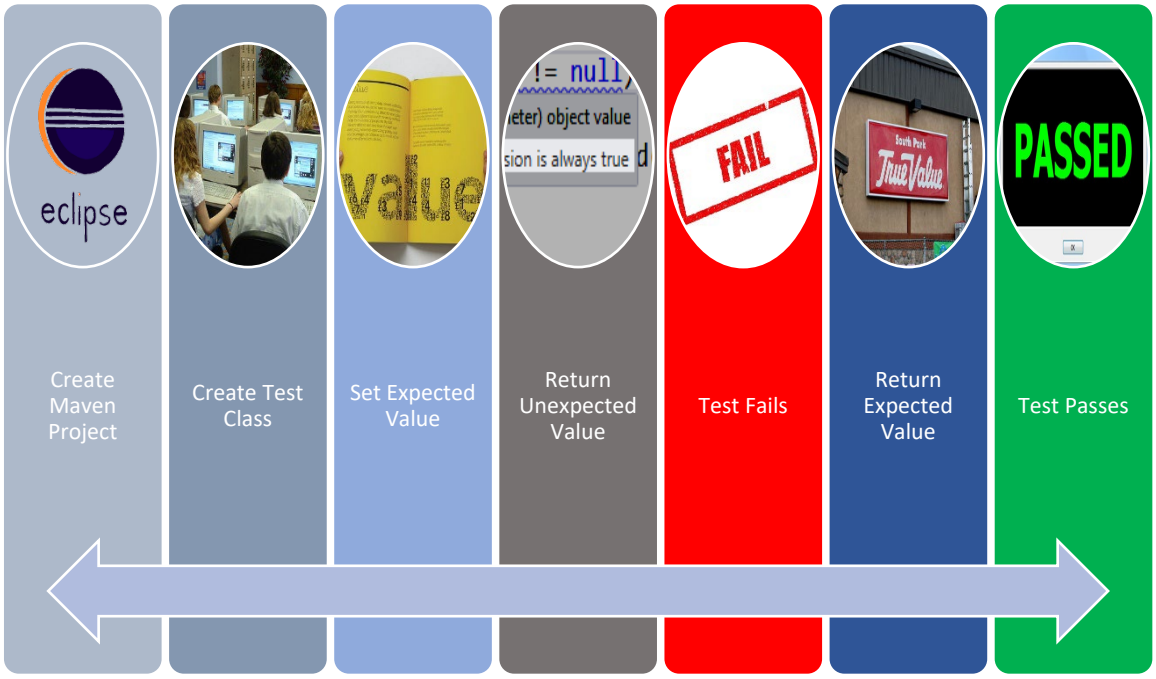
Trigger Jenkins Build

In this demo, we will utilize **Maven** to set up a *test class* in **Eclipse**. Our **Maven** project *test class* will run an *echo* method that returns the value that we pass into it, which will be evident when the test passes. We will also attempt to insert a value that does not meet the "expected" requirements, which will cause the test to fail.

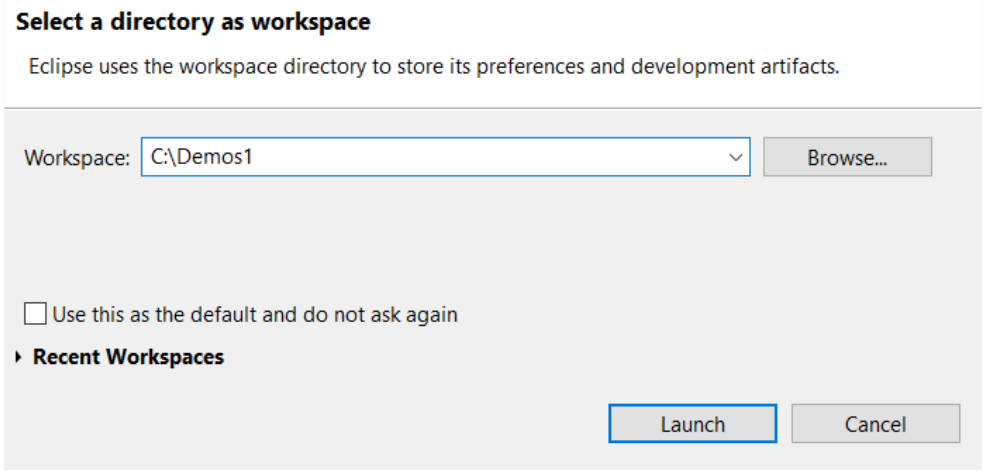


Once our test has been run, we will create a ***post commit trigger*** for our local instance of **Jenkins**. Once we specify the path & insert it into our local repository, **Jenkins** will fire off the build on our local repository. This eliminates the necessity of pulling it from **GitHub**, or any other remote repository.

Periodic builds tend to get polluted with irrelevant code that was never changed, so it is generally better to create a triggered build. When Developers trigger a build process every time, they check in new code to the software repository, it reduces stress on the server as it is only run when necessary.

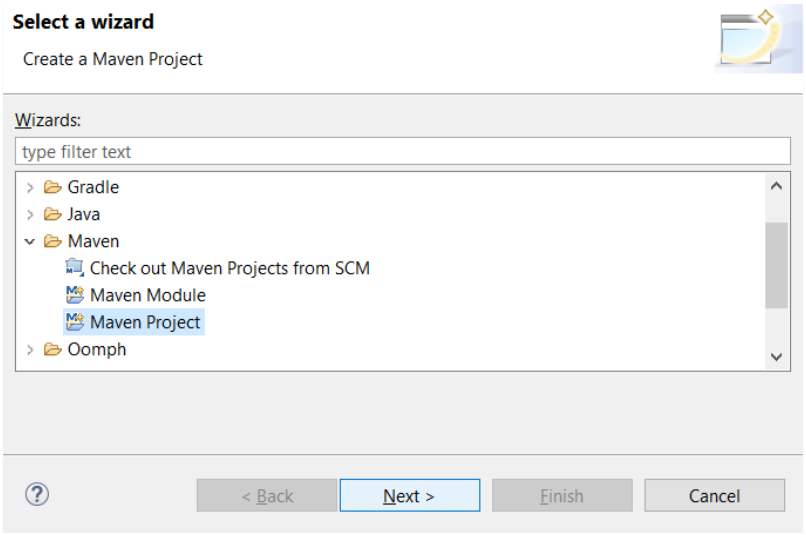
Git hooks are scripts that **Git** executes before or after events such as: *commit, push*, and *receive*. They are a built-in feature, and they run locally. Utilization of these hooks can increase productivity as a developer, as they are able to push code staging or production environments without ever leaving Git. This exercise will give a high-level example of how this can be accomplished, so let's get started.

