

Continuous Integration with Jenkins-CI

Student Labs

Table of Contents

Lab 1 - Configure Tools in Jenkins.....	3
Lab 2 - Create a Jenkins Job	7
Lab 3 - Advanced Jobs	24
Lab 4 - Template Jobs	33
Lab 5 - Node.js based Jenkins Job and Manual Approval	38
Lab 6 - Create a Pipeline.....	41
Lab 7 - Advanced Pipeline with Groovy DSL	52
Lab 8 - Configure Jenkins Security	63

Lab 1 - Configure Tools in Jenkins

In this lab you will verify that Jenkins Continuous Integration is already installed and you will configure it.

At the end of this lab you will be able to:

1. Verify Jenkins is running
2. Configure tools in Jenkins

Part 1 - Configure Jenkins

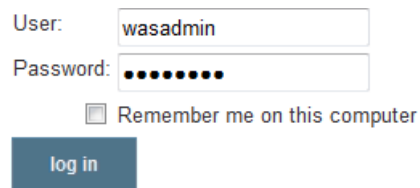
After the Jenkins installation, you can configure few other settings to complete the installation before creating jobs.

You will be setting JDK HOME and Maven Installation directory.

__1. To connect to Jenkins, open Firefox and enter the following URL.

`http://localhost:8080/`

__2. Enter **wasadmin** as user and password and click **Log in**.

A screenshot of the Jenkins login page. It features a 'User:' label next to a text input field containing 'wasadmin'. Below it is a 'Password:' label next to a password input field filled with dots. Under the password field is a checkbox labeled 'Remember me on this computer'. At the bottom is a blue button with the text 'log in'.

__3. Don't save the password if prompt or select Never Remember password for this site.

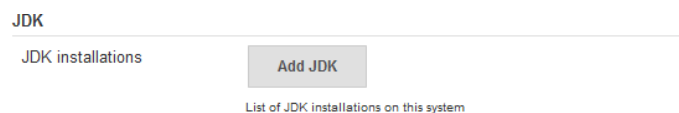
__4. Click on the **Manage Jenkins** link.



__5. Click **Global Tool Configuration**.



__6. Scroll down and find the JDK section, Click **Add JDK**.



__7. Enter **jdk1.8** for JDK name.

__8. Don't check the 'Install automatically' option. Uncheck if already checked.

__9. Enter JAVA_HOME value as **C:\Program Files\Java\jdk1.8.0_291**

__10. Verify your settings look as below:

JDK

JDK installations

Add JDK

Name	JAVA_HOME
jdk1.8	C:\Program Files\Java\jdk1.8.0_291

☐ Install automatically

__11. In the Maven section, click **Add Maven**.

Maven

Maven installations

Add Maven

List of Maven installations on this system

__12. Enter **Maven** for Maven name.

__13. Uncheck the 'Install automatically' option.

__14. Enter **C:\Software\apache-maven-3.3.9** for MAVEN_HOME. Make sure this folder is correct.

__15. Verify your settings look as below:

Maven

Maven installations


Name	MAVEN_HOME
Maven	C:\Software\apache-maven-3.3.9

☐ Install automatically

Delete Maven

__16. Scroll and find **Git**, you may see an error regarding the git path. Enter the following path and then hit the tab key:

C:\Program Files\Git\bin\git.exe

 Git	
Name	Default
Path to Git executable	C:\Program Files\Git\bin\git.exe

__17. Click **Save**.

Part 2 - Review

In this lab you configured the Jenkins Continuous Integration Server.

Lab 2 - Create a Jenkins Job

In this lab you will create and build a job in Jenkins.

Jenkins supports several different types of build jobs. The two most commonly-used are the freestyle builds and the Maven 2/3 builds. The freestyle projects allow you to configure just about any sort of build job, they are highly flexible and very configurable. The Maven 2/3 builds understand the Maven project structure, and can use this to let you set up Maven build jobs with less effort and a few extra features.

At the end of this lab you will be able to:

1. Create a Jenkins Job that accesses a Git repository.

Part 1 - Enable Jenkins' Maven Plugin

__1. Go to the Jenkins console:

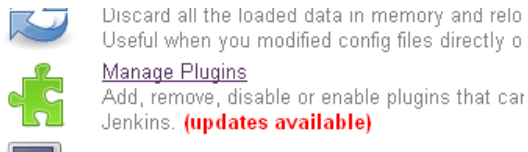
`http://localhost:8080`

__2. Enter **wasadmin** as user and password and click **Log in**.

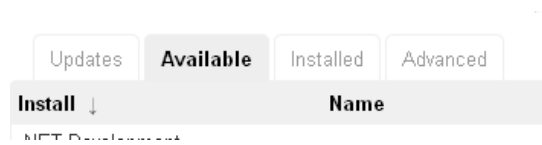
__3. Click on the **Manage Jenkins** link



__4. Click on **Manage Plugins**



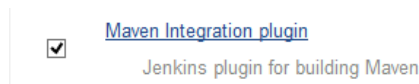
__5. Click on the **Available** tab.



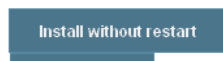
__6. In the **filter** box at the top-right of the window, enter 'maven'. Note: Don't hit **Return!**



__7. Entering the filter text will narrow down the available plugins a little. Scroll down to find the '**Maven Integration**' listing, and click on the checkbox next to it.



__8. Click on **Install Without Restart**.



__9. Click on **Go back to the top page**.

Installing Plugins/Upgrades

Preparation


- Checking internet connectivity
- Checking update center connectivity
- Success

Javadoc Plugin

Success

Maven Integration plugin

Success

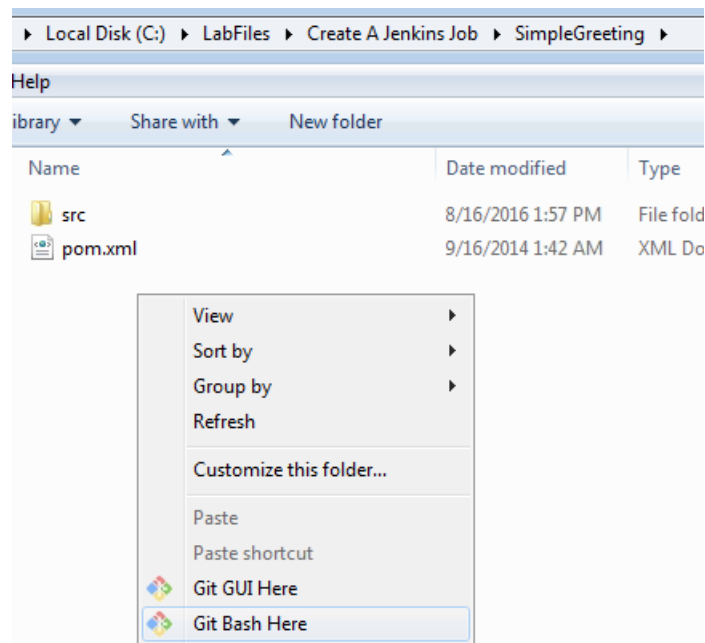
 [Go back to the top page](#)
(you can start using the installed plugins right away)

Part 2 - Create a Git Repository

As a distributed version control system, Git works by moving changes between different repositories. Any repository apart from the one you're currently working in is called a "remote" repository. Git doesn't differentiate between remote repositories that reside on different machines and remote repositories on the same machine. They're all remote repositories as far as Git is concerned.

In this lab, we're going to start from a source tree, create a local repository in the source tree, and then clone it to a local repository. Then we'll create a Jenkins job that pulls the source files from that remote repository. Finally, we'll make some changes to the original files, commit them and push them to the repository, showing that Jenkins automatically picks up the changes.

- ___1. Open to the folder **C:\LabFiles\Create A Jenkins Job\SimpleGreeting**
- ___2. Right click in the empty area and select **Git Bash Here**. The Git command prompt will open.



