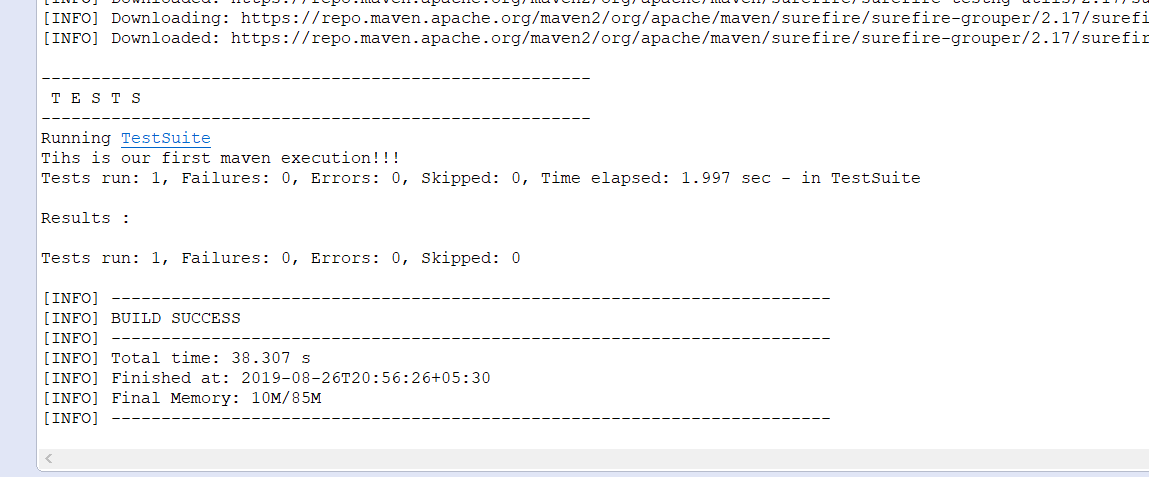
<class name="Test1" />

</classes>

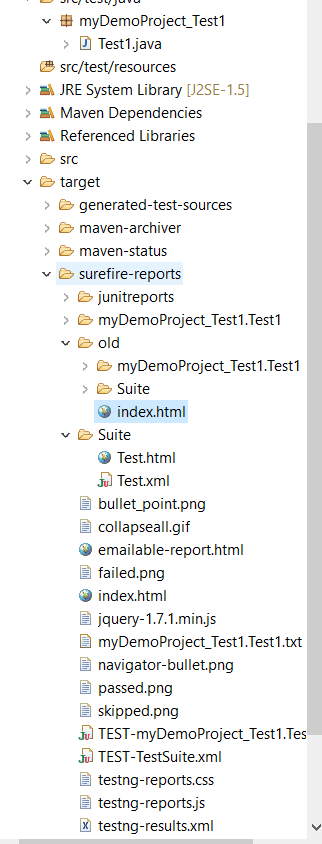
</test>

</suite>

Now I’ll run Maven test from eclipse in this Selenium Maven tutorial and see the results:



You can now check the reports that have been generated by default



# Sample POM.XML

Below is the sample POM.xml

<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<groupId>mymavenorg</groupId>

<artifactId>mavenproj</artifactId>

<version>0.0.1-SNAPSHOT</version>

<name>maventest</name>

<dependencies>

<dependency>

<groupId>org.seleniumhq.selenium</groupId>

<artifactId>selenium-java</artifactId>

<version>4.10.0</version>

</dependency>

<dependency>

<groupId>org.testng</groupId>

<artifactId>testng</artifactId>

<version>6.9.10</version>

<scope>test</scope>

</dependency>

</dependencies>

<build>

<pluginManagement>

<plugins>

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-compiler-plugin</artifactId>

<version>3.11.0</version>

<configuration>

<source>1.8</source>

<target>1.8</target>

</configuration>

</plugin>

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-surefire-plugin</artifactId>

<version>3.1.2</version>

<configuration>

<suiteXmlFiles>

<suiteXmlFile>src/test/resources/TestFile.xml</suiteXmlFile>

</suiteXmlFiles>

<testFailureIgnore>true</testFailureIgnore>

</configuration>

</plugin>

</plugins>

</pluginManagement>

</build>

</project>