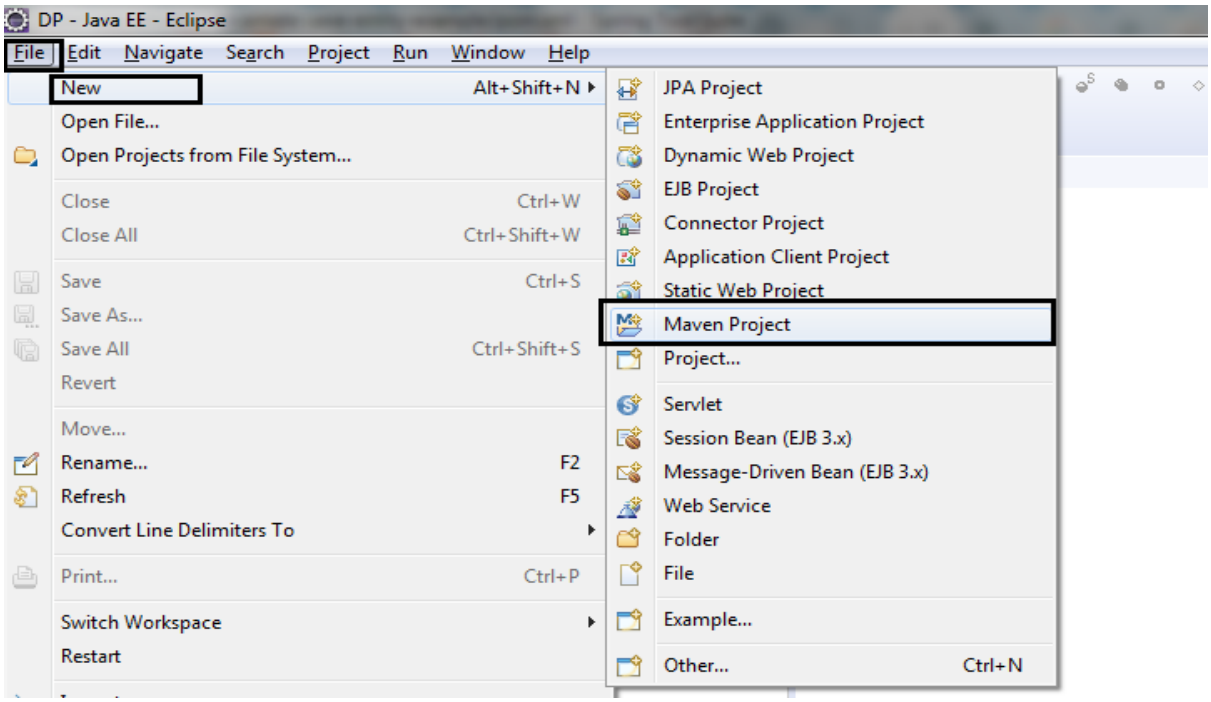
Q13) Integrate Selenium with Maven

a. Maven integration for Eclipse is already installed in your Practice lab. Refer to QA to QE lab guide for Phase 2 for more information.

Step 1.14.2: Creating a Maven project

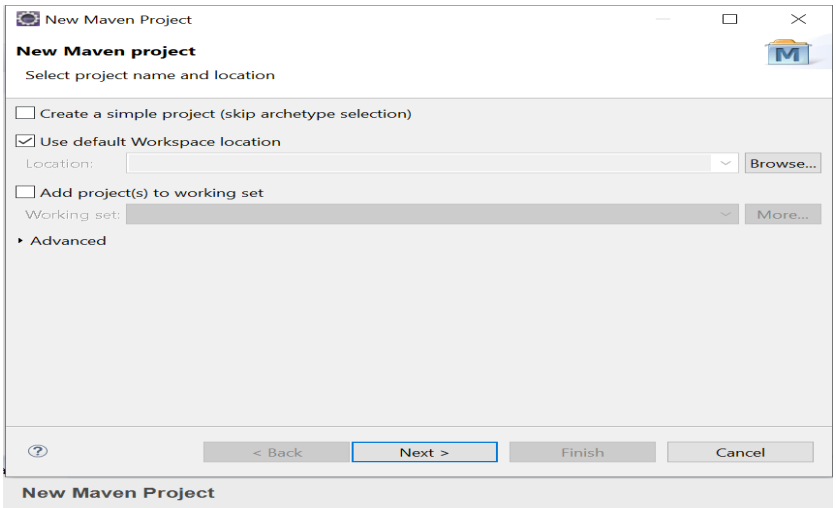
a. Open Eclipse.

b. Click on File->New->Maven project.

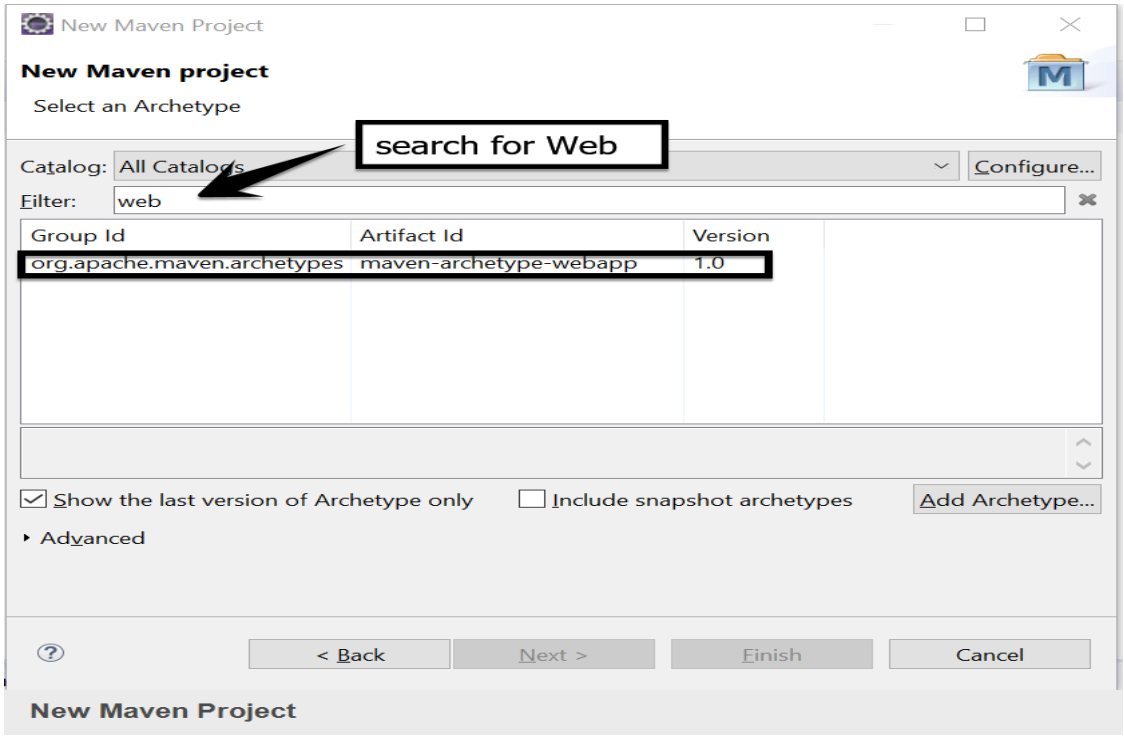


c. Select the default workspace location.