- ___3. Enter the following command:

ls

```
wasadmin@WIN-GRFMN8J26PD /C:/LabFiles/Create A Jenkins Job/SimpleGreeting
$ ls
pom.xml  src
wasadmin@WIN-GRFMN8J26PD /C:/LabFiles/Create A Jenkins Job/SimpleGreeting
$ -
```

__4. Enter the following lines. Press enter after each line:

```
git config --global user.email "wasadmin@fenago.com"
git config --global user.name "Bob Smith"
```

The lines above are actually part of the initial configuration of Git. Because of Git's distributed nature, the user's identity is included with every commit as part of the commit data. So we have to tell Git who we are before we'll be able to commit any code.

__5. Enter the following lines to actually create the Git repository:

```
git init
git add .
git commit -m "Initial Commit"
```

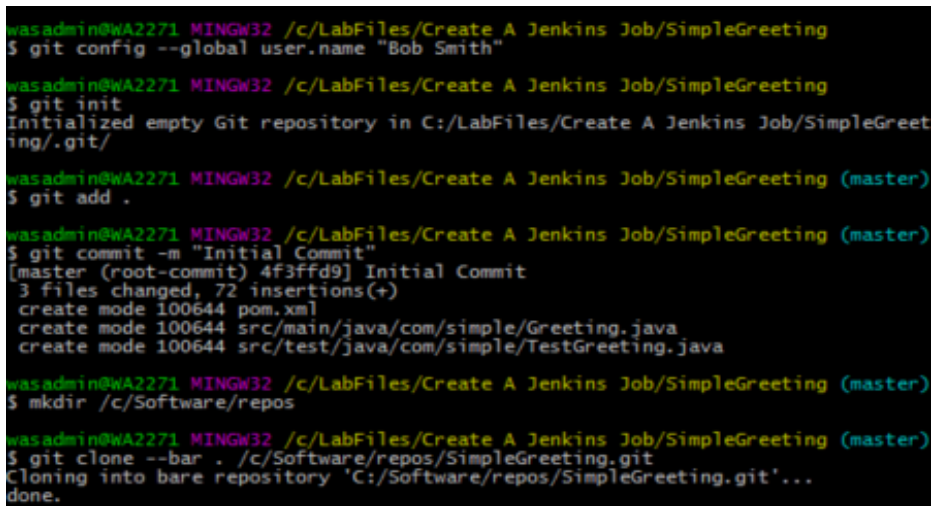
The above lines create a git repository in the current directory (which will be **C:\LabFiles\Create a Jenkins Job\SimpleGreeting**), add all the files to the current commit set (or 'index' in git parlance), then actually performs the commit.

__6. Enter the following, to create a folder called **repos** under the C:\Software folder.

```
mkdir /c/Software/repos
```

__7. Enter the following to clone the current Git repository into a new remote repository.

```
git clone --bar . /c/Software/repos/SimpleGreeting.git
```



```
wasadmin@WA2271 MINGW32 /c/LabFiles/Create A Jenkins Job/SimpleGreeting
$ git config --global user.name "Bob Smith"

wasadmin@WA2271 MINGW32 /c/LabFiles/Create A Jenkins Job/SimpleGreeting
$ git init
Initialized empty Git repository in C:/LabFiles/Create A Jenkins Job/SimpleGreeting/.git/

wasadmin@WA2271 MINGW32 /c/LabFiles/Create A Jenkins Job/SimpleGreeting (master)
$ git add .

wasadmin@WA2271 MINGW32 /c/LabFiles/Create A Jenkins Job/SimpleGreeting (master)
$ git commit -m "Initial Commit"
[master (root-commit) 4f3ffd9] Initial Commit
3 files changed, 72 insertions(+)
 create mode 100644 pom.xml
 create mode 100644 src/main/java/com/simple/Greeting.java
 create mode 100644 src/test/java/com/simple/TestGreeting.java

wasadmin@WA2271 MINGW32 /c/LabFiles/Create A Jenkins Job/SimpleGreeting (master)
$ mkdir /c/Software/repos

wasadmin@WA2271 MINGW32 /c/LabFiles/Create A Jenkins Job/SimpleGreeting (master)
$ git clone --bar . /c/Software/repos/SimpleGreeting.git
Cloning into bare repository 'C:/Software/repos/SimpleGreeting.git'...
done.
```

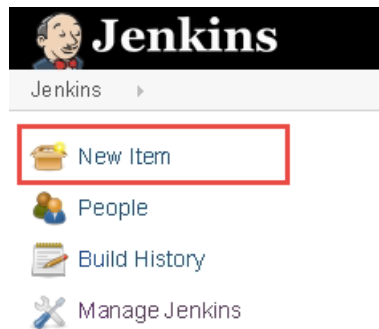
At this point, we have a "remote" Git repository in the folder **C:\Software\repos\SimpleGreeting.git**. Jenkins will be quite happy to pull the source files for a job from this repo.

Part 3 - Create the Jenkins Job

__1. Go to the Jenkins console:

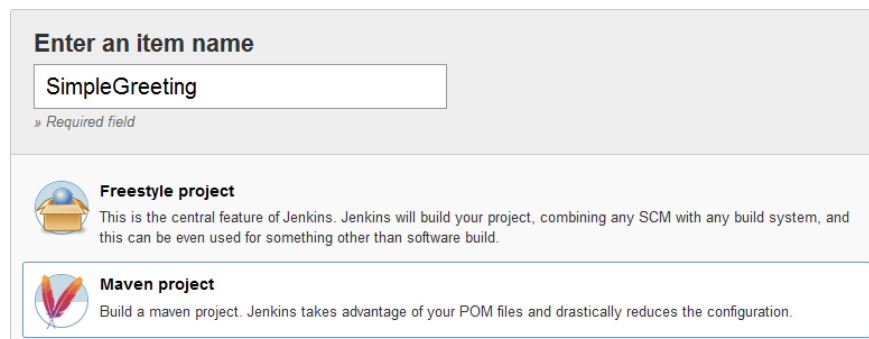
`http://localhost:8080`

__2. Click on the **New Item** link.



__3. Enter **SimpleGreeting** for the project name.

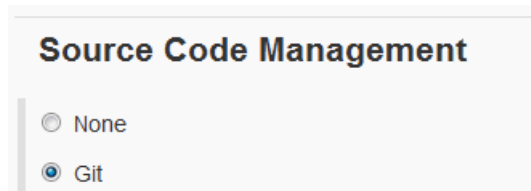
__4. Select **Maven Project** as the project type.

A screenshot of the Jenkins "Enter an item name" form. The form has a title "Enter an item name" and a text input field containing "SimpleGreeting". Below the input field is a small note "» Required field". Underneath the input field, there are two options for project types. The first option is "Freestyle project" with a folder icon and a description: "This is the central feature of Jenkins. Jenkins will build your project, combining any SCM with any build system, and this can be even used for something other than software build." The second option is "Maven project" with a feather icon and a description: "Build a maven project. Jenkins takes advantage of your POM files and drastically reduces the configuration." The "Maven project" option is highlighted with a blue border.

__5. Click **OK**, to add a new job.

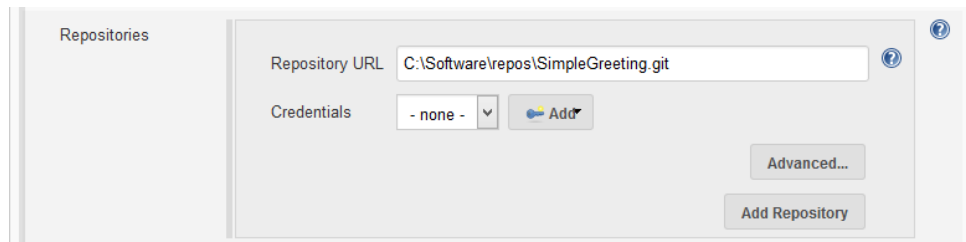
After the job is created, you will be on the job configuration page.

__6. Scroll down to the **Source Code Management** section and then select **Git**.



__7. Under Repositories, enter **C:\Software\repos\SimpleGreeting.git**

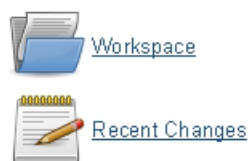
__8. Click the Tab key in your keyboard.