1. Launch **Eclipse**
2. Choose your workspace

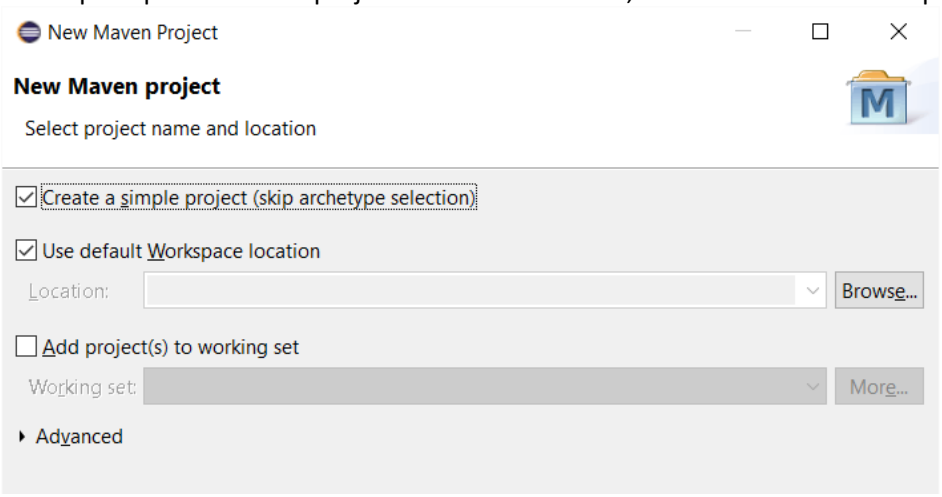


3. Close the welcome window

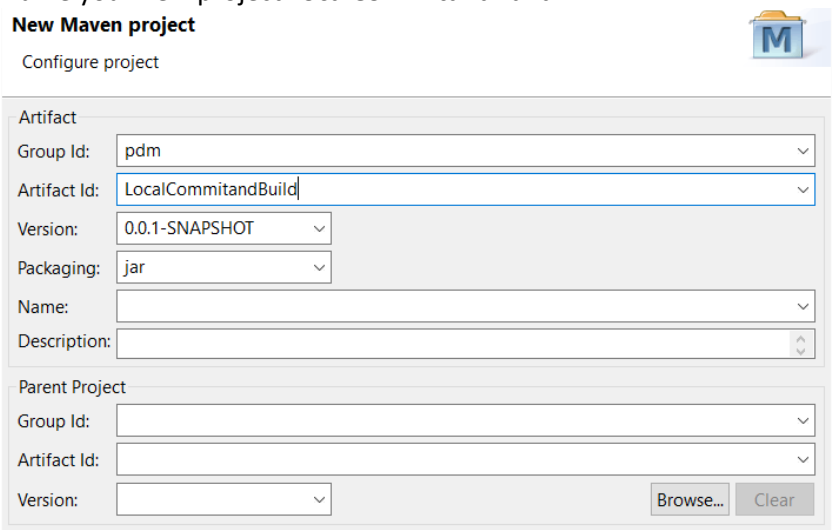
4. Open a new Maven Project



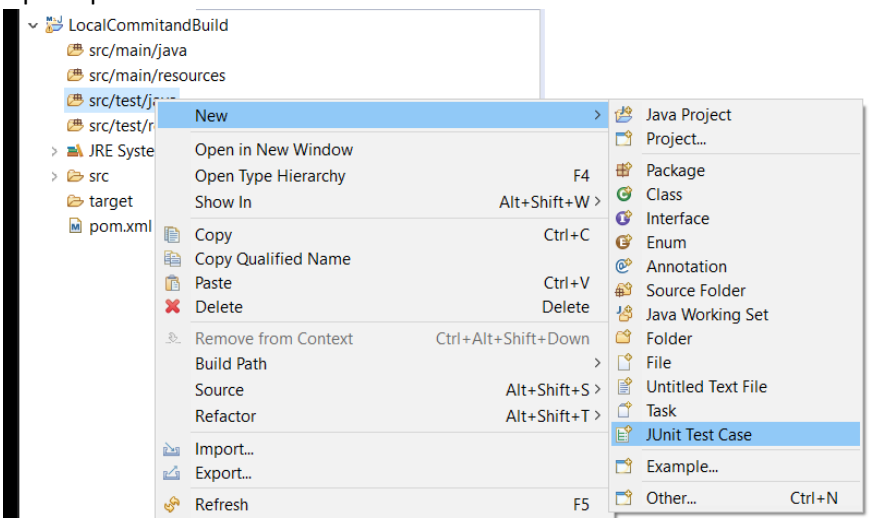
5. When prompted to select project name and location, check the “Create a simple project” box



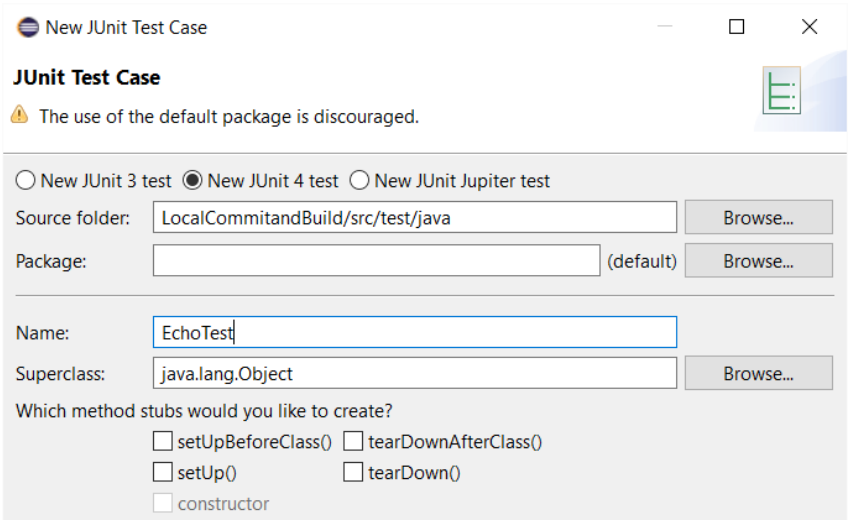
6. Name your new project **LocalCommitandBuild**



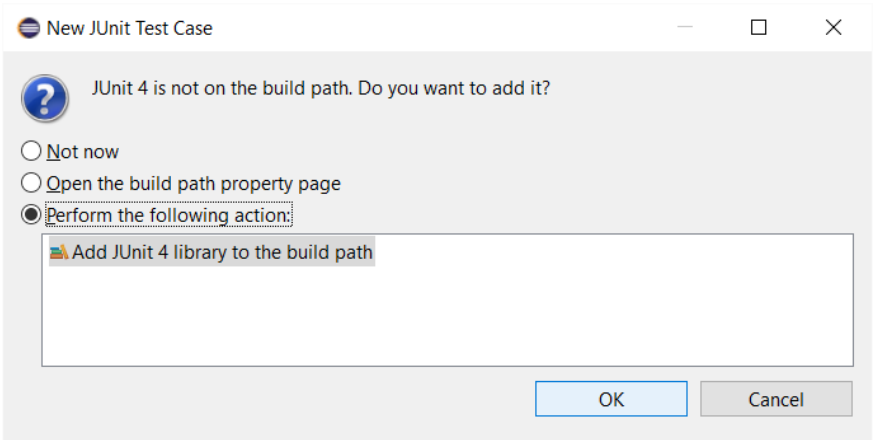
7. Open up a basic **J-Unit Test Case**



8. Name it **EchoTest**



9. Accept the Defaults



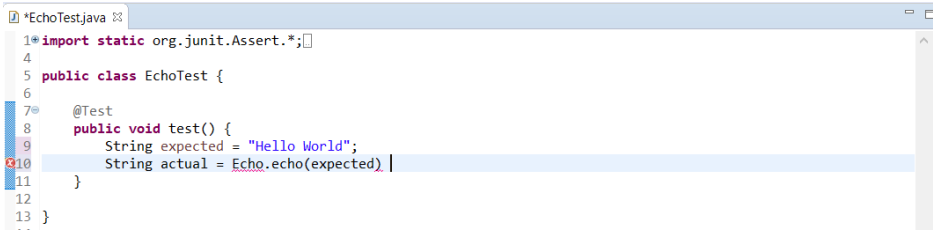
10. Highlight the **fail("Not yet implemented");** annotation

EchoTest.java

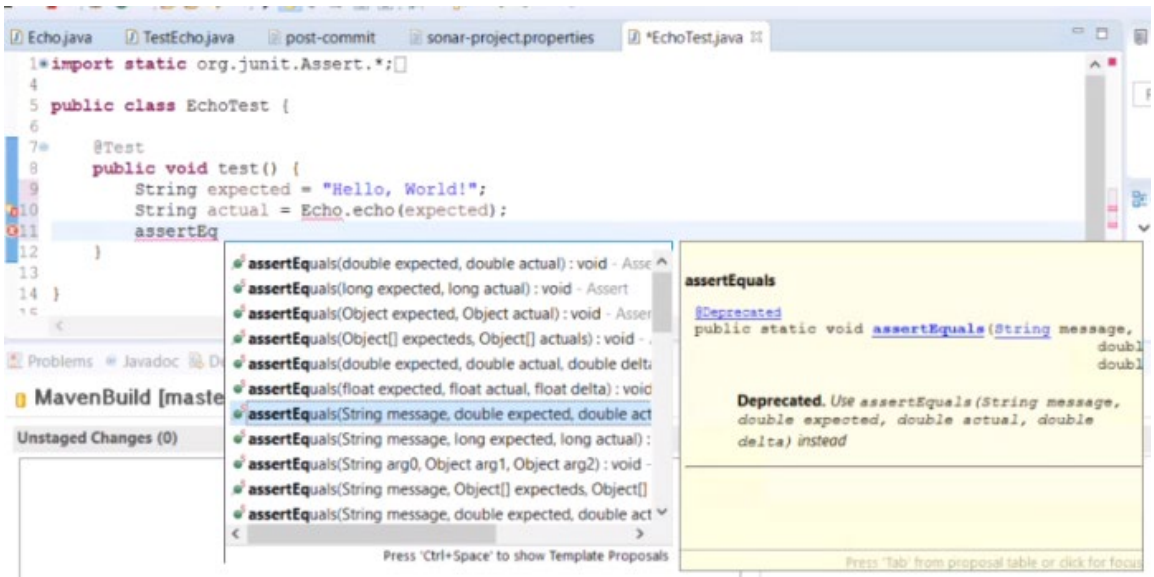
```
1  import static org.junit.Assert.*;
4
5  public class EchoTest {
6
7      @Test
8      public void test() {
9          fail("Not yet implemented");
10     }
11
12 }
```