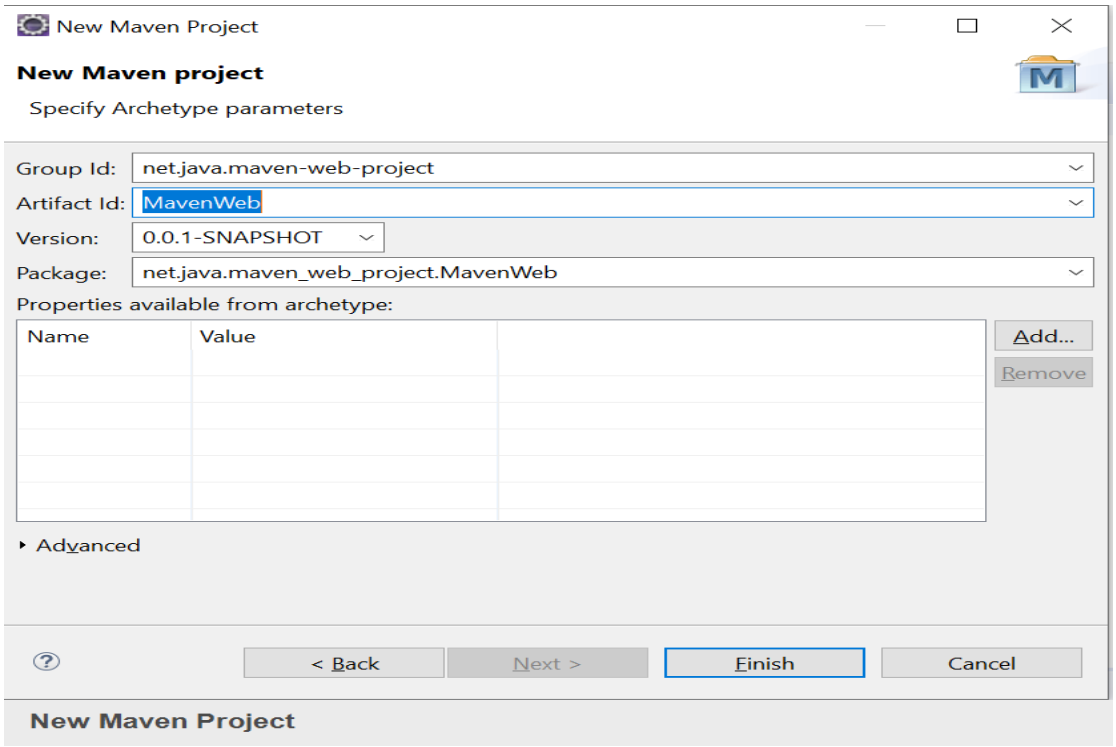
Click on **Next**.



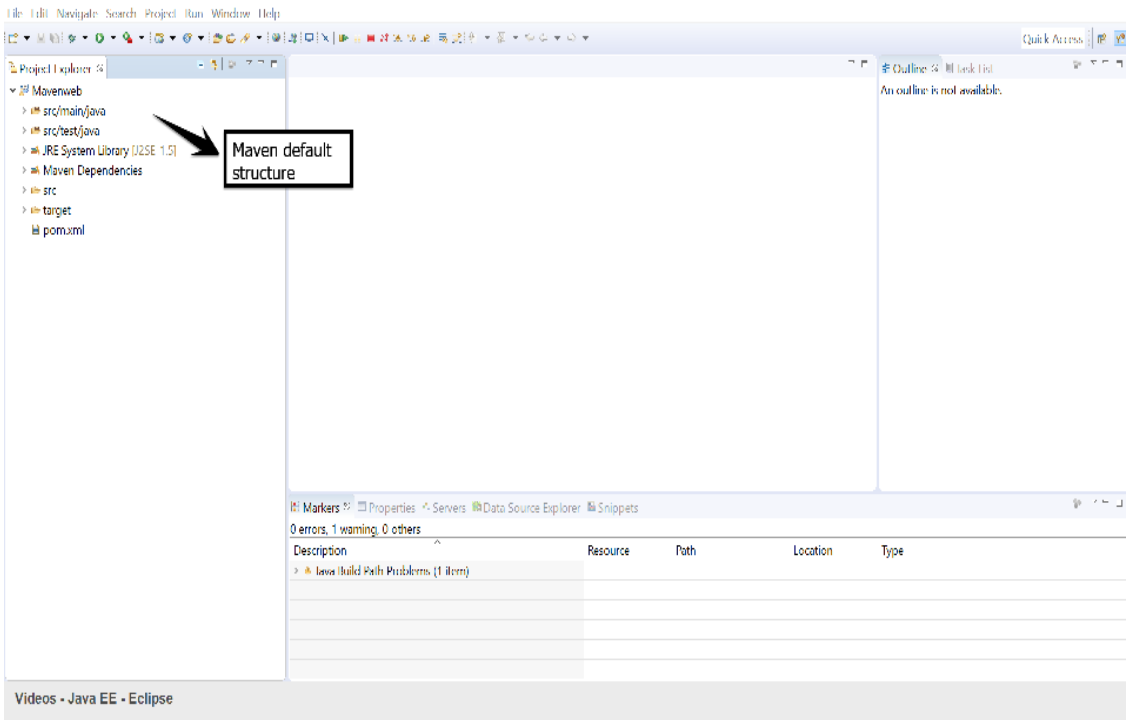
d. Select the Maven archetype as maven-archetype-webapps and click on **Next**.



e. Enter the Group ID and Artifact ID and click on **Finish**.



f. You are all set. The new Maven structure will look like the below screenshot:

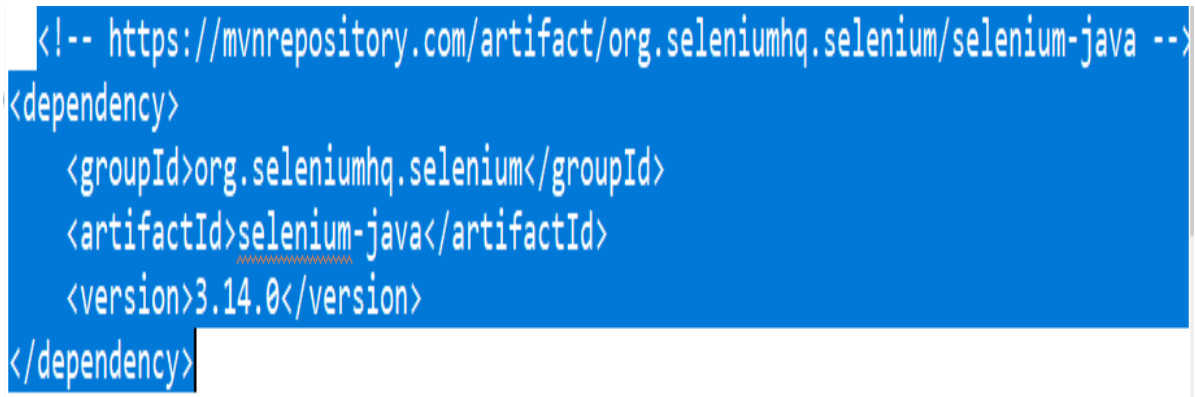


Step 1.14.3: Adding Maven Dependency

- If we are using the Selenium web driver, then we need to specify the Selenium web driver dependencies to our pom.xml.

    a. Go to Maven Repository: https://mvnrepository.com/artifact/org.seleniumhq.selenium/selenium-java/3.14.0
    b. Copy the dependency.