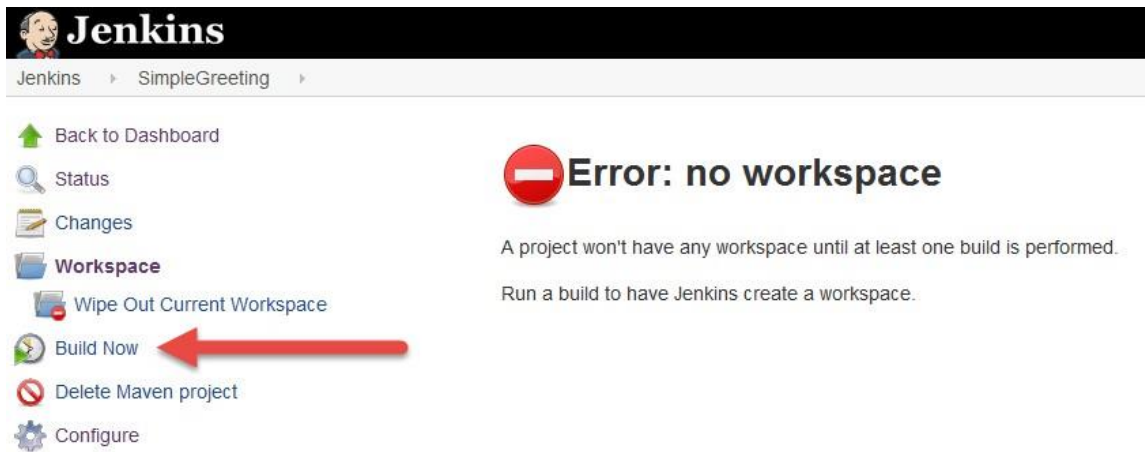
__9. Click **Save**.

__10. You will see the Job screen. Click **Workspace**.

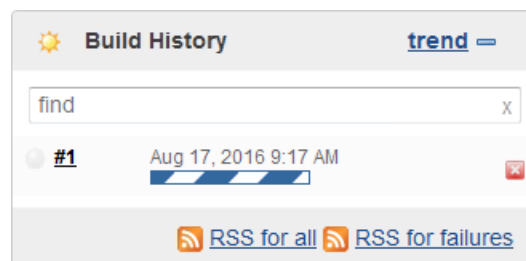
Maven project SimpleGreeting



__11. Click **Build Now**.



You should see the build in progress in the **Build History** area.

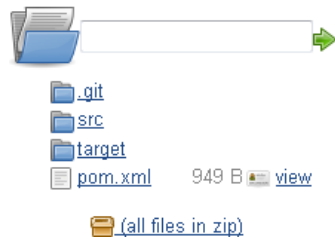


__12. After a few seconds the build will complete, the progress bar will stop. Click on **Workspace**.



You will see that the directory is populated with the source code for our project.

Workspace of SimpleGreeting on master



__13. Find the **Build History** box, and click on the 'time' value for the most recent build. You should see that the build was successful.

Build #1 (Aug 17, 2016 9:17:22 AM)



No changes.



Started by user [Administrator](#)



Revision: 4f3ffd9329c0bb74c5355effc53800df4ab2575a

- refs/remotes/origin/master

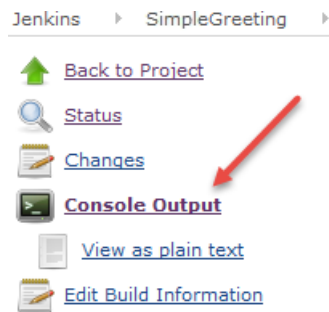


[Test Result](#) (no failures)

Module Builds

 [SimpleGreeting](#) 20 sec

__14. Click the **Console Output** from the left menu.



__15. At the end of the console you will also see the build success and successful build finish.

```
[INFO] Connected: http://localhost:8080/jenkins/
[INFO] Installing C:\Program Files\Jenkins\workspace\SimpleGreeting\target\S
\SimpleGreeting\1.0-SNAPSHOT\SimpleGreeting-1.0-SNAPSHOT.jar
[INFO] Installing C:\Program Files\Jenkins\workspace\SimpleGreeting\pom.xml
1.0-SNAPSHOT.pom
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 20.554 s
[INFO] Finished at: 2016-08-17T09:17:51-04:00
[INFO] Final Memory: 12M/46M
[INFO] -----
[JENKINS] Archiving C:\Program Files\Jenkins\workspace\SimpleGreeting\pom.xml
[JENKINS] Archiving C:\Program Files\Jenkins\workspace\SimpleGreeting\target
channel stopped
Finished: SUCCESS
```

You have created a project and built it successfully.

Part 4 - Enable Polling on the Repository

So far, we have created a Jenkins job that pulls a fresh copy of the source tree prior to building. But we triggered the build manually. In most cases, we would like to have the build triggered automatically whenever a developer makes changes to the source code in the version control system.

__1. In the Jenkins web application, navigate to the **SimpleGreeting** project. You can probably find the project in the breadcrumb trail near the top of the window. Alternately, go to the Jenkins home page and then click on the project.

__2. Click the **Configure** link.

__3. Scroll down to find the **Build Triggers** section.



__4. Click on the checkbox next to **Poll SCM**, and then enter '* * * * *' into the **Schedule** text box. [Make sure there is a space between each *]



Note: The above schedule sets up a poll every minute. In a production scenario, that's a higher frequency than we need, and it can cause unnecessary load on the repository server and on the Jenkins server. You'll probably want to use a more reasonable schedule - perhaps every 15 minutes. That would be 'H/15 * * * *' in the schedule box.

__5. Click **Save**.

Part 5 - Import the Project into Eclipse

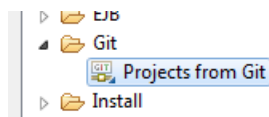
In order to make changes to the source code, we'll clone a copy of the Git repository into an Eclipse project.

__1. Start Eclipse by running **C:\Software\eclipse\eclipse.exe** and use **C:\Workspace** as Workspace.

__2. Close the Welcome page.

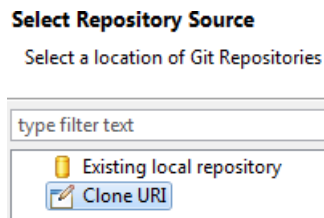
__3. From the main menu, select **File** → **Import...**

__4. Select **Git** → **Projects from Git**.



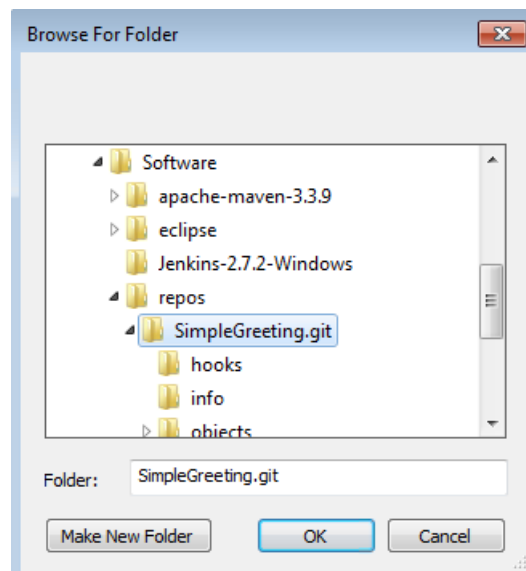
__5. Click **Next**.

__6. Select **Clone URI** and then click **Next**.



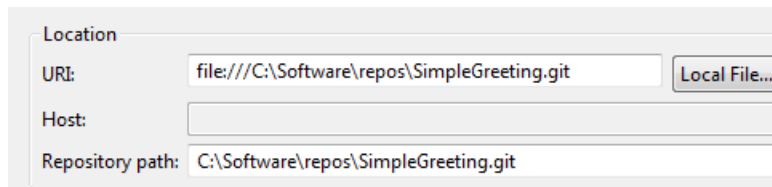
You might think that 'Existing local repository' would be the right choice, since we're cloning from a folder on the same machine. Eclipse, however, expects a "local repository" to be a working directory, not a bare repository. On the other hand, Jenkins will complain if we try to get source code from a repository with a working copy. So the correct thing is to have Jenkins pull from a bare repository, and use **Clone URI** to have Eclipse import the project from the bare repository.