11. **Replace the text with *String* expected**

```
String expected = "Hello World";
String actual = Echo.echo(expected)
```



12. Use **assertEquals** as a string by typing: *assertEquals* and selecting the "*(string message, double expected, double actual double delta)"* option



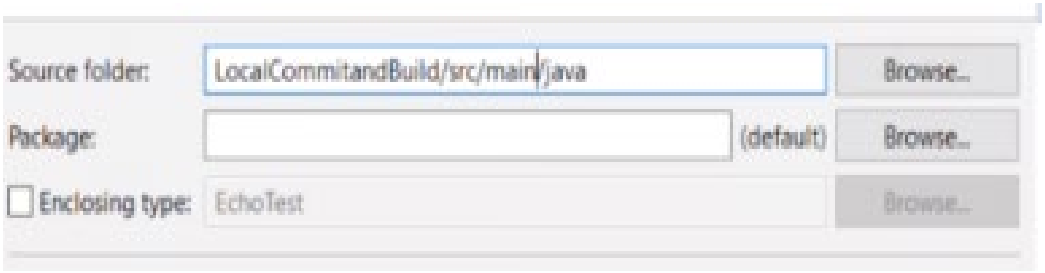
13. Complete the string by typing: ***assertEquals*(“should get “ + expected, expected, actual);**

```
choTest.java   Echo.java   string.java
import static org.junit.Assert.*;

public class EchoTest {

    @Test
    public void test() {
        String expected = "Hello World!";
        String actual = Echo.echo(expected);
        assertEquals("should get " + expected, expected, actual);
    }

}
```

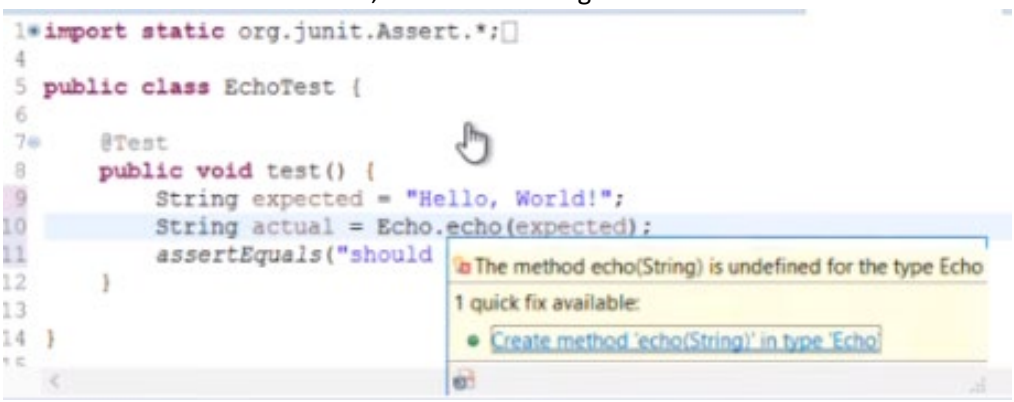
Now we have our test, but we do not yet have our class.

14. Hover over Echo.echo, and use the code generation tool to create our test class by choosing **Create Class “Echo**”

15. Accept the default package, and change src/test to src/main



16. Our echo method doesn't exist, so use the code generator to create it.

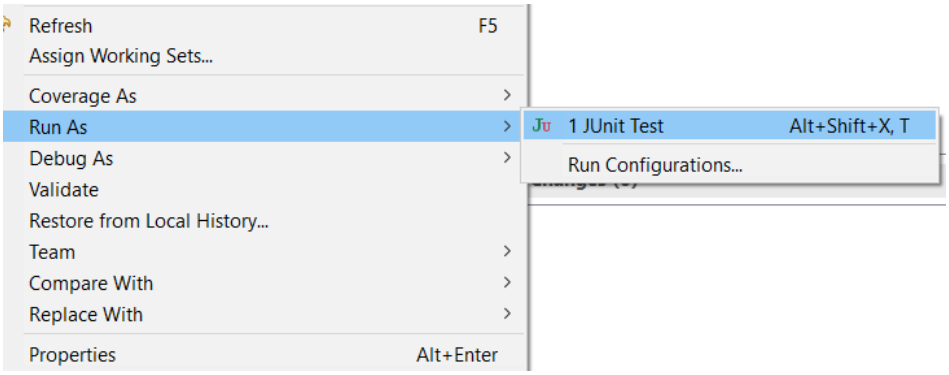


17. Right now, notice that our method returns a value of **null**

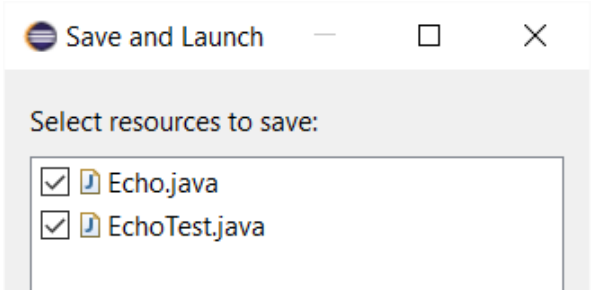
```
public class Echo {

    public static String echo(String expected) {
        // TODO Auto-generated method stub
        return null;
    }

}
```