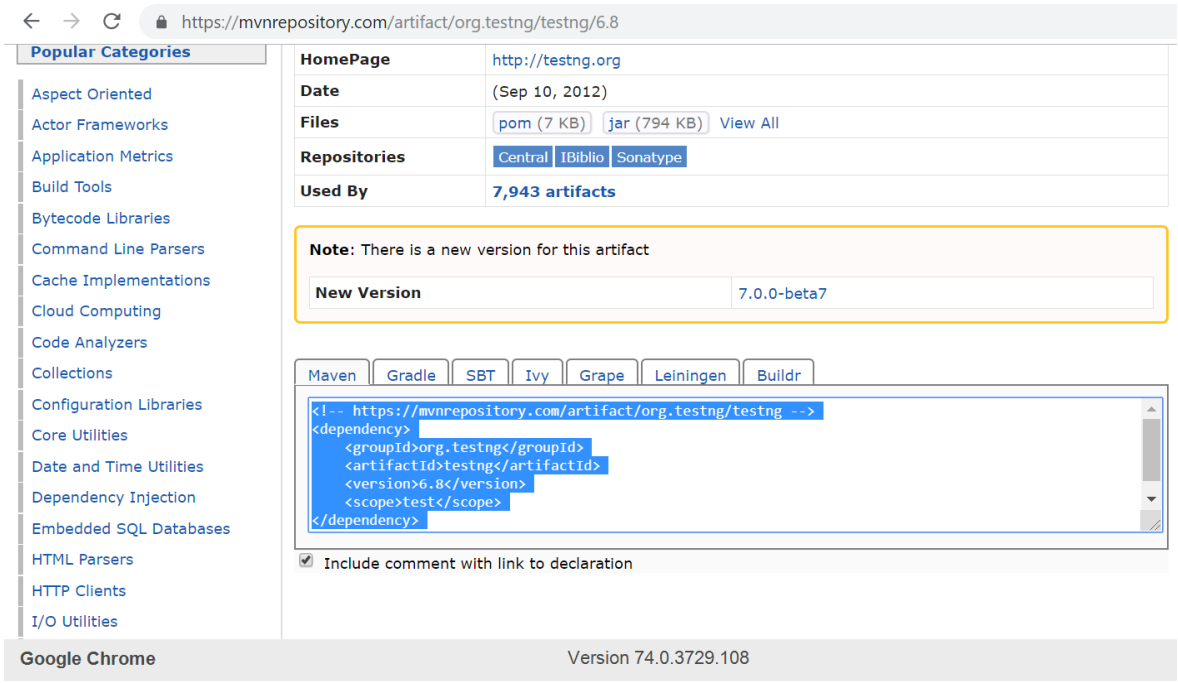
c. Go to Eclipse.
d. Open the Pom.xml file from the project.
e. Paste the dependency in the Pom.xml file:



- If we are using TestNG, then we need to specify the TestNG dependencies to our pom.xml

a. Go to the Maven Repository: https://mvnrepository.com/artifact/org.testng/testng/6.8

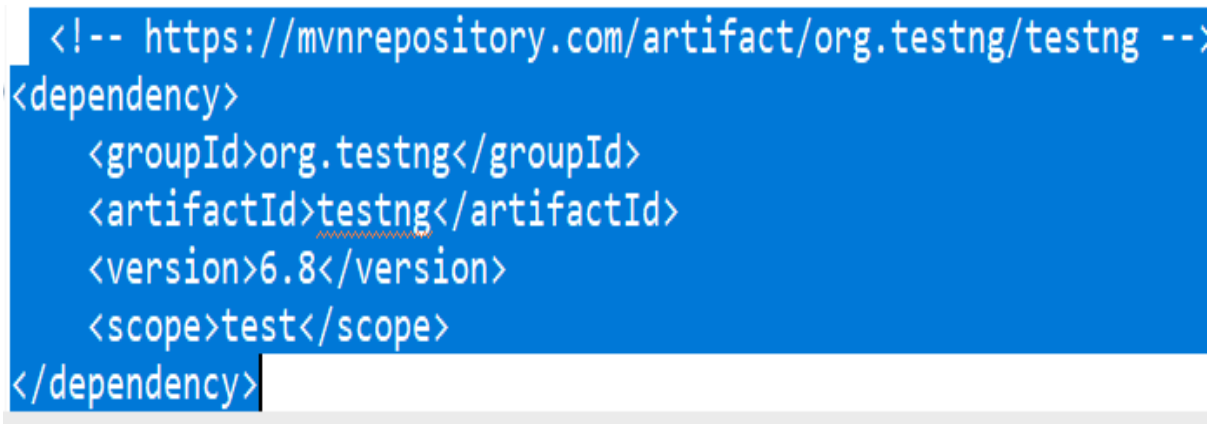
b. Copy the dependency.



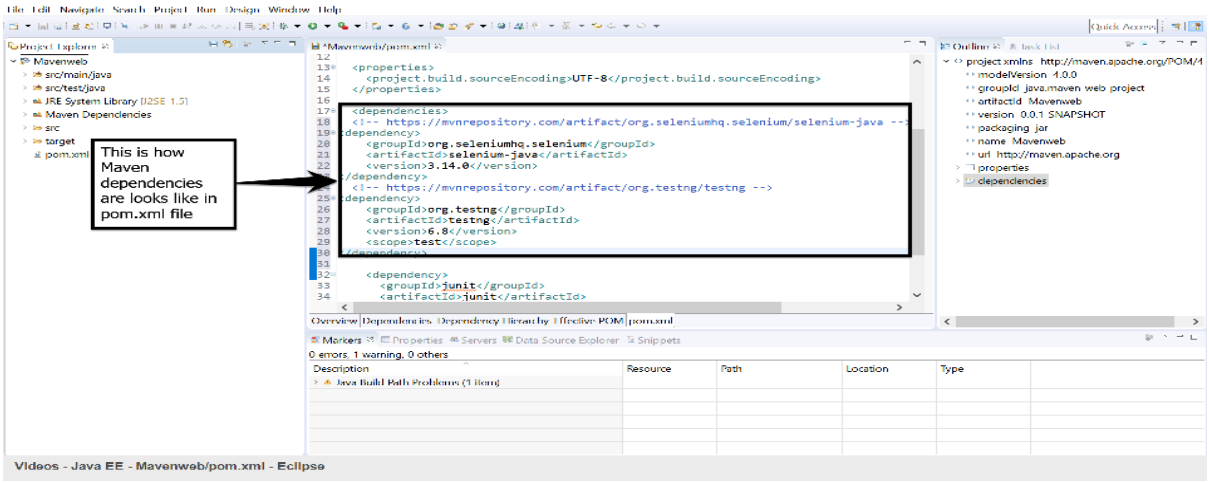
c. Go to Eclipse.

d. Open the Pom.xml file from the project.

e. Paste the dependency in the Pom.xml file:



f. This is what the Maven dependencies will look like in the pom.xml file.



Q14) Demonstrate integration of Selenium with Ant.

Writing a code for build.xml

build.xml is the most important component of the Ant build tool. For a Java project, all learning, setup, compilation and deployment related tasks are mentioned in this file in an XML format. When we execute this XML file using a command line or any IDE plugin, all instructions written into this file will get executed in a sequential manner.

Let's understand the code within a sample build.xml.

- Project tag is used to mention a project name and basedir attribute. The basedir is the root directory of an application.

```
<project name="YTMonetize" basedir=".">
```

- Property tags are used as variables in the build.xml file to be used in further steps.

```
<property name="build.dir" value="${basedir}/build"/>
                <property name="external.jars" value=".\resources"/>
    <property name="ytoperation.dir" value="${external.jars}/YTOperation"/>
<property name="src.dir"value="${basedir}/src"/>
```

- Target tags are used as steps that will execute in a sequential order. The Name attribute is the name of the target. You can have multiple targets in a single build.xml.

```
<target name="setClassPath">
```

- **path** tag is used to bundle all the files logically which are in the common location.

```
<path id="classpath_jars">
```

- **pathelement** tag will set the path to the root of the common location where all the files are stored.

```
<pathelement path="${basedir}/"/>
```

- **pathconvert** tag is used to convert paths of all the common file inside the path tag to the system's classpath format.

```
<pathconvert pathsep=";" property="test.classpath" refid="classpath_jars"/>
```

- **fileset** tag is used to set the classpath for different third-party jars in our project.

```
<fileset dir="${ytoperation.dir}" includes="*.jar"/>
```

- **Echo** tag is used to print the text on the console.

```
<echo message="deleting existing build directory"/>
```

- **Delete** tag will clean the data from the given folder.

```
<delete dir="${build.dir}"/>
```

- **mkdir** tag will create a new directory.

```
<mkdir dir="${build.dir}"/>
```

- **javac** tag is used to compile the java source code and move the .class files to a new folder.

```
<javac destdir="${build.dir}" srcdir="${src.dir}">
    <classpath refid="classpath_jars"/>
</jalvac>
```

- **jar** tag will create a jar file from the .class files.

```
<jar destfile="${ytoperation.dir}/YTOperation.jar" basedir="${build.dir}">
```

- **manifest** tag will set your main class for execution.

```
<manifest>
        <attribute name="Main-Class" value="test.Main"/>
</manifest>
```

- 'depends' attribute is used to make a target dependent on another target.