__7. Click on **Local File...** and then navigate to **C:\Software\repos\SimpleGreeting.git**



__8. Click **OK**.

__9. Back in the **Import Projects** dialog, click **Next**.



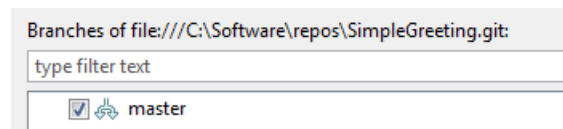
Location

URI: file:///C:/Software/repos/SimpleGreeting.git Local File...

Host:


Repository path: C:/Software/repos/SimpleGreeting.git

__10. Click **Next** to accept the default 'master' branch.

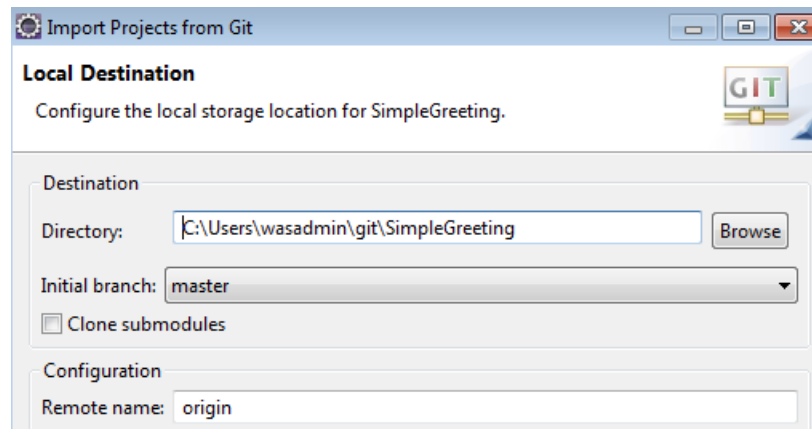


Branches of file:///C:/Software/repos/SimpleGreeting.git:

type filter text

☒  master

__11. In the **Local Destination** pane, leave the defaults and click **Next**.



Import Projects from Git

Local Destination

Configure the local storage location for SimpleGreeting.

Destination

Directory: C:/Users/wasadmin/git/SimpleGreeting Browse

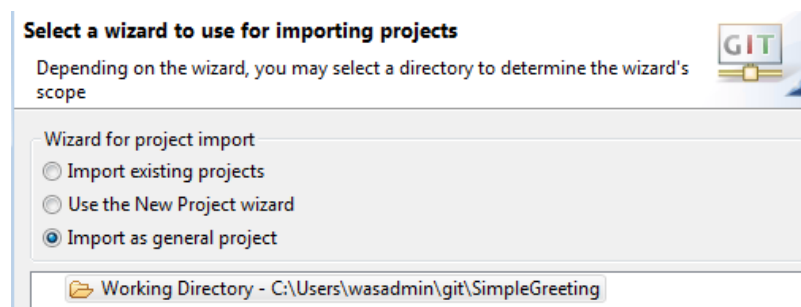
Initial branch: master

☐ Clone submodules

Configuration

Remote name: origin

__12. Select **Import as a General Project** and click **Next**.



Select a wizard to use for importing projects

Depending on the wizard, you may select a directory to determine the wizard's scope

Wizard for project import

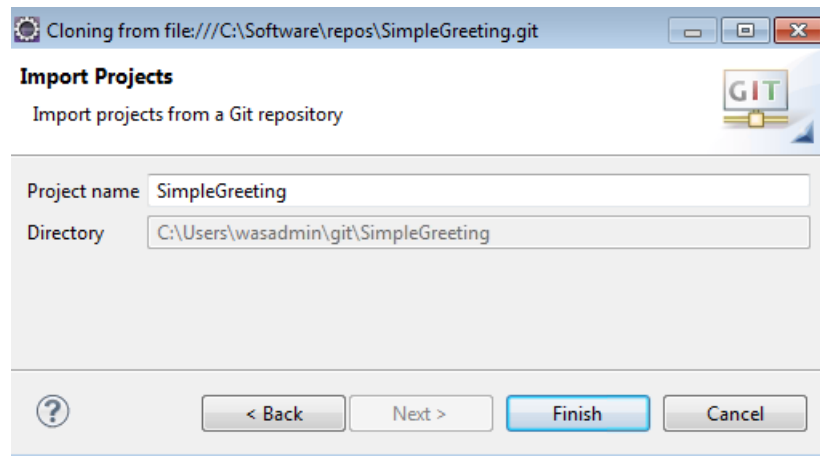
☐ Import existing projects

☐ Use the New Project wizard

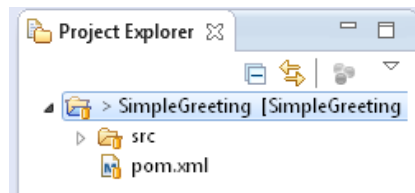
☒ Import as general project

Working Directory - C:/Users/wasadmin/git/SimpleGreeting

__13. Click **Finish**.

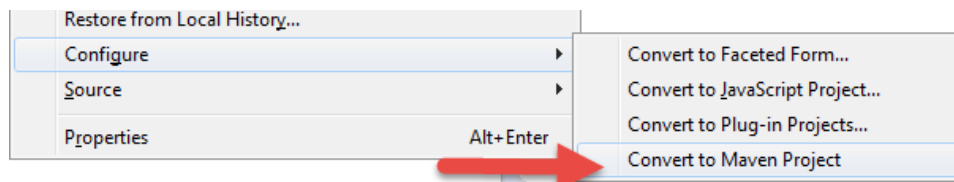


__14. You should see the new project in the **Project Explorer**, expand it.

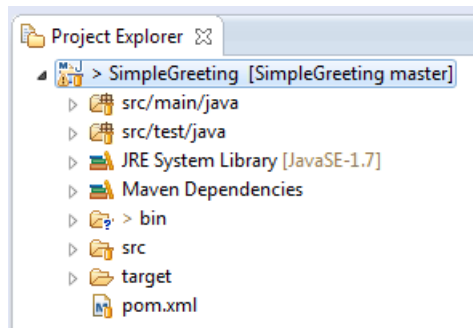


At this point, we could go ahead and edit the files, but Eclipse doesn't understand the project's layout. Let's tell Eclipse what we know - that this project is built using Apache Maven.

__15. Right-click on the **SimpleGreeting** project in the **Project Explorer**, and then select **Configure** → **Convert to Maven Project**.



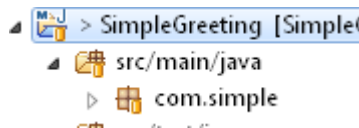
__16. After few seconds, you should now see the project represented as a Maven project in the **Project Explorer**.



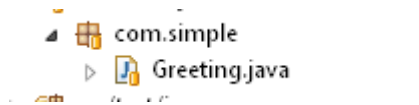
Part 6 - Make Changes and Trigger a Build

The project that we used as a sample consists of a basic "Hello World" style application, and a unit test for that application. In this section, we'll alter the core application so it fails the test, and then we'll see how that failure appears in Jenkins.