18. Navigate back to the **EchoTest** class

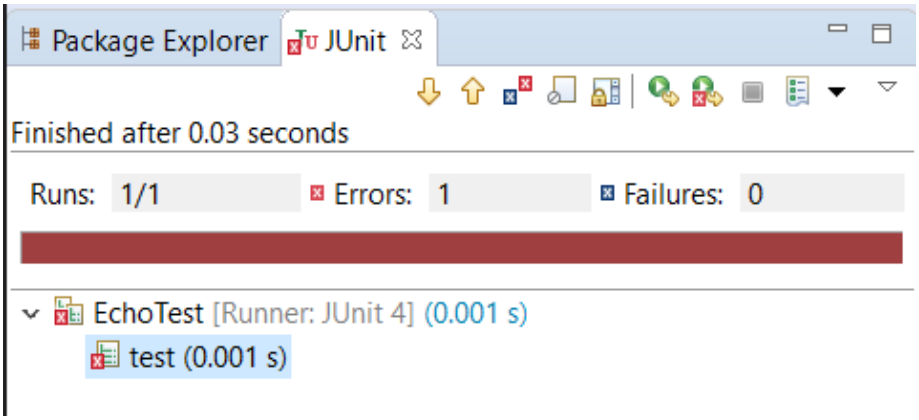
19. Run it as a J-unit test



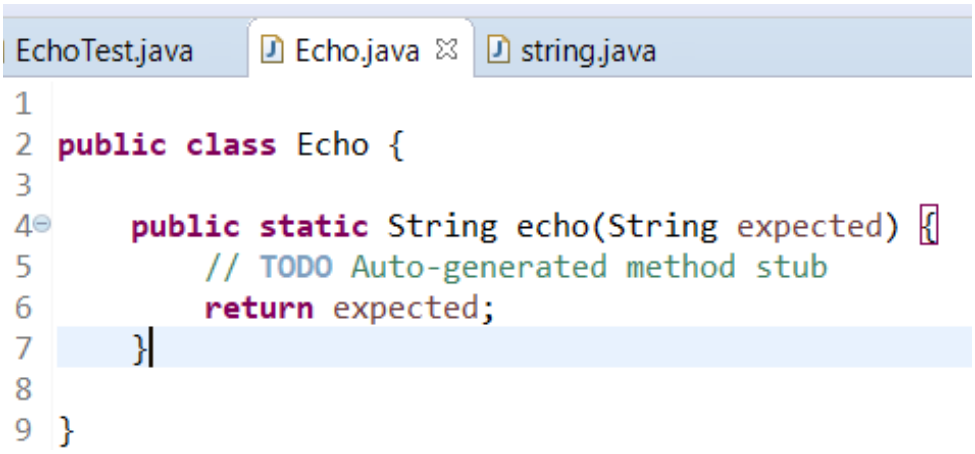
20. Save your files



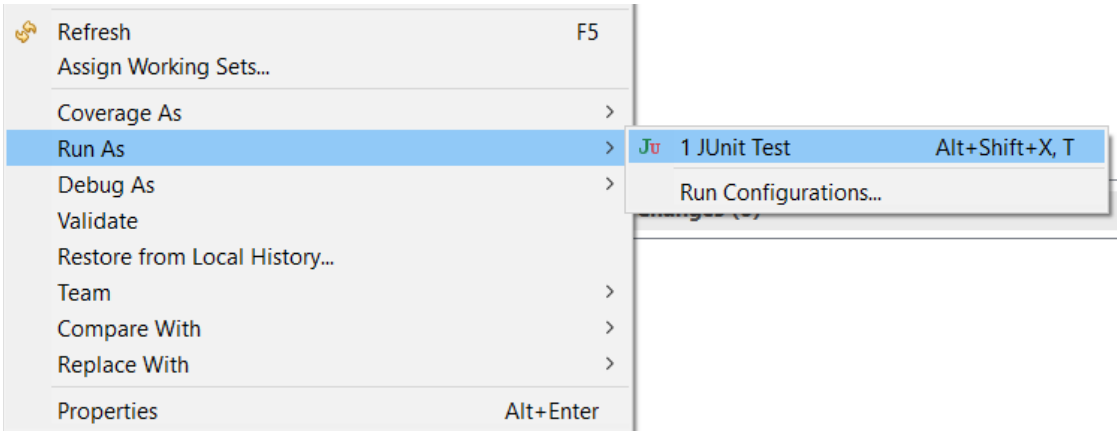
Notice that with your current settings, the **J-Unit** test fails. This is because we have not yet fixed our method to return a value of expected.



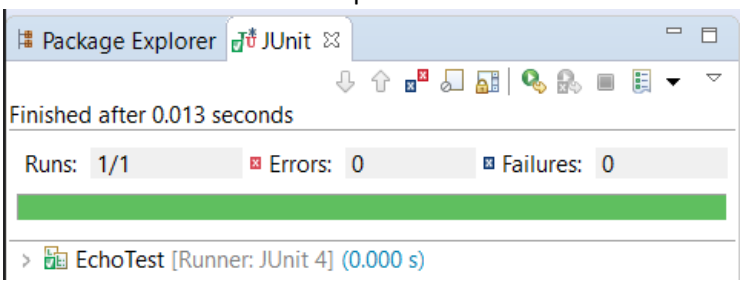
21. Navigate back to the echo method and replace **null** with **expected.**



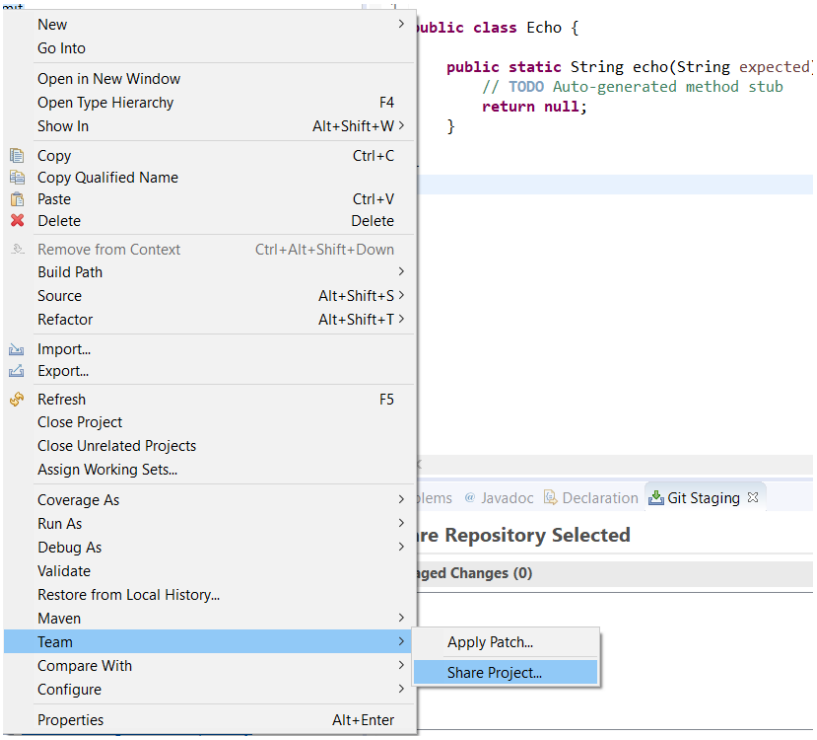
22. Navigate back to **EchoTest** and run another **J-unit Test**.



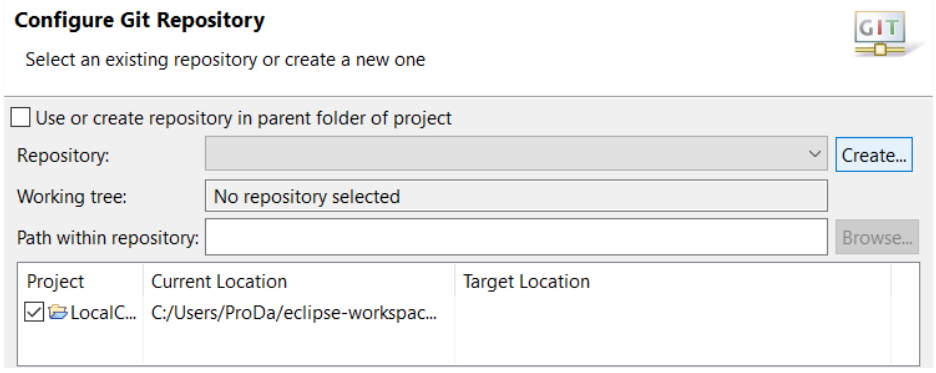
23. Notice that this time the test passes



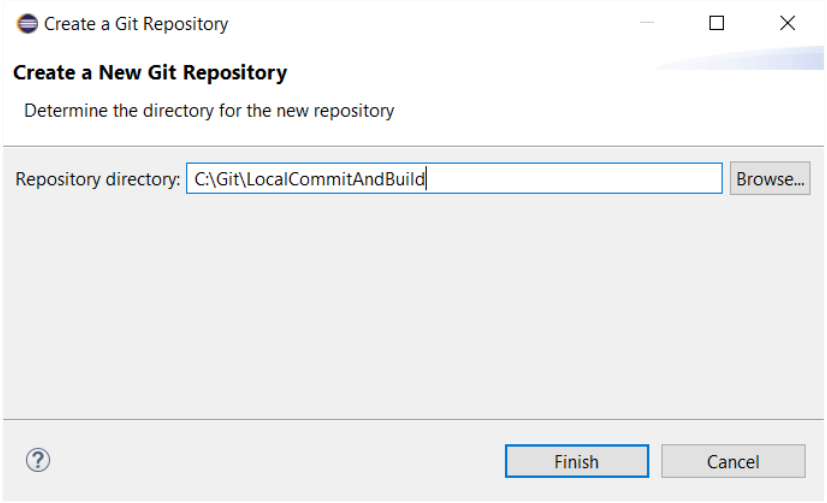
24. Now that we have our project working correctly on local commit and build, navigate back to the Project Explorer. Hover your mouse over **LocalCommandBuild**, Right click, and choose **Team > Share** Project.