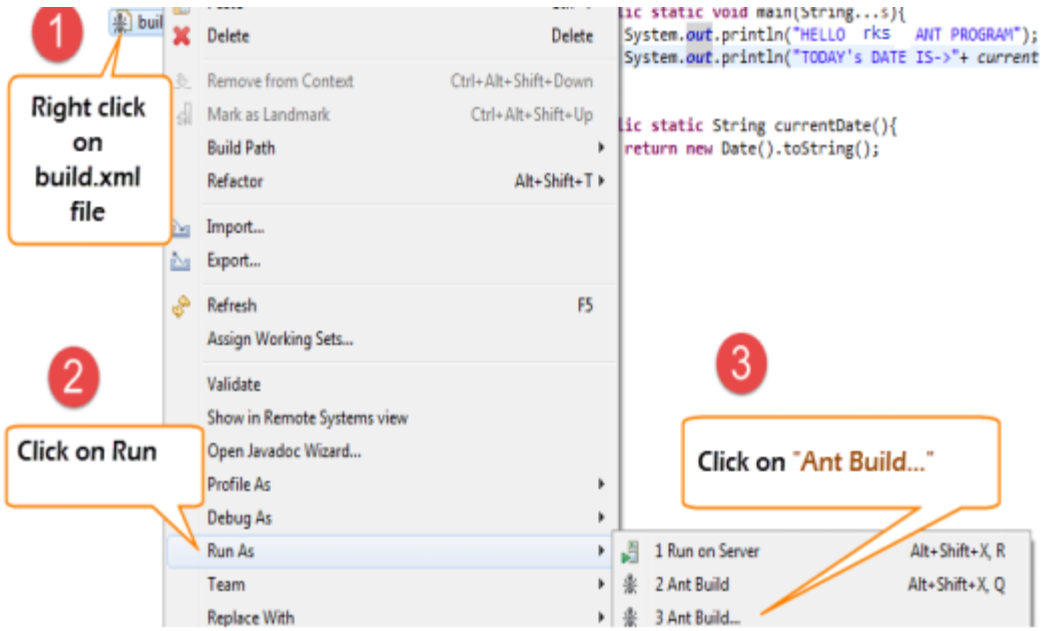
```
<target name="run" depends="compile">
```

- **java** tag will execute the main function from the jar created in the compile target section.

```
<java jar="${ytoperation.dir}/YTOperation.jar" fork="true"/>
```

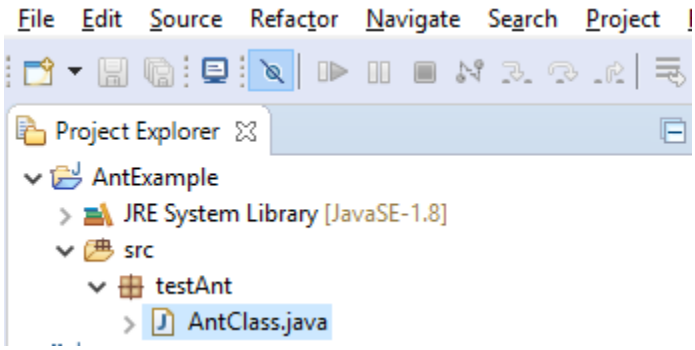
Steps 1.15.3: Running Ant using Eclipse plugin

- To run Ant from Eclipse, go to build.xml file -> right click on the file -> Run as… -> Build file.



Steps 1.15.4: Writing a code to implement the functionality of Ant

- We will take a small sample program that will explain the Ant functionality very clearly. Our project structure will look something like –



- Here in this example, we have 4 targets:

1. Set the classpath for external jars.
2. Clean the previously compiled code.
3. Compile the existing java code.
4. Run the code.

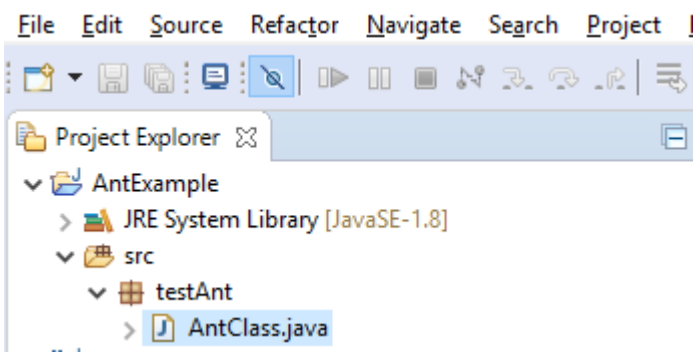
```
AntClass.class

package testAnt;
import java.util.Date;


public class AntClass {
  public static void main(String...s){
                System.out.println("HELLO ANT PROGRAM");
                System.out.println("TODAY's DATE IS->"+ currentDate() );
}
public static String currentDate(){
     return new Date().toString();
     }
}
```

Steps 1.25.5: Writing a build.xml file

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

<!--Project tag used to mention the project name, and basedir attribute will be the root directory of the application-->


<project name="YTMonetize" basedir=".">

    <!--Property tags will be used as variables in build.xml file to use in further steps-->


      <property name="build.dir" value="${basedir}/build"/>

   <property name="external.jars" value=".\resources"/>

      <property name="ytoperation.dir" value="${external.jars}/YTOperation"/>

<property name="src.dir"value="${basedir}/src"/>

<!--Target tags used as steps that will execute in sequential order. name attribute will be the name  of the target and < a name=OLE_LINK1 >'depends' attribute used to make one target to depend on another target -->

            <target name="setClassPath">

                              <path id="classpath_jars">

                                        <pathelement        path="${basedir}/"/>

                              </path>

<pathconvert pathsep=";"property="test.classpath" refid="classpath_jars"/>

</target>

      <target name="clean">

                  <!--echo tag will use to print text on console-->

      <echo message="deleting existing build directory"/>

       <!--delete tag will clean data from given folder-->

                  <delete dir="${build.dir}"/>

                  </target>

      <target name="compile" depends="clean,setClassPath">

      <echo message="classpath:${test.classpath}"/>

                              <echo message="compiling....."/>

      <!--mkdir tag will create new director-->

      <mkdir dir="${build.dir}"/>
```

```
                    <echo message="classpath:${test.classpath}"/>
                    <echo message="compiling....."/>
 <!--javac tag used to compile java source code and move .class files to a new folder-->
<javac destdir="${build.dir}" srcdir="${src.dir}">
                                <classpath refid="classpath_jars"/>
            </javac>
            <!--jar tag will create jar file from .class files-->
            <jar destfile="${ytoperation.dir}/YTOperation.jar"basedir="${build.dir}">
                        <!--manifest tag will set your main class for execution-->
            <manifest>
            <attribute name="Main-Class" value="testAnt. AntClass"/>
            </manifest>
            </jar>
             </target>
            <target name="run" depends="compile">
<!--java tag will execute main function from the jar created in compile target section-->
<java jar="${ytoperation.dir}/YTOperation.jar"fork="true"/>
</target>
            </project>
```

Steps 1.15.6: Executing the TestNG code using Ant



- Here we will create a class with the TestNG methods and set the classpath for Testing in build.xml.
- Now to execute the TestNG method, we will create another testng.xml file and call this file from the build.xml file.

Step 1) We create an "AntClass.class" in package testAnt.