__1. In the **Project Explorer**, expand the **src/main/java** tree node.



__2. Expand the **com.simple** package to reveal the **Greeting.java** file.



__3. Double-click on **Greeting.java** to open the file.

__4. Find the line that says 'return "GOOD";'. Edit the line to read 'return "BAD";'

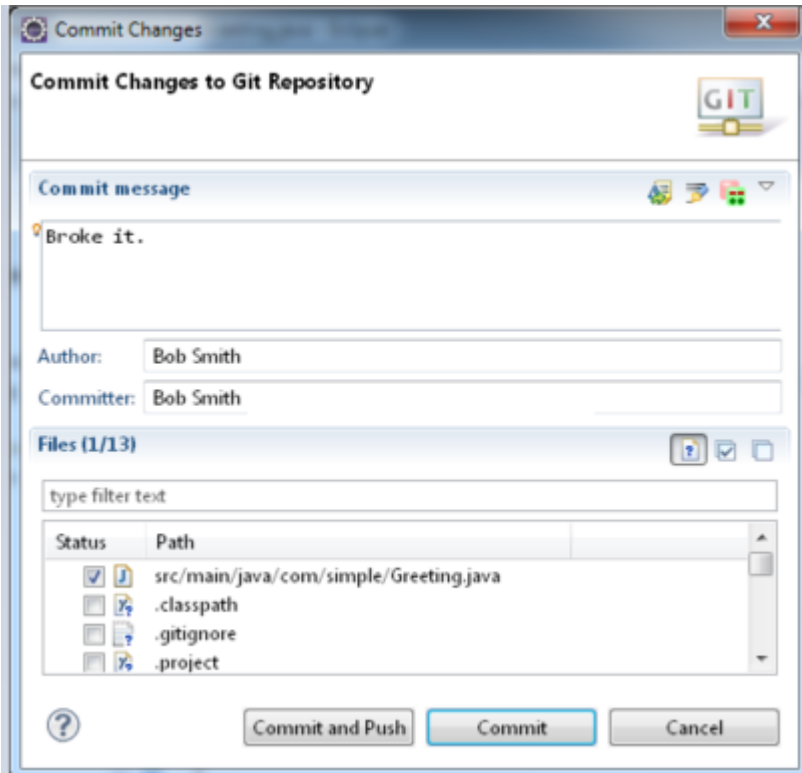
```
public String getStatus(){  
    return "BAD";  
}
```

__5. Save the file by pressing Ctrl-S or selecting **File** → **Save**.

Now we've edited the local file. The way Git works is that we'll first 'commit' the file to the local workspace repository, and then we'll 'push' the changes to the upstream repository. That's the same repository that Jenkins is reading from. Eclipse has a short-cut button that will commit and push at the same time.

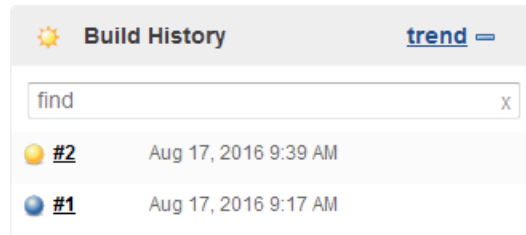
__6. Right-click on **SimpleGreeting** in the **Project Explorer** and then select **Team** → **Commit...**

__7. Enter a few words as a commit message, and then click **Commit and Push**.



__8. Click **OK** in the status dialog that pops up.

__9. Now, flip back to the web browser window that we had Jenkins running in. If you happen to have closed it, open a new browser window and navigate to <http://localhost:8080/SimpleGreeting>. After a few seconds, you should see a new build start up.



__10. If you refresh the page, you should see that there is now a 'Test Result Trend' graph that shows we have a new test failure.



What happened is that we pushed the source code change to the Git repository that Jenkins is reading from. Jenkins is continually polling the repository to look for changes. When it saw that a new commit had been performed, Jenkins checked out a fresh copy of the source code and performed a build. Since Maven automatically runs the unit tests as part of a build, the unit test was run. It failed, and the failure results were logged.

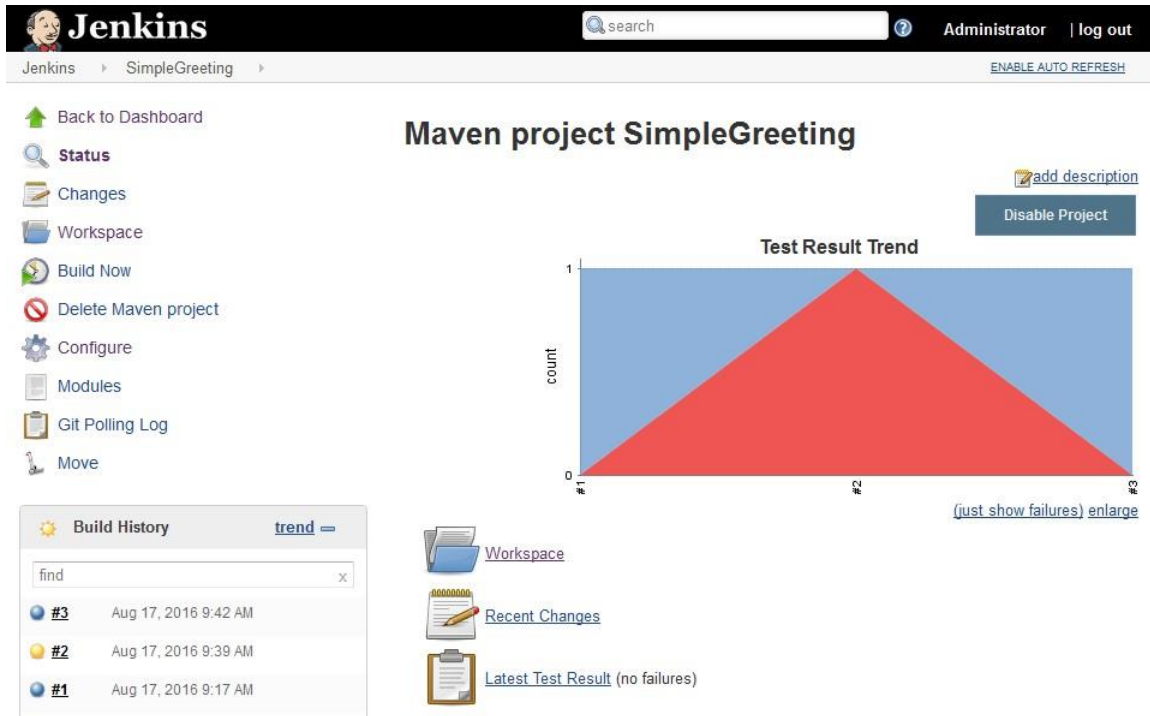
Part 7 - Fix the Unit Test Failure

__1. Back in eclipse , edit the file **Greeting.java** so that the class once again returns 'GOOD'.

```
public String getStatus(){  
    return "GOOD";  
}
```

__2. As above, save, commit and push the change.

__3. Watch the Jenkins web browser window. After a minute or two you should see the build start automatically, when the build is done then refresh the page.



Part 8 - Review

In this lab you learned

- How to Set-up a set of distributed Git repositories
- How to create a Jenkins Job that reads from a Git repository
- How to configure Jenkins to build automatically on source code changes.

Lab 3 - Advanced Jobs

In this lab, you will create parameterized Jenkins and Multi-configuration jobs.

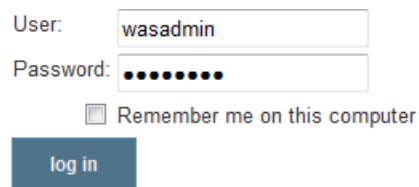
Part 1 - Connect to Jenkins

In this part you will connect to Jenkins.

__1. To connect to Jenkins, open Chrome / Firefox and enter the following URL.

`http://localhost:8080/`

__2. Enter **wasadmin** as user and password and click **Log in**.

A screenshot of the Jenkins login page. It features two input fields: 'User:' with the text 'wasadmin' and 'Password:' with masked characters. Below these is a checkbox labeled 'Remember me on this computer'. At the bottom is a blue button labeled 'log in'.

__3. Don't save the password if prompt or select Never Remember password for this site.

Part 2 - Create a new Maven Project-based Jenkins Job

In this part, you will create a new Maven Project-based Jenkins job.