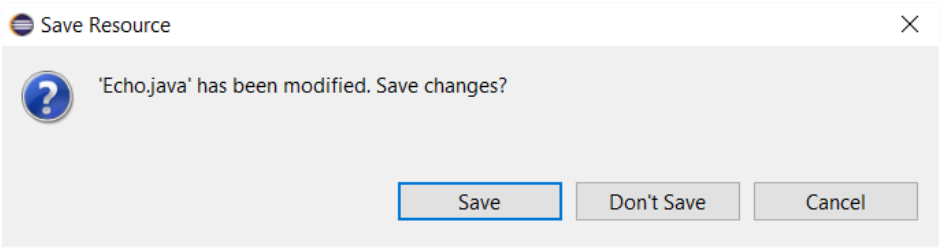
25. Click the **Create** button



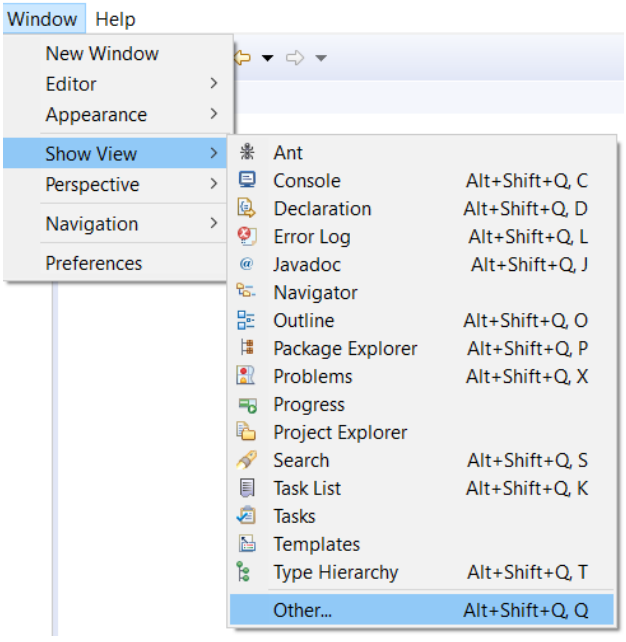
26. Name it **LocalCommitAndBuild**



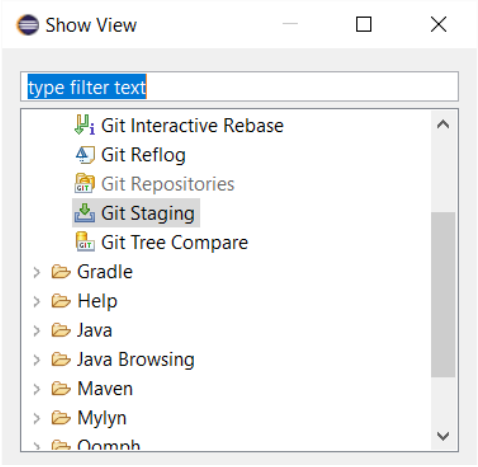
27. Save your changes



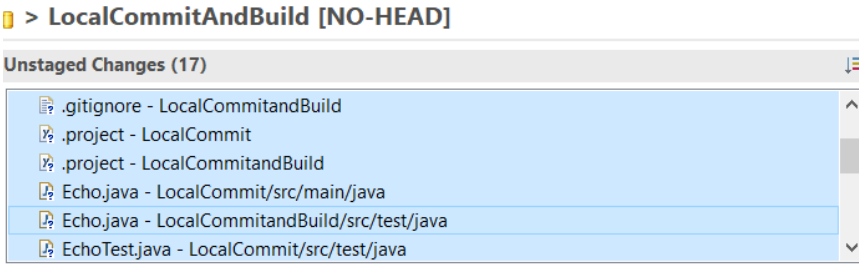
28. Click on the **Window** menu, choose **show view** > **other**



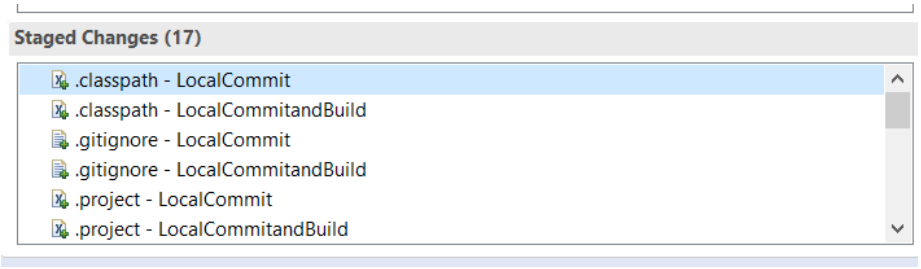
29. In the **Show View** dialogue box, select **Git Staging**



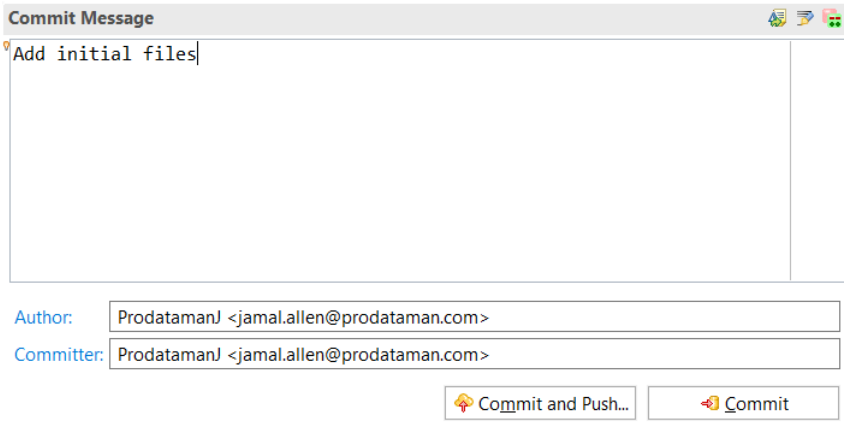
30. Navigate to the **Unstaged Changes** window, click on a file, and press ctrl + all to select each file



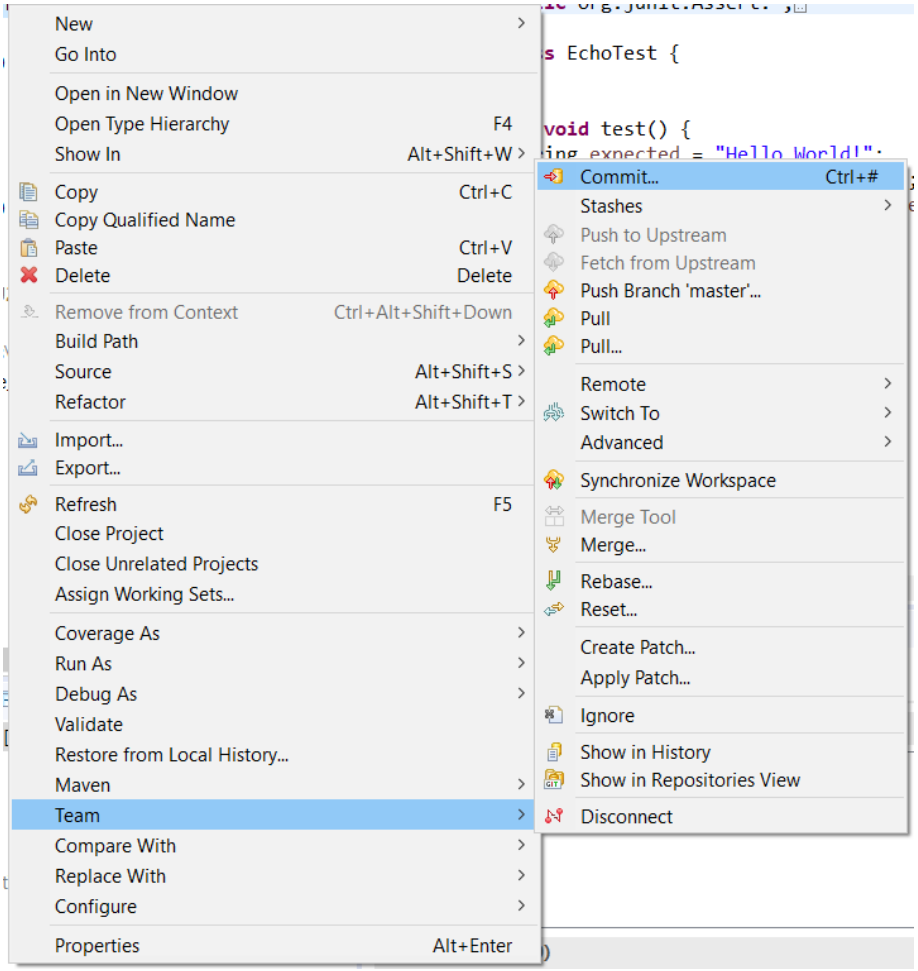
31. Drag and drop the files into the **Staged Changes** dialogue,



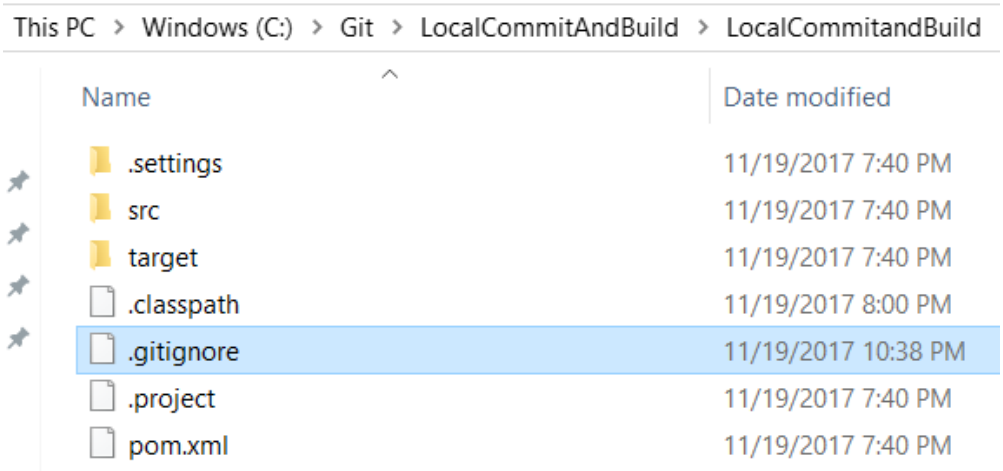
32. Type **Add initial files** in the Commit Message dialogue, and pressing the **Connect** button.