```
AntClass.class

package testAnt;

import java.util.Date;

import org.testng.annotations.Test;

public class AntClass {

   @Test

        public void AntTestNGMethod(){

                System.out.println("HELLO ANT PROGRAM");

                System.out.println("TODAY's DATE IS->"+ currentDate()
 );

        }

        public static String currentDate(){

                return new Date().toString();

        }

}
```

Step 2) Create a target to load this class in build.xml.

```
<!-- Load testNG and add to the class path of application -->

        <target name="loadTestNG" depends="setClassPath">

<!—using taskdef  tag we can add a task to run on the current project. In below  line, we are adding testing task in this project. Using testing task here now we    can run testing code using the ant script -->

        <taskdef resource="testngtasks"
classpath="${test.classpath}"/>

</target>
```

Step 3) Create testng.xml

```
testng.xml

<?xml version="1.0"encoding="UTF-8"?>
<!DOCTYPE suite SYSTEM "http://testng.org/testng-1.0.dtd">
<suite name="YT"thread-count="1">
                                <test name="TestNGAnt">
                                <classes>
                                  <class name="testAnt. AntClass">
        </class>
</classes>
</test>
</suite>
```

Step 4) Create a Target in build.xml to run this TestNG code.

```
<target name="runTestNGAnt" depends="compile">
<!-- testng tag will be used to execute testng code using corresponding testng.xml    file. Here classpath attribute is setting classpath for testng's jar to the project-->
            <testng classpath="${test.classpath};${build.dir}">
<!—xmlfileset tag is used here to run testng's code using testing.xml file. Using
includes tag we are mentioning path to testing.xml file-->
             <xmlfileset dir="${basedir}" includes="testng.xml"/>
</testng>
```

Step 5) The complete build.xml will look like:

```
<?xml version="1.0"encoding="UTF-8"standalone="no"?>
<!--Project tag used to mention the project name, and basedir attribute will be the root directory of the application-->
                        <project name="YTMonetize" basedir=".">
<!--Property tags will be used as variables in build.xml file to use in further steps-->
```

```
                        <property name="build.dir"value="${basedir}/build"/>
<!-- put  testng related jar in the resource  folder -->
                <property name="external.jars" value=".\resource"/>
                        <property name="src.dir" value="${basedir}/src"/>
<!--Target tags used as steps that will execute in sequential order. Name attribute will be the name of the target and 'depends' attribute used to make one target to depend on another target-->
<!-- Load testNG and add to the class path of application -->
       <target name="loadTestNG"depends="setClassPath">
            <taskdef resource="testngtasks"classpath="${test.classpath}"/>
                        </target>
                        <target name="setClassPath">
                             <path id="classpath_jars">
                        <pathelement path="${basedir}/"/>
                                    <fileset dir="${external.jars}"
includes="*.jar"/>
        </path>
       <pathconvert
pathsep=";"property="test.classpath"refid="classpath_jars"/>
             </target>
             <target name="clean">
           <!--echo tag will use to print text on console-->
                        <echo message="deleting existing build directory"/>
           <!--delete tag will clean data from given folder-->
                        <delete dir="${build.dir}"/>
                         </target>
<target name="compile"depends="clean,setClassPath,loadTestNG">
                    <echo message="classpath:${test.classpath}"/>
                        <echo message="compiling....."/>
                              <!--mkdir tag will create new director-->
                               <mkdir dir="${build.dir}"/>
                        <echo message="classpath:${test.classpath}"/>
```

```
                                <echo message="compiling....."/>
<!--javac tag used to compile java source code and move .class files to a new folder-->
                    <javac destdir="${build.dir}"srcdir="${src.dir}">
                        <classpath refid="classpath_jars"/>
                          </javac>
 </target>
<target name="runTestNGAnt"depends="compile">
<!-- testng tag will be used to execute testng code using corresponding testng.xml file -->
              <testng classpath="${test.classpath};${build.dir}">
             <xmlfileset dir="${basedir}"includes="testng.xml"/>
              </testng>
</target></project>
```

Steps 1.15.7: Integrating Ant with Selenium WebDriver

- We have learned that by using Ant we can put all the third party jars in a particular location in the system and set their path for our project.
- Using this method, we are setting all the dependencies of our project in a single place and making it more reliable for compilation, execution, and deployment.
- Similarly, for our testing projects, using Selenium, we can easily mention Selenium dependency in build.xml and we don't need to add a classpath for it manually in our application.
- So, now you can ignore the below-mentioned traditional way to set classpaths for our project.

Steps 1.15.8: Modifying previous examples

We are going to modify the previous example now.

Step 1) Set the property Selenium.jars to the Selenium related jar in the resource folder.

```
<property name="selenium.jars" value=".\selenium"/>
```

Step 2) In the target **setClassPath**, add the Selenium files.

```
<target name="setClassPath">
                <path id="classpath_jars">
```

```
                    <pathelement path="${basedir}/"/>
                    <fileset dir="${external.jars}" includes="*.jar"/>
                <!-- selenium jar added here -->
                        <fileset dir="${selenium.jars}" includes="*.jar"/>
        </path>
```

Step 3) Complete build.xml:

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


<!--Project tag used to mention the project name, and basedir attribute will be the root directory of the application-->
                            <project name="YTMonetize" basedir=".">
            <!--Property tags will be used as variables in build.xml file to use in further steps-->
        <property name="build.dir" value="${basedir}/build"/>
     <!-- put  testng related jar in the resource  folder -->
            <property name="external.jars" value=".\resource"/>
<!-- put  selenium related jar in resource  folder -->
    <property name="selenium.jars" value=".\selenium"/>
        <property name="src.dir" value="${basedir}/src"/>
<!--Target tags used as steps that will execute in sequential order. name attribute will be the name of the target and 'depends' attribute used to make one target to depend on another target-->
     <!-- Load testNG and add to the class path of application -->
      <target name="loadTestNG" depends="setClassPath">
        <taskdef resource="testngtasks" classpath="${test.classpath}"/>
                    </target>
<target name="setClassPath">
            <path id="classpath_jars">
                                        <pathelement path="${basedir}/"/>
        <fileset dir="${external.jars}" includes="*.jar"/>
                            <!-- selenium jar added here -->
```

```
                <fileset dir="${selenium.jars}"includes="*.jar"/>
        </path>
    <pathconvert pathsep=";" property="test.classpath"
refid="classpath_jars"/>
</target>
<target name="clean">
<!--echo tag will use to print text on console-->
            <echo message="deleting existing build directory"/>
                    <!--delete tag will clean data from given folder-->
                            <delete dir="${build.dir}"/>
                                    </target>
<target name="compile" depends="clean,setClassPath,loadTestNG">
        <echo message="classpath:${test.classpath}"/>
            <echo message="compiling....."/>
        <!--mkdir tag will create new director-->
              <mkdir dir="${build.dir}"/>
                    <echo message="classpath:${test.classpath}"/>
                    <echo message="compiling....."/>
<!--javac tag used to compile java source code and move .class files to
new folder-->
    <javac destdir="${build.dir}"srcdir="${src.dir}">
          <classpath refid="classpath_jars"/>
        </javac>
        </target>
        <target name="runTestNGAnt" depends="compile">
<!-- testng tag will be used to execute testng code using
corresponding testng.xml file -->
                                <testng
classpath="${test.classpath};${build.dir}">
            <xmlfileset dir="${basedir}" includes="testng.xml"/>
                    </testng>
        </target>
```

```
</project>
```

Step 4) Now change the previously created class **AntClass.java** with the new code.