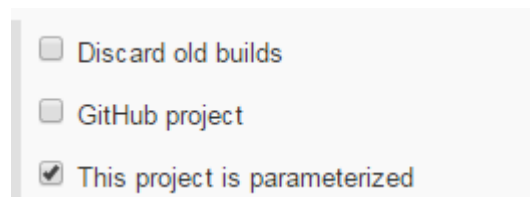
__1. On left side of the page, click **New Item**.

__2. Enter "ParamTest", without quotes, in the **Enter an item name** text box.

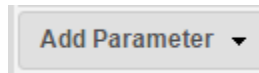
__3. Select **Maven project** as project type.

__4. Click **OK**.

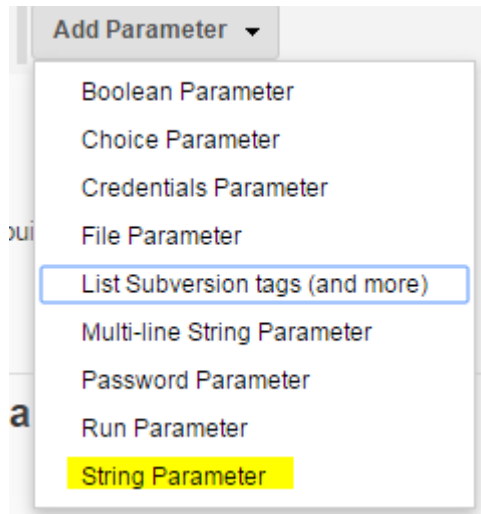
__5. Check the box in front of **This project is parameterized**.

A screenshot of the Jenkins job configuration page. It shows three checkboxes: 'Discard old builds', 'GitHub project', and 'This project is parameterized'. The 'This project is parameterized' checkbox is checked.

__6. Click **Add Parameter**.



__7. Select **String Parameter**.



__8. Enter the following values in the fields:

Name: FileName
Default Value: SimpleGreeting
Description: This is the file name

String Parameter

Name:

Default Value:

Description:

[Plain text] [Preview](#)

Add Parameter ▾

__9. Click **Add Parameter** again.

Add Parameter ▾

__10. Select **Choice Parameter**.

Boolean Parameter

Choice Parameter

Credentials Parameter

File Parameter

List Subversion tags (and more)

Multi-line String Parameter

Password Parameter

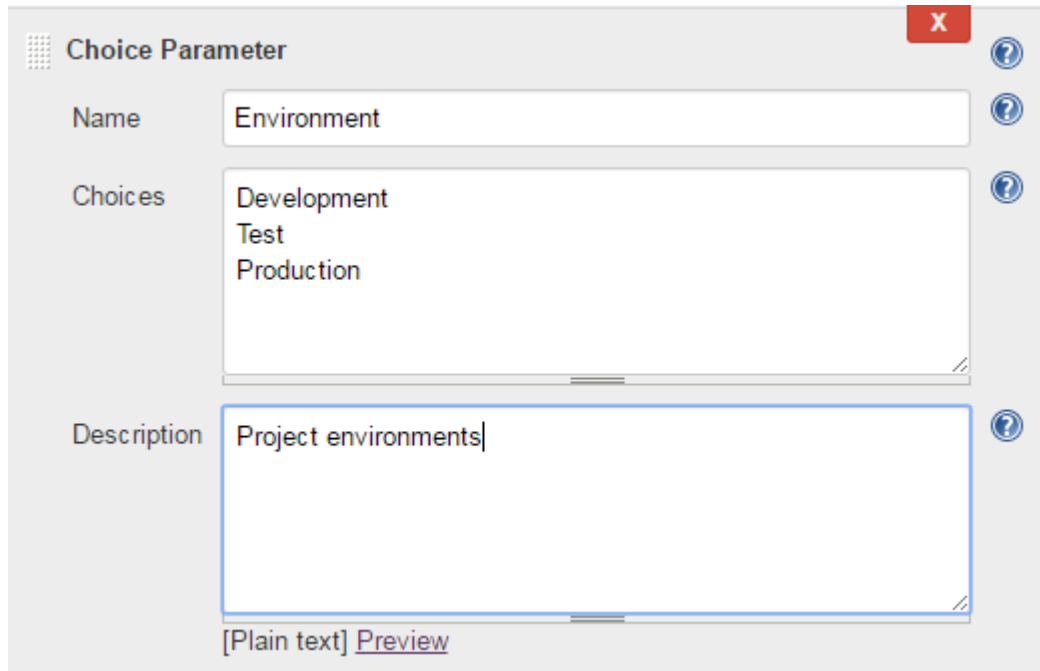
Run Parameter

String Parameter

Add Parameter ▾

__11. Enter the following values in the fields.

Name: Environment
Choices: Development
 Test
 Production
Description: Project environments



Choice Parameter

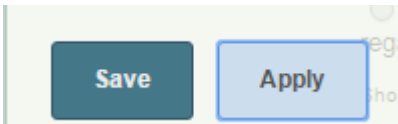
Name: Environment

Choices: Development
Test
Production

Description: Project environments

[Plain text] [Preview](#)

__12. Click **Apply** button.



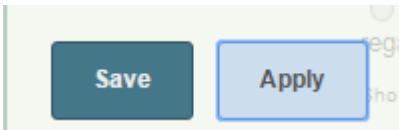
__13. Scroll through the web page and navigate to **Source Code Management** section.

__14. Select **Git** radio button.

__15. In **Repository URL**, enter C:\Software\repos\SimpleGreeting.git

__16. Press Tab.

__17. Click **Apply** button.



__18. Scroll through the page and locate **Build** section.

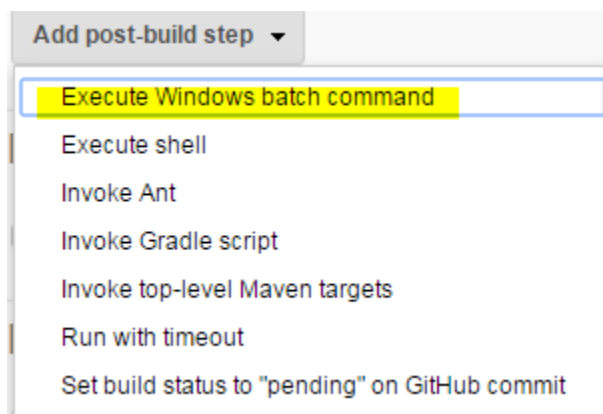
__19. In **Goals and options**, enter **install**

__20. Click **Apply** button.

__21. Scroll through the page and locate **Post Steps** section.

__22. Click **Add post-build step**.

__23. Select **Execute Windows batch command**.

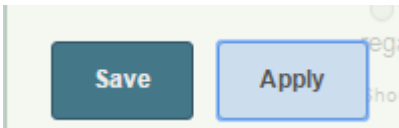


__24. Enter the following command.

```
xcopy /S /R /I /Y "%WORKSPACE%\target\SimpleGreeting-1.0-SNAPSHOT.jar"  
"c:\software\%Environment%\%FileName%*"
```

Note: Don't forget the * at the end of %FileName%. It tells xcopy to copy it as a file as opposed to copying it as a directory.

__25. Click the **Save** button.



Part 3 - Run and Test the Job

In this part, you will run the job you created in the previous part of this lab.

__1. On the left side of the page, click **Build with Parameters**.

__2. Enter the following parameter values:

FileName: SimpleGreeting-prod.jar
Environment: Production

Maven project ParamTest

This build requires parameters:

FileName
This is the file name

Environment Project environments

__3. Click the **Build** button.

__4. While the build is running, click the drop down for the build and click **Console Output**.

__5. Scroll through the console output and notice a message similar to this, near the end:

```
C:\Program Files (x86)\Jenkins\workspace\ParamTest>xcopy /S /R /I /Y "C:\Program Files (x86)\Jenkins\workspace\ParamTest\target\SimpleGreeting-1.0-SNAPSHOT.jar" "c:\software\Production\SimpleGreeting-prod.jar*"
C:\Program Files (x86)\Jenkins\workspace\ParamTest\target\SimpleGreeting-1.0-SNAPSHOT.jar
1 File(s) copied
```

__6. Open Windows Explorer and navigate to **C:\Software**.

Notice there's a directory named **Production**. Under the **Production** directory notice there's a file named **SimpleGreeting-prod.jar**

__7. Delete **C:\Software\Production** directory.

__8. In the web browser on the Jenkins page, click **Jenkins** to go back to the home page.

Part 4 - Create a Multi-Configuration project

In this part, you will create a multi-configuration project.

__1. On the left side of the page, click **New Item**.

__2. Enter "MultiConfigTest", without quotes, in the **Enter an item name** text box.

__3. Select **Multi-configuration project** as the project type.

__4. Click **OK**.

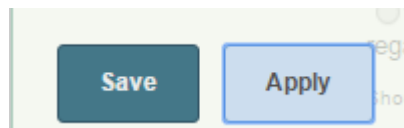
__5. Scroll through the web page and navigate to the **Source Code Management** section.

__6. Select the **Git** radio button.

__7. In **Repository URL**, enter C:\Software\repos\SimpleGreeting.git

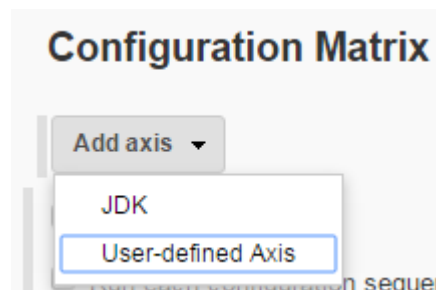
__8. Press tab.

__9. Click the **Apply** button.



__10. Scroll through the page and locate the **Configuration Matrix** section.

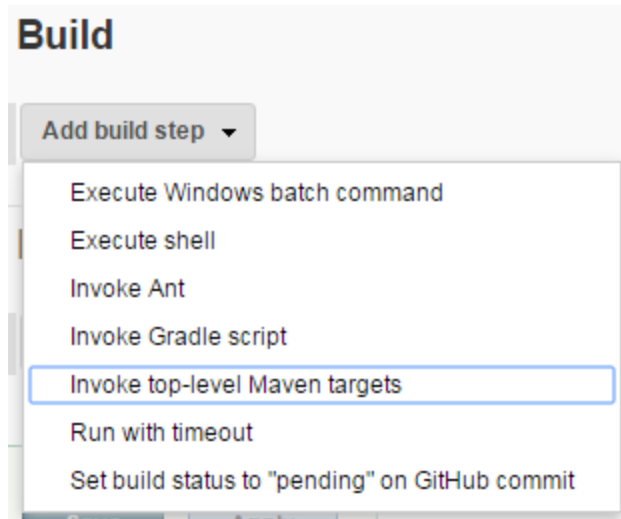
__11. Click **Add axis** and select **User-defined axis**.



__12. Enter the following values in the fields:

Name: Environments
Values: Development Test Production

- __13. Click **Apply** button.
- __14. Scroll through the page and locate the **Build** section.
- __15. Click **Add build step** and then click **Invoke top-level Maven targets**.



- __16. In **Maven Version**, select Maven (Note: You might have different name).
- __17. In **Goals** enter **install**
- __18. Click the **Apply** button.
- __19. Click the **Save** button.

Part 5 - Run and Test the Multi-Configuration Project

In this part, you will run and test the project you created in the previous part of this lab.

- __1. On the left side of the page, click **Build Now**.
- __2. At the bottom of the page, notice that there are 3 entries showing up like this.

Project MultiConfigTest

Configurations



- __3. Wait for the build to complete for all configurations.
- __4. Using Windows Explorer, navigate to **workspace** path that is logged in build console.

Notice there's a directory named Environments (same as the user-axis name you previously created).

Note: Workspace will be similar to following. Check logs to confirm path.

C:\Windows\system32\config\systemprofile\AppData\Local\Jenkins\.jenkins\workspace\MultiConfigTest\Environments

- __5. Under the **Environment** directory, there are three sub-directories: Development, Production, Test.

- __6. Examine the contents of each directory.

Notice each directory has target\SimpleGreeting-*.jar file.

Part 6 - Review

In this lab, you created parameterized Jenkins and Multi-configuration jobs.

Lab 4 - Template Jobs

In this lab you will install Template Project plugin, create a template project, and utilize it in another Jenkins job.

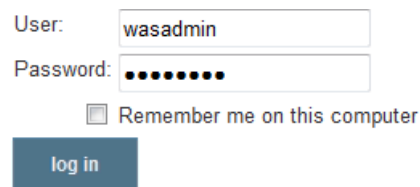
Part 1 - Connect to Jenkins

In this part you will connect to Jenkins.

__1. To connect to Jenkins, open Chrome / Firefox and enter the following URL.

`http://localhost:8080/`

__2. Enter **wasadmin** as user and password and click **Log in**.

A screenshot of the Jenkins login page. It features a 'User:' label next to a text input field containing 'wasadmin'. Below it is a 'Password:' label next to a password input field with masked characters. A checkbox labeled 'Remember me on this computer' is positioned below the password field. At the bottom is a blue 'log in' button.

__3. Don't save the password if prompt or select Never Remember password for this site.

Part 2 - Install Template Project and Multiple SCMs plugin

In this part you will install Template Project plugin in Jenkins.

__1. On left side of the page, click **Manage Jenkins**.

__2. Click **Manage Plugins**.

__3. Click **Available** tab.

__4. In **Filter** text box, enter "template" (without quotes).

__5. Click the check box in front of **Template Project plugin**.

__6. Click **Install without restart**.

Wait for the installation to complete.

__7. Go back to the **Available** tab.

__8. In **Filter** text box, enter "multiple" (without quotes).

__9. Click the check box in front of **Multiple SCMs plugin**.

__10. Click **Download now and install after restart**.

__11. Wait for the Download to complete then click the check box in front of **Restart Jenkins when installation is complete and no jobs are running**.

__12. After Jenkins restarts, enter wasadmin/wasadmin credentials to login in to Jenkins.

If Jenkins doesn't start then go to Services and restart it from there.

__13. Click **Jenkins** on top of the page to navigate to Jenkins home page.

Part 3 - Create a new Maven Project-based Jenkins Job

In this part you will create a new Maven Project-based Jenkins job which will act as a template.

__1. On left side of the page, click **New Item**.

__2. Enter "TemplateProject", without quotes, in **Enter an item name** text box.

__3. Select **Maven project** as project type.

__4. Click **OK**.

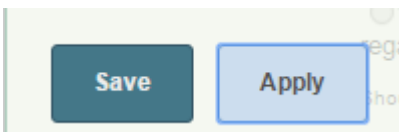
__5. Scroll through the web page and navigate to **Source Code Management** section.

__6. Select **Git** radio button.

__7. In **Repository URL**, enter following text and hit tab.

```
C:\Software\repos\SimpleGreeting.git
```

__8. Click **Apply** button.



__9. Scroll through the page and locate **Post-build Actions** section.

__10. Click **Add post-build action**.

__11. Click **Archive the artifacts**.

__12. In **Files to archive** enter following.

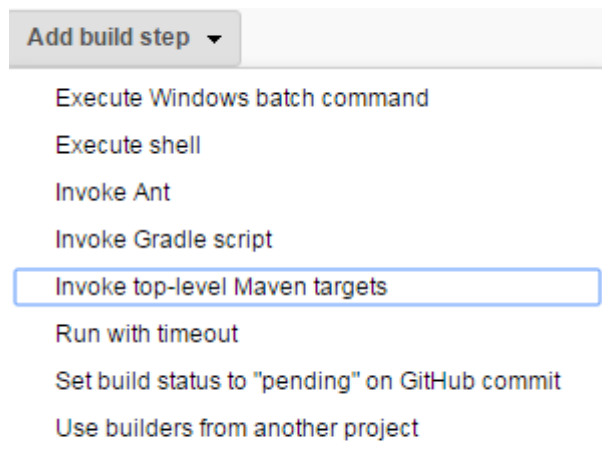
```
target/*.jar
```

__13. Click **Save** button.

Part 4 - Create a project which uses the template project

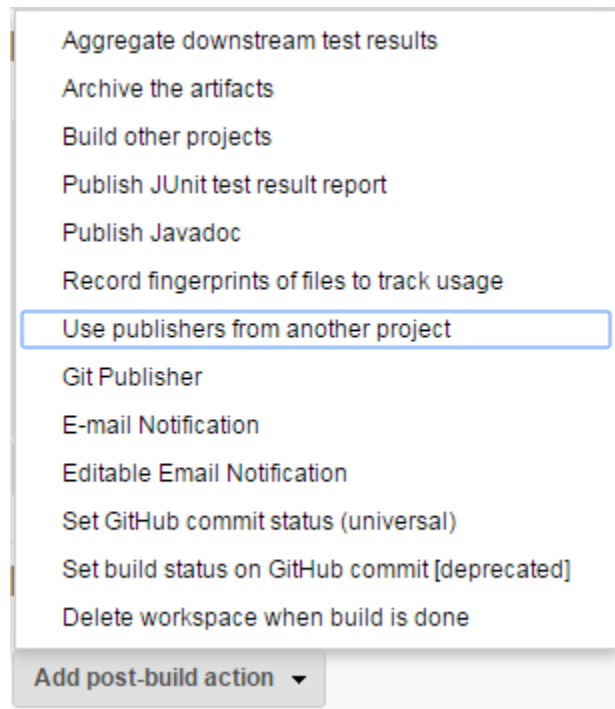
In this part you will create another job which utilizes the template you created in the previous part of this lab.

- __1. On top of the page, click **Jenkins** to go to Jenkins home page.
- __2. On left side of the page, click **New Item**.
- __3. In **Enter an item name**, enter "SampleProject" (without quotes).
- __4. Select **Freestyle project**.
- __5. Click **OK** button.
- __6. Scroll through the page and locate **Source Code Management** section .
- __7. Select **Use SCM from another project**.
- __8. In **Template Project** enter **TemplateProject**
- __9. Click **Apply** button.
- __10. Scroll through the page and locate **Build** section.
- __11. Click **Add build step**.
- __12. Click **Invoke top-level Maven targets**.



- __13. In **Goals** enter **clean install**
- __14. Click **Apply** button.
- __15. Scroll through the page and locate **Post-build Actions**.
- __16. Click **Add post-build action**.

__17. Click **Use publishers from another project**.



__18. In **Template Project** enter **TemplateProject**

__19. Click **Save** button.

Part 5 - Run and Test the SampleProject

In this part you will run and test the job you created in the previous part of this lab.

__1. On left side of the page, click **Build Now**.

Wait for the build to complete.

__2. Click drop down for the completed build in the **Build History**.

__3. Click **Console Output**.

__4. Notice at the bottom of the console it built the project using TemplateProject as the template.

```
[TemplateProject] Starting publishers from: TemplateProject  
Archiving artifacts  
[TemplateProject] Successfully performed publishers from: 'TemplateProject'  
Finished: SUCCESS
```

__5. Using Windows Explorer navigate to path logged in jenkins build console:

`UpdatePath\jobs\SampleProject\builds`.

Notice there's a directory for the build you just performed.

__6. Navigate to the directory containing your latest build.

__7. Navigate to **archive** directory.

__8. Navigate to **target** directory.

Notice SimpleGreeting-*.jar file is archived in this directory.

__9. Close all.

Part 6 - Review

In this lab you installed Template Project plugin, created a template project, and utilized it in another Jenkins job.

Lab 5 - Node.js based Jenkins Job and Manual Approval

In this lab you will create a Jenkins job based on a Node.js application that is provided in a Git repository in the LabFiles folder. The job will run tests, publish test results using Jenkins.

Part 1 - Create a Jenkins job and add Node.js related steps to it

In this part you will create a Jenkins job and add steps to it. The steps will run Node.js application unit tests and generate test report.

__1. In the web browser, ensure you are on Jenkins web page.

`http://localhost:8080`

__2. Enter **wasadmin** as user and password and click **Log in**.

__3. Click **New Item** link.

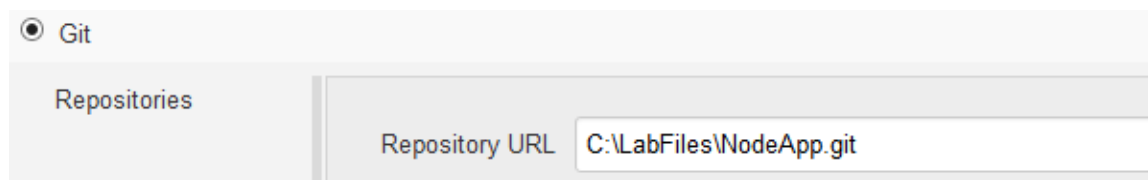
__4. In **Enter an item name** enter NodeApp

__5. Select **Freestyle project**.

__6. Click **OK**.

__7. Under **Source Code Management**, click the radio button for **Git**.

__8. Enter 'C:\LabFiles\NodeApp.git' as the repository URL.



The screenshot shows the Jenkins configuration page for Source Code Management. The 'Git' radio button is selected. Below it, the 'Repository URL' field is populated with 'C:\LabFiles\NodeApp.git'.

__9. Scroll down the page and locate **Build** section.

__10. Click **Add build step** and in the drop down click **Execute Windows batch command**.

__11. In the **Command** text box enter following command:

```
npm install
```

Node modules are stored in the 'node_modules' folder, but we don't want those files in source code control - the repository would be huge if we put them in. So, there is an entry in '.gitignore' that bypasses 'node_modules'. The 'npm install' command fetches and installs all the modules that are called out in 'package.json'.

__12. Click **Apply** button.

__13. In the **Build** section, click **Add build step** and click **Execute Windows batch command**.

__14. In the **Command** text box enter following command (the two lines that start with '..\node...' should be entered as one line in the text box).

```
cd tests
```

```
..\node_modules\.bin\mocha --reporter mocha-junit-reporter --reporter-  
options mochaFile="%WORKSPACE%\test-results.xml"
```

Notice the command switches to the tests directory and runs mocha. The binary for mocha is stored under the 'node_modules' directory at the top level of the workspace. It passes various arguments to generate test the results file that will be picked up by Jenkins' JUnit test report plugin.

__15. In the **Post-build Actions** section click **Add post-build action** and select **Publish JUnit test result report**.

__16. In the **Test report XMLs** text box enter following text.

```
test-results.xml
```

__17. Click **Save** button.

__18. Click **Build Now**.

__19. In **Build History** notice there's one entry with blue colored icon. Click it.

__20. Click **Console Output**.

Notice everything was executed successfully.

__21. Click **Test Result**.

Notice there are 0 failures.

__22. Under **All Tests** click (root).

All Tests

Package
(root)

__23. Click "returns a hello message".

Notice "returns a hello message" has **Passed** status.

__24. Click "The message module returns a hello message" link.

All Tests

Test name	Duration	Status
The message module returns a hello message	1 ms	Passed

Notice there are no error messages.

__25. In the web browser click **Back to Project**.

__26. Close all.

Part 2 - Review

In this lab you created a Node.js job, built it using Jenkins, and generated unit test results.

Lab 6 - Create a Pipeline

In this lab you will explore the Pipeline functionality.

At the end of this lab you will be able to:

1. Create a simple pipeline
2. Use a 'Jenkinsfile' in your project
3. Use manual input steps in a pipeline

Part 1 - Create a Simple Pipeline

We can create a pipeline job that includes the pipeline script in the job configuration, or the pipeline script can be put into a 'Jenkinsfile' that's checked-in to version control.

To get a taste of the pipeline, we'll start off with a very simple pipeline defined in the job configuration.

Prerequisite: We need to have a project in source control to check-out and build. For this example, we're using the 'SimpleGreeting' project that we previously cloned to a 'Git' repository at 'C:\Software\repos\SimpleGreeting.git'

__1. To connect to Jenkins, open Firefox and enter the following URL .