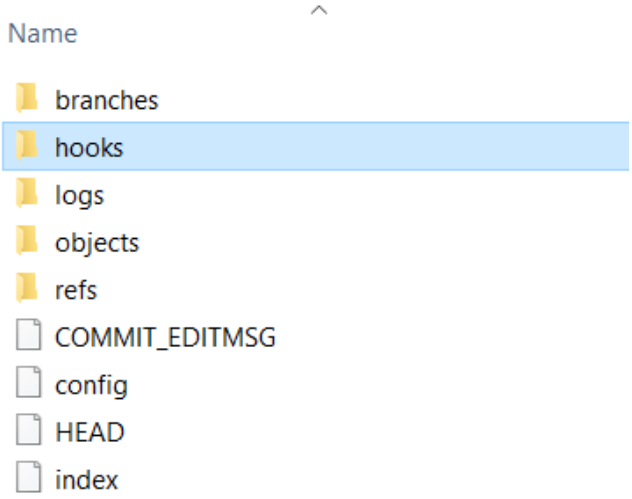
33. Re-commit your files by selecting **LocalCommit [LocalCommitAndBuildMaster]**, right clicking, and choosing **Team > Commit**



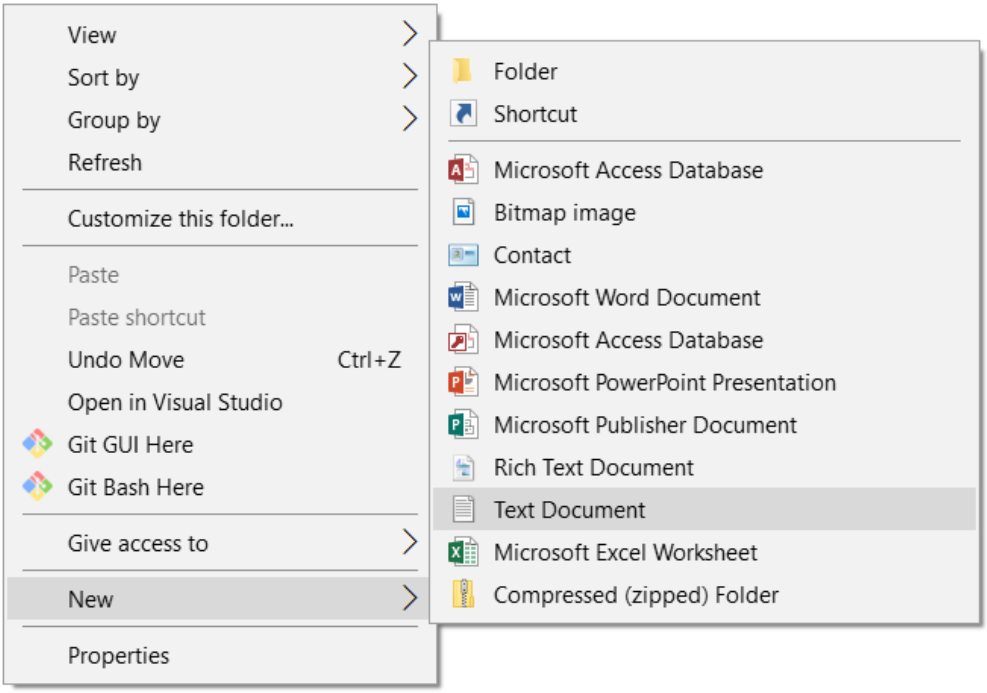
34. Navigate to **C: Drive > Git > LocalCommitAndBuild**
Notice you can see all of your files.



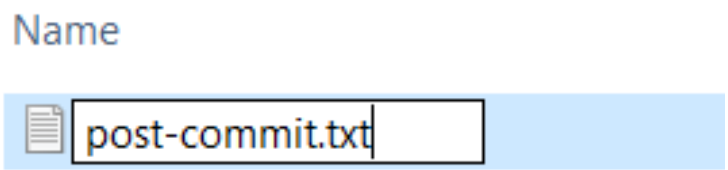
35. Go back to the **Git** folder and navigate to **LocaCommitAndBuild** >**.git** > **hooks**



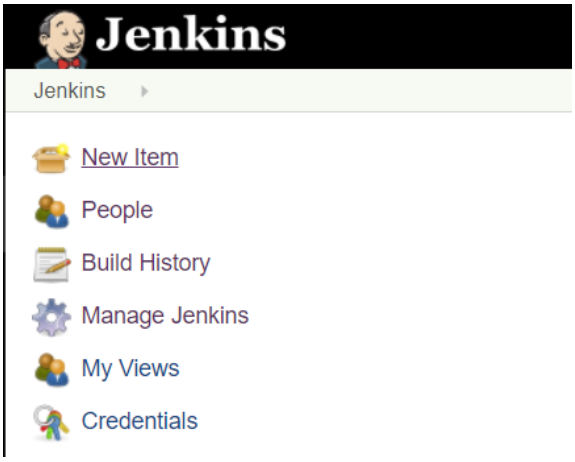
36. In the **hooks** folder, open a new text document.



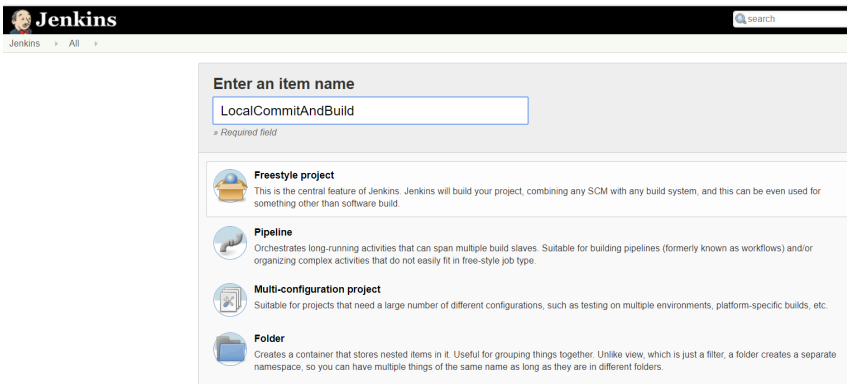
37. Name your new text file **post-commit**

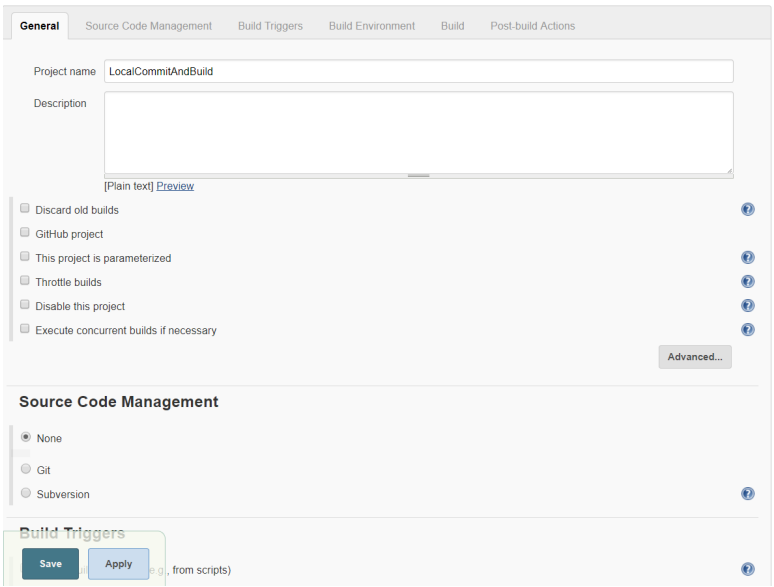


38. Navigate to the **Jenkins** dashboard, and click on **New Item**.

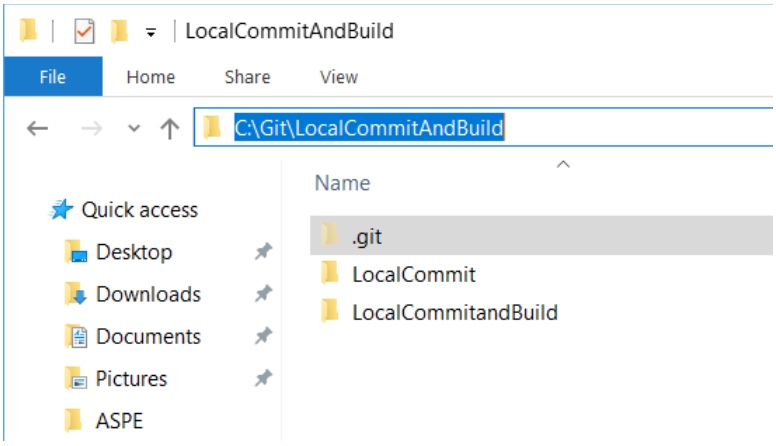


39. Create a Freestyle project, and name it **LocalCommitAndBuild**

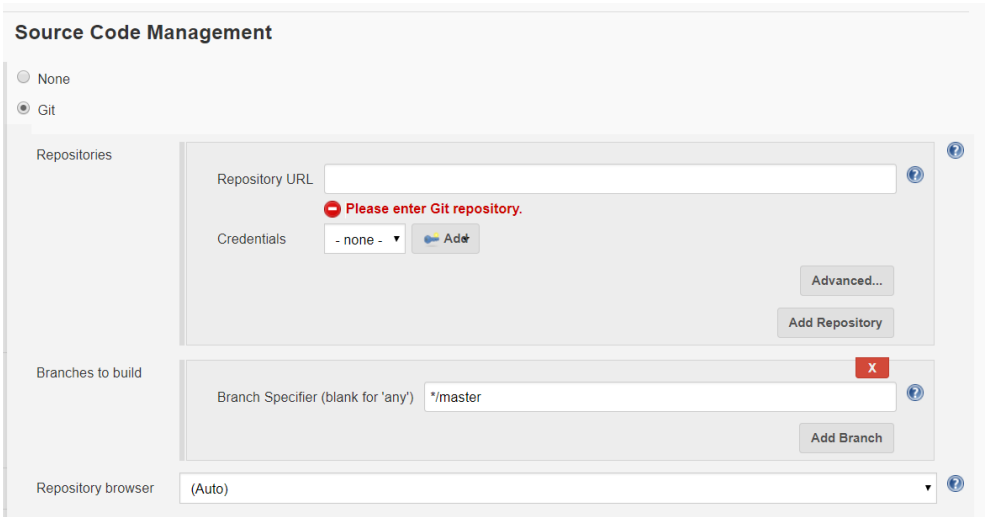




40. In the next step, we will need the address for our **Git** repository, so navigate back to it, and copy the URL to your clipboard.



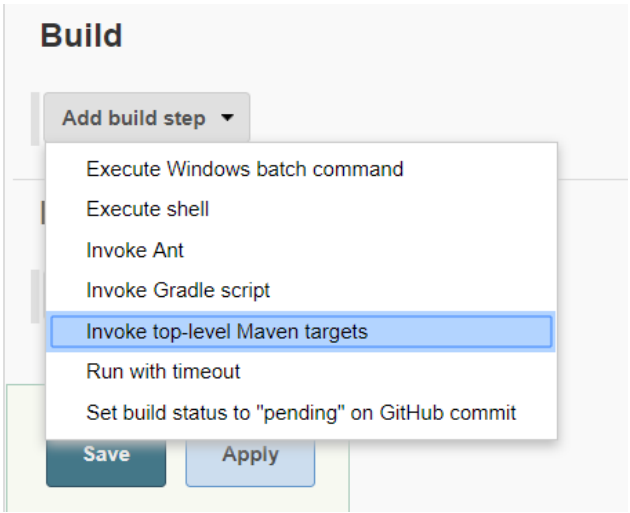
41. Navigate back to **Jenkins**, scroll down to the **Source Code Management** Tab, and click the **Git** option.



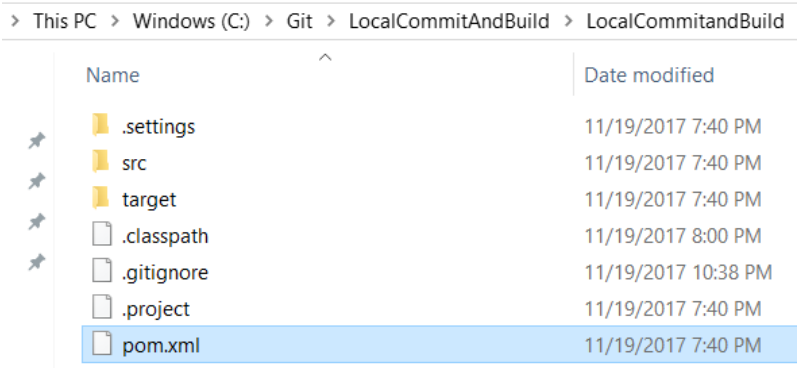
42. For our *build trigger*, we are going to have it poll **Source Control**, but we are not going to set a schedule. That basically just means it's going to use the **Git** plug-in to look for any remote requests sent by the URL to have it build this job.

Since we want to poll source control, scroll down to **Bild Triggers**, and check the **Poll SCM** checkbox.

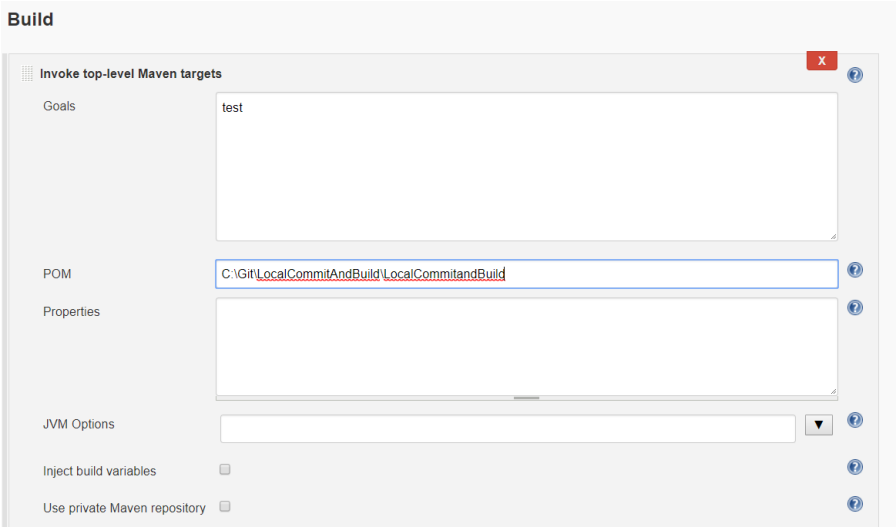
43. Scroll down to the **Build** tab, select **Add Build step** > Invoke top level **Maven** targets



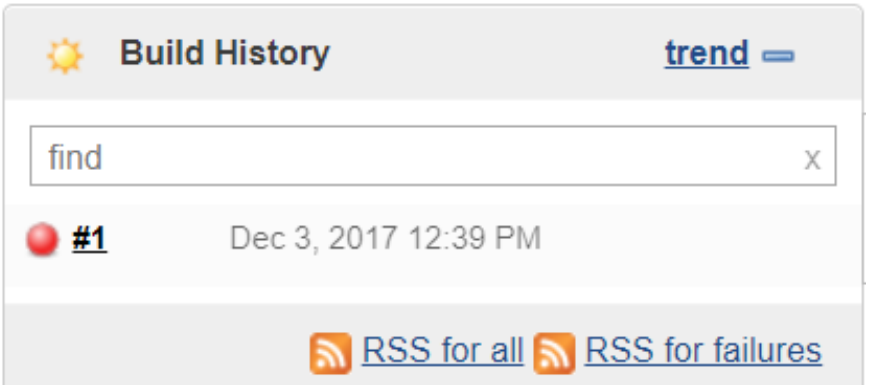
44. Navigate back to your local **Git** repository, and copy the address where your pom.xml is located: **Git > LocalCommitAndBuild > LocalCommitAndBuild**



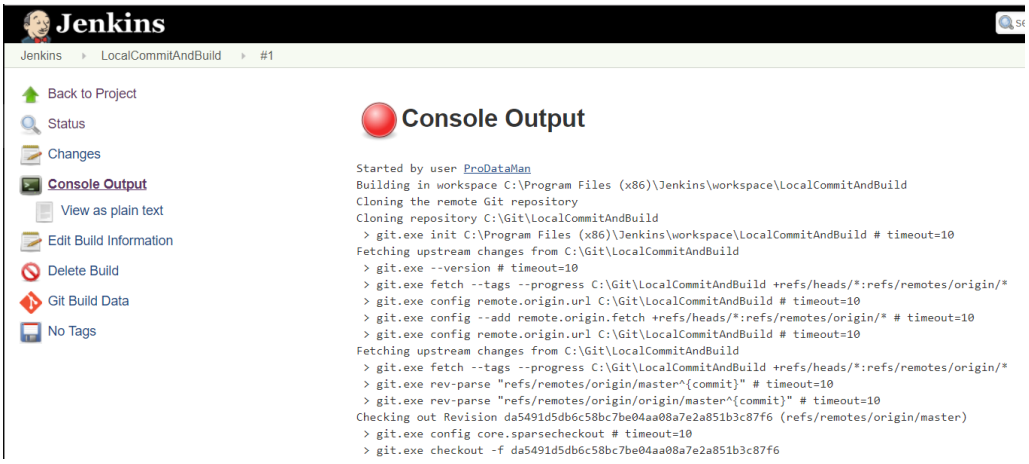
45. In the **Build** tab, click on the **Advanced** button, type test next to the **Goals** option, paste the local path to your repository from your clipboard into the **POM** dialogue, and click **Save**.



46. In the **Build History**, notice that your build has an error.

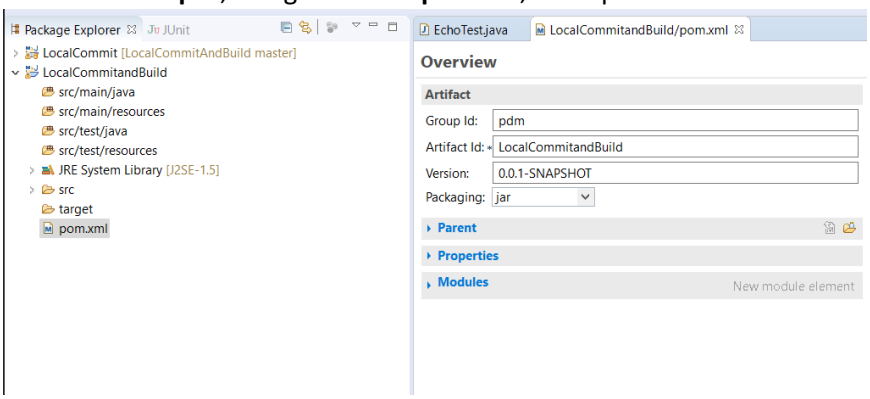


47. Open your build and choose the **Console Output** option from the menu.



This is because we didn't edit the **pom** file with the **J-Unit** dependency.

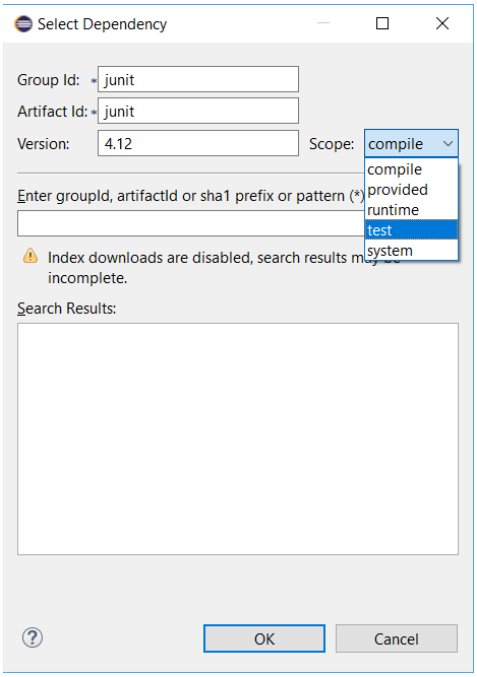
48. Go back to **Eclipse**, navigate to the **pom** file, and open it:



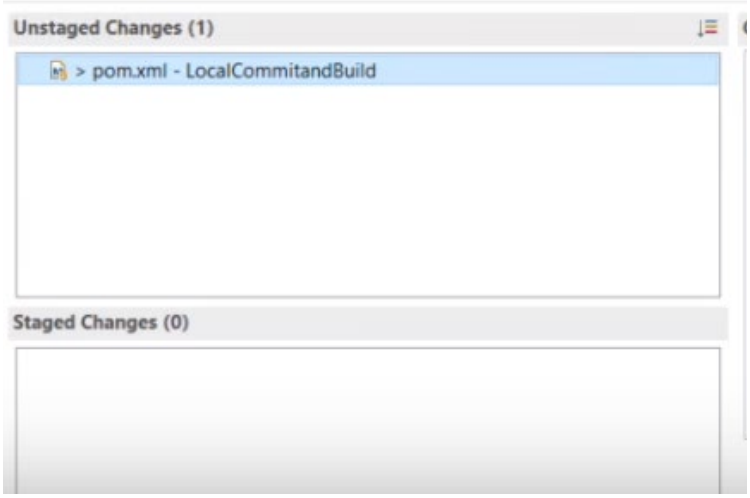
49. Click on the **Dependencies** tab, and choose **Add**

50. Enter these values and click the **OK** button:

    In the Group id: junit
    Artifact ID: junit
    Version: 4.12
    Scope: test

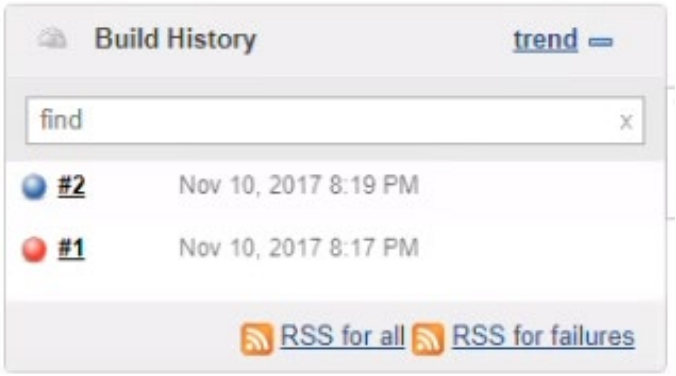


51. Save your **pom** file to the **Unstaged Changes** area, and drag it into the **Staged Changes**



52. Type update pom in the Commit Message dialogue, and click the **Commit** button

53. Navigate back to **Jenkins** and click on the **Back to Project** link. In the **Build History**, notice that a build was triggered, and this time we can see that this time, there were no errors because of the blue dot which indicates a successful build.



Our post commit build was fired off successfully once we went back and updated our dependencies to allow our **J-unit** test to run and pass. Why? Because we are AWESOME!

Conclusion

In this demo, we created a **Maven** project in **Eclipse**. Our **Maven** project had a test class called **EchoTest** which ran a simple echo method that echoed back the same value that we passed into it.

In this case "**Hello World**" was the expected value that we expected to be echoed back to us. We compared that expected value with the actual value that was returned, and since we got **Hello World** back, we know that it passed. In other words, our **Java** class accepted a string in a parameter called **expected**, which returned or "echoed" back that same string, thus the test passes.

We also created a ***post commit trigger*** in this demo, which fired off our build trigger in the **Git** plugin in our local instance of **Jenkins**. Essentially, we put the path to our local repository, which allowed **Jenkins** to fire off the build without having to pull it from **Git Hub**, or some other remote repository. In an actual production scenario, you would want to have a centralized repository for shared integration tests. This demo simply demonstrates a way that developers can fire off a build locally without wasting time on a series of complex processes.