AntClass.java:

```
package testAnt;
import java.util.List;
import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.firefox.FirefoxDriver;
import org.testng.annotations.Test;

public class AntClass {
   @Test
                public void TestNGMethod(){
        WebDriver driver = new FirefoxDriver();
                 driver.get("http://demo.guru99.com/test/home/");
 List<WebElement> listAllCourseLinks =
driver.findElements(By.xpath("//div
[@class='canvas-middle']//a"));
        for(WebElement webElement : listAllCourseLinks) {
                System.out.println(webElement.getAttribute("href"));
                 }
                }
}
```

Step 5) Now, you can run your test.

Steps 1.15.9: Pushing the code to your GitHub repositories

Open your command prompt and navigate to the folder where you have created your files.

Q15) Demonstrate using listeners in Selenium.

Using iTestListener:

- Listener is an interface that modifies the TestNG behavior.
- Listener listens to the event defined in the Selenium and behaves accordingly.
- It is used in Selenium by implementing Listeners Interface.
- iTestListener is used in Selenium to generate logs or customize the TestNG reports.
- iTestListener has the following methods:
    a. **onStart:** onStart method is called when any Test starts.
    b. **onTestSuccess:** onTestSuccess method is called on the success of any Test.
    c. **onTestFailure:** onTestFailure method is called on the failure of any test.
    d. **onTestSkipped:** onTestSkipped method is called when any test gets skipped.
    e. **onTestFailedButWithinSuccessPercentage:** onTestFailedButWithinSuccessPercentage method is called each time Test fails but within the success percentage.
    f. **onFinish:** onFinish method is called after all the tests are executed.
- Open Eclipse and create a Project.
- Create a Listener class that will implement iTestListener.
- The code in Eclipse will look like:

```
package test.testing;

import org.testng.ITestContext;
import org.testng.ITestListener;
import org.testng.ITestResult;

public class ListenersTest implements ITestListener {

	public void onFinish(ITestContext Result) {
	  System.out.println(Result.getName()+"case finished");

	}

	public void onStart(ITestContext Result) {
	  // TODO Auto-generated method stub

	}

	public void onTestFailedButWithinSuccessPercentage(ITestResult Result) {
	  // TODO Auto-generated method stub

	}
```

```
        public void onTestFailure(ITestResult Result) {
            // TODO Auto-generated method stub
            System.out.println("The name of the testcase failed is :"+Result.getName());
        }

        public void onTestSkipped(ITestResult Result) {
            // TODO Auto-generated method stub
            System.out.println("The name of the testcase Skipped is :"+Result.getName());
        }

        public void onTestStart(ITestResult Result) {
            // TODO Auto-generated method stub
            System.out.println(Result.getName()+" test case started");
        }

        public void onTestSuccess(ITestResult Result) {
            // TODO Auto-generated method stub
            System.out.println("The name of the testcase passed is :"+Result.getName());
        }

}
```

- Create a Java class and implement a Listener class in this.
- The code in Eclipse will look like:

```
package test.testing;

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

@Listeners(test.testing.ListenersTest.class)

    public class Home extends Baseclass {

        @Test
        public void clickOnCategory()
        {
         driver.findElement(By.xpath("//div[@id='navigation']/a[1]")).click();
         System.out.println("Clicked on links");
        }

}
```

Step 1.16.2: Running the code

- Run the code through Eclipse.

Q16) Demonstrate how artifactory can be installed.

Installing Artifactory

- Go to this link: https://jfrog.com/open-source/#artifactory.
- Download the tar.gz file as shown in the following screenshot

- Use the following command to extract the tar file.

cd Downloads

tar -xvf jfrog-artifactory-oss-7.5.5-linux.tar.gz

- The tar file will be extracted in a directory named artifactory-oss-7.5.5
- Open the command prompt and go to the bin of the Artifactory folder and run the following commands:

cd Downloads/artifactory-oss-7.5.5/artifactory/app/bin

sudo ./installService.sh

- Now, run the command:

systemctl start artifactory.service

- Now, go to the browser type- localhost:8081.
- Enter the following details:

ID : admin

Password: password

- Click on ‘Welcome Admin.’
- Click on ‘Quick Setup.’
- Create a maven repository.

Step 2.1.2: Setting up Artifactory with Jenkins

- Jenkins is already installed in your practice lab. Refer to QA to QE lab guide for Phase 2 for more information.
- Open command line and run the following command:

sudo less /var/lib/jenkins/secrets/initialAdminPassword

- Copy the password displayed in the command line
- Go to localhost:8080 in the browser and paste the password in the given field
- Click on install suggested plugins
- Go to ‘Manage Jenkins.’
- Click on ‘Manage Plugin.’
- Select the available tab.
- Search for Artifactory.

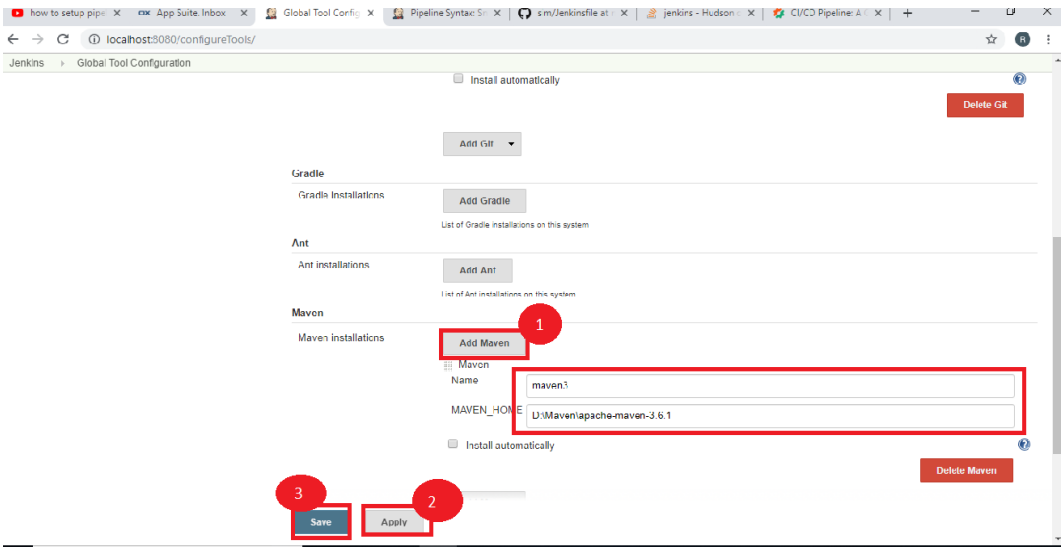
- Again, go to ‘Manage Jenkins.’
- Go to ‘Configure System.’
- There is an option Artifactory.

- Select ‘Add Artifactory.’
- Pass the artifactory localhost URL.
- Now, pass the credentials.
- Click on ‘Apply’ and ‘Save.’

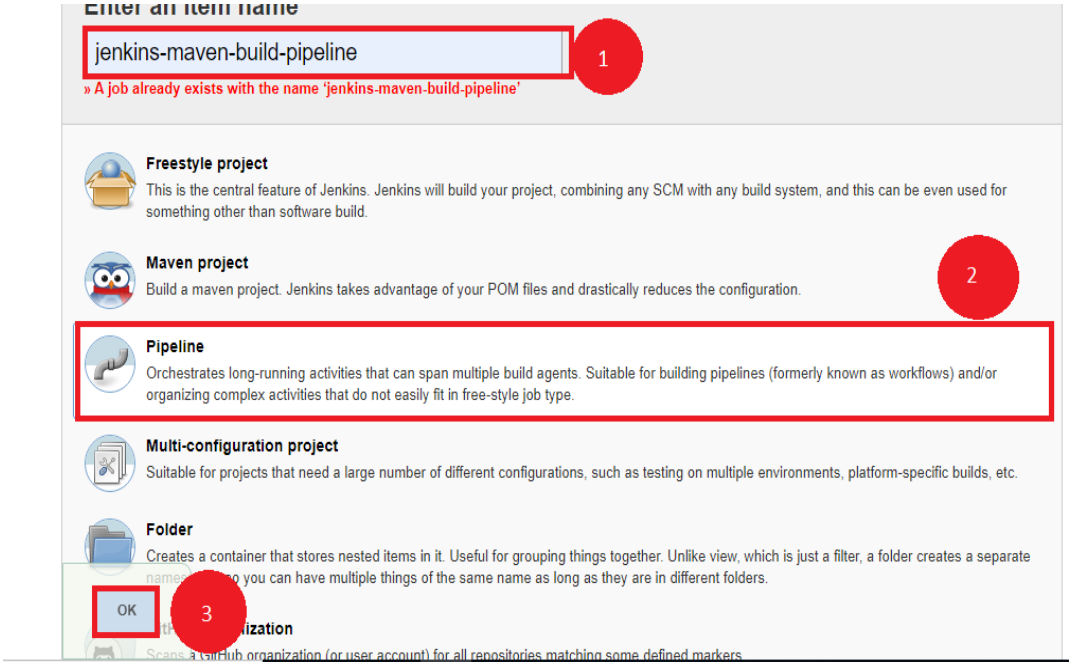
Q17) Build and configure CI/CD pipeline with Maven Project

Setting up a Pipeline

- Jenkins is already set up in your practice lab. Refer to the lab guide for phase 2 for more information.
- Open Jenkins- locahhost:8080.
- Click on **Manage Jenkins.**
- Select **Global Tool Configuration.**
- Set the path of **Java and Maven.**
- Then, **Apply and Save.**



- Select a **New Item.**
- Name the job: **Jenkins-maven-build-pipeline.**
- Select the **Pipeline.**
- Now, click on **OK.**



Step 2.2.2: Writing a Jenkins file

- Create a job that needs to be configured.
- Click on **Pipeline.**
- Select the definition as- **Pipeline script from SCM.**
- Pass the GitHub URL.
- Now, we need to write the Jenkins file inside that git file.

```
node{
 stage('SCM Checkout'){
        git 'https://github.com/Autamation/sim.git'
        }
  stage('Compile-Package'){
   def mvnHome = tool name: 'maven3', type: 'maven'
     bat "${mvnHome}/bin/mvn package"
 bat 'mvn package'
 }
```

```
}
```

- Now, click on ‘Apply’ and ‘Save.’
- Click on **Build Now.**

Q18) Demonstrate how to build and configure CI/CD pipeline with Selenium WebDriver.

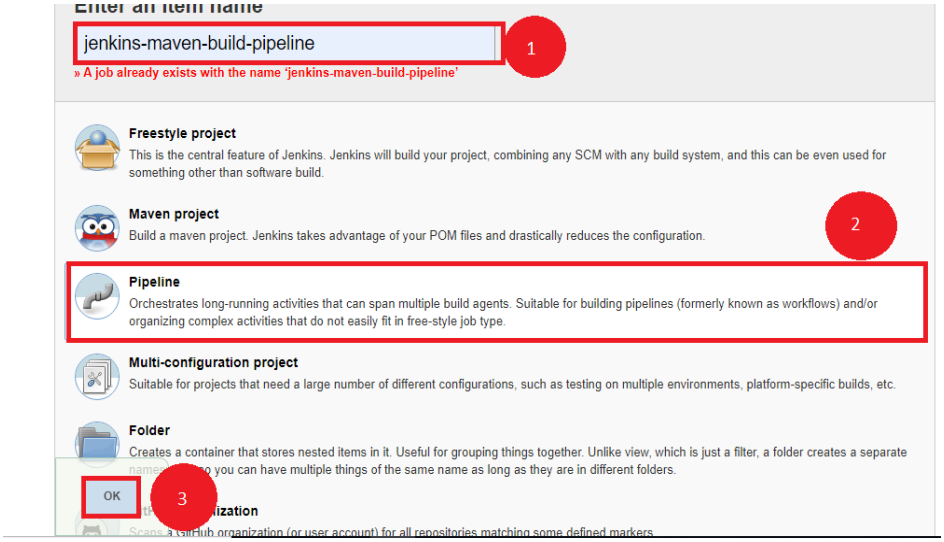
Foking the git repository

- Fork the following repository

  https://github.com/canindit75/JenkinsDemo

Step 2.3.2: Create a Jenkins pipeline job

- Java 1.8 is already installed in your practice lab. Refer to QA to QE lab guide for Phase 1 for more information.
- Jenkins.war file is already present in your practice lab in cd /usr/share/jenkins directory.
- Go to jenkins.war location Start the Jenkins by using command on command prompt:**java -jar jenkins.war.**
- Open browser and type **localhost:8080.**
- Enter the password.
- Create a job.
- Pass a name.
- Select **Pipeline.**
- Click on Ok.



- Create a text file name it **run.sh** in your lab and keep the below given code in it.

```
java -cp bin;lib/* org.testng.TestNG testng.xml
```

- Give executable permission to **run.sh** using the commands below:

chmod 755 run.sh

chmod 777 run.sh

- Push **run.sh in your repository** under master branch.

git push <reponame> master

git status

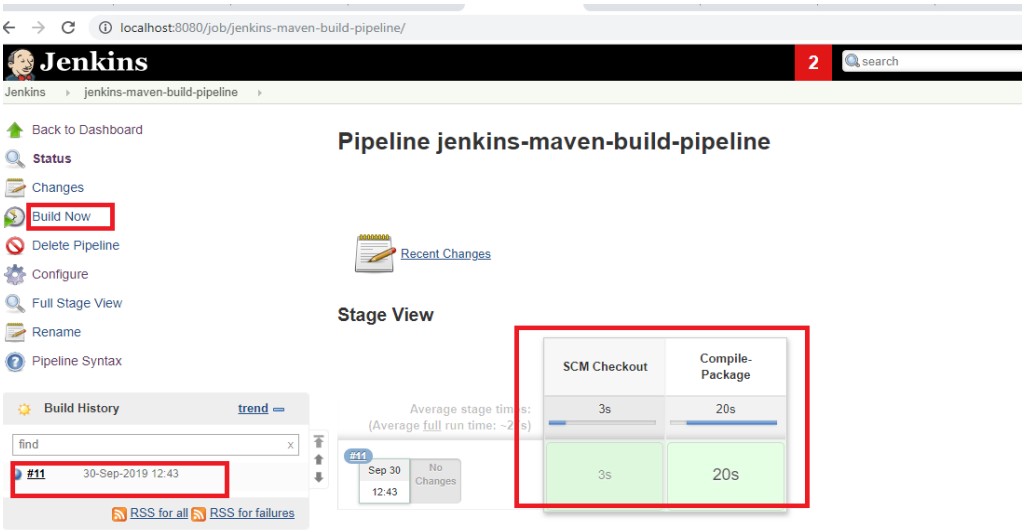
- Go to Jenkins pipeline job.
- Write a groovy script in the pipeline.

```
node {
  def mvnHome
  stage('Preparation') { // for display purposes
    // Get some code from a GitHub repository
```

```
    git 'https://github.com/jglick/simple-maven-project-with-tests.git'
    // Get the Maven tool.
    // ** NOTE: This 'M3' Maven tool must be configured
    // **       in the global configuration.
    mvnHome = tool 'maven3'
}
stage('Build') {
    // Run the maven build
    withEnv(["MVN_HOME=$mvnHome"]) {
        if (isUnix()) {
            sh '"$MVN_HOME/bin/mvn" -Dmaven.test.failure.ignore clean package'
        } else {
            sh '"%MVN_HOME%\bin\mvn" -Dmaven.test.failure.ignore clean package'
        }
    }
}
stage('Results') {
    junit '**/target/surefire-reports/TEST-*.xml'
    archiveArtifacts 'target/*.jar'
}}
```

- Click on Apply and Save.

- Click on Build now.



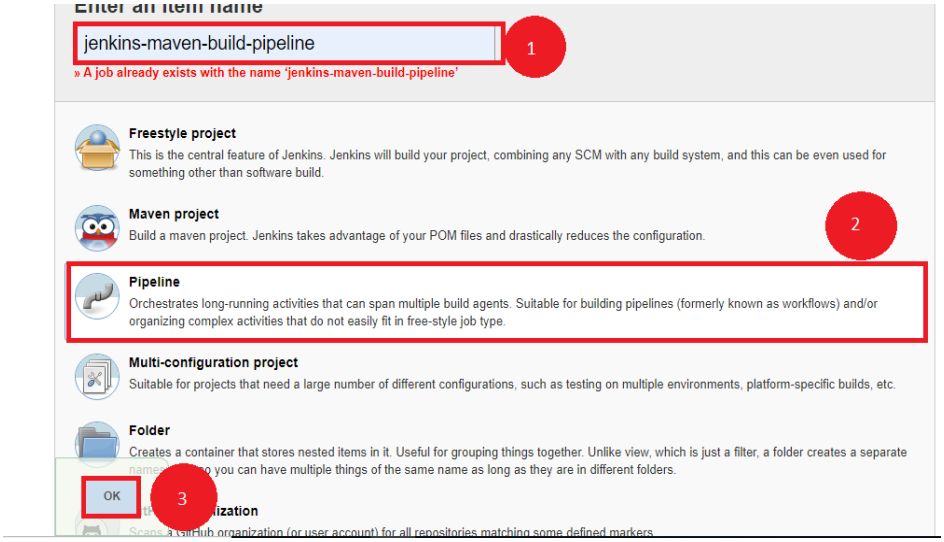
: Foking the git repository

- Fork the following repository

  https://github.com/canindit75/JenkinsDemo

Step 2.3.2: Create a Jenkins pipeline job

- Java 1.8 is already installed in your practice lab. Refer to QA to QE lab guide for Phase 1 for more information.
- Jenkins.war file is already present in your practice lab in cd /usr/share/jenkins directory.
- Go to jenkins.war location Start the Jenkins by using command on command prompt:**java -jar jenkins.war.**
- Open browser and type **localhost:8080.**
- Enter the password.
- Create a job.
- Pass a name.
- Select **Pipeline.**
- Click on Ok.



- Create a text file name it **run.sh** in your lab and keep the below given code in it.

```
java -cp bin;lib/* org.testng.TestNG testng.xml
```

- Give executable permission to **run.sh** using the commands below:

chmod 755 run.sh

chmod 777 run.sh

- Push **run.sh in your repository** under master branch.

git push <reponame> master

git status

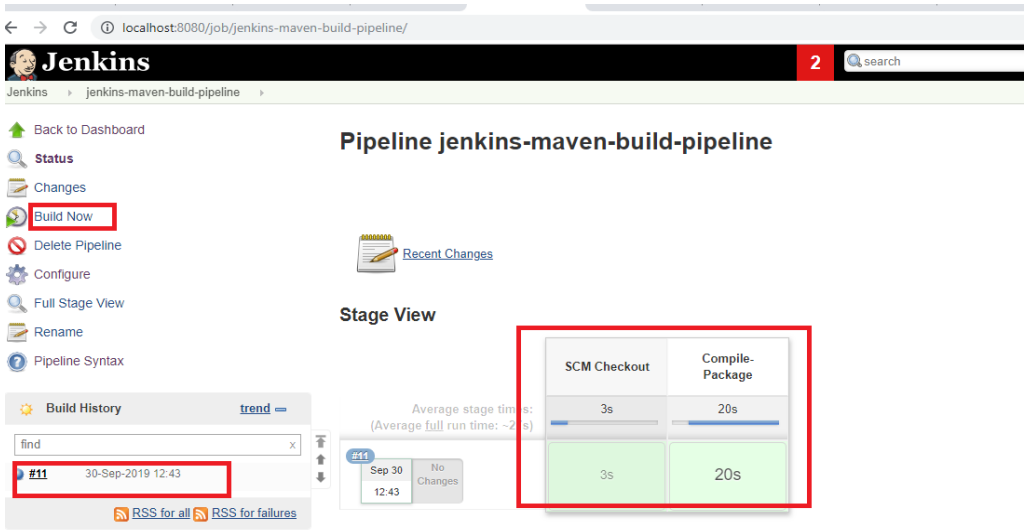
- Go to Jenkins pipeline job.
- Write a groovy script in the pipeline.

```
node {
  def mvnHome
  stage('Preparation') { // for display purposes
    // Get some code from a GitHub repository
```

```
    git 'https://github.com/jglick/simple-maven-project-with-tests.git'
    // Get the Maven tool.
    // ** NOTE: This 'M3' Maven tool must be configured
    // **       in the global configuration.
    mvnHome = tool 'maven3'
}
stage('Build') {
    // Run the maven build
    withEnv(["MVN_HOME=$mvnHome"]) {
        if (isUnix()) {
            sh '"$MVN_HOME/bin/mvn" -Dmaven.test.failure.ignore clean package'
        } else {
            sh '"%MVN_HOME%\bin\mvn" -Dmaven.test.failure.ignore clean package'
        }
    }
}
stage('Results') {
    junit '**/target/surefire-reports/TEST-*.xml'
    archiveArtifacts 'target/*.jar'
}}
```

- Click on Apply and Save.

- Click on Build now.



Q19) Demonstrate Selenium integration with Jenkins.

Creating a Jenkins Pipeline job

- Java 1.8 is already installed in your practice lab.
- Jenkins.war file is already present in your practice lab in directory /usr/share/jenkins.
- Go to the jenkins.war location. Now, start Jenkins by using the following command on command prompt: java -jar jenkins.war.
- Open your browser and type- localhost:8080.
- Enter the password.
- Create a job.
- Enter the name.
- Select the Pipeline.
- Now, click on OK.

Step 2.4.2: Integrating Selenium WebDriver with Jenkins

- Go to the Selenium script location.

- Add all the Selenium libraries.

- Create a text file run.bat and keep it within double quotation marks.

- Go to the Jenkins Pipeline job.
- Pass the Selenium script location.
- Write a groovy script in Pipeline.
- Click on Apply and then Save.

- Now, click on Build Now.

Q20) Demostrate the TDD with TestNG

Performing a TDD test:

- To perform a TDD test, follow the below steps:
    a. Add the test.
    b. Execute the test and see if the new one fails.
    c. Write the code.
    d. Execute the test.
    e. Refactor the code.
    f. Now, repeat the steps mentioned above.

    Now, let's look at the above steps in detail:

a. Firstly, write the code that will be based on the requirements in Eclipse. It should look something like:

```
package test.testing;

import org.testng.Assert;
import org.testng.annotations.Test;

public class AddNumbers {

        @Test

        public void addIntegerNumbers()
```

```
    {
        Calculator myCalculator = new Calculator();
        int expected= 30;
        int actual= myCalculator.add(10,20);
        Assert.assertEquals(actual, expected);
    }
}
```

b. Now, if we execute our test for the first time, we should get the below error:
FAILED: addIntegerNumbers
java.lang.Error: Unresolved compilation problems:
Calculator cannot be resolved to a type
Calculator cannot be resolved to a type

c. Write the code shown below to resolve the above error in Eclipse. It will look something like:

```
package test.testing;

public class Calculator {

    public int add(int number1, int number2)
    {
            return 0;
    }
}
```

d. Now, execute our test:
FAILED: addIntegerNumbers
java.lang.AssertionError: expected [30] but found [0]

e. Refactor the code in Eclipse. It will look like:

```
package test.testing;

public class Calculator {

    public int add(int number1, int number2)
    {
            return (number1+number2);
    }
}
```

f. Now, if execute our test again, it will show the below message:
PASSED: addIntegerNumbers

Step 3.1.2: Running the code:

- Run the code through Eclipse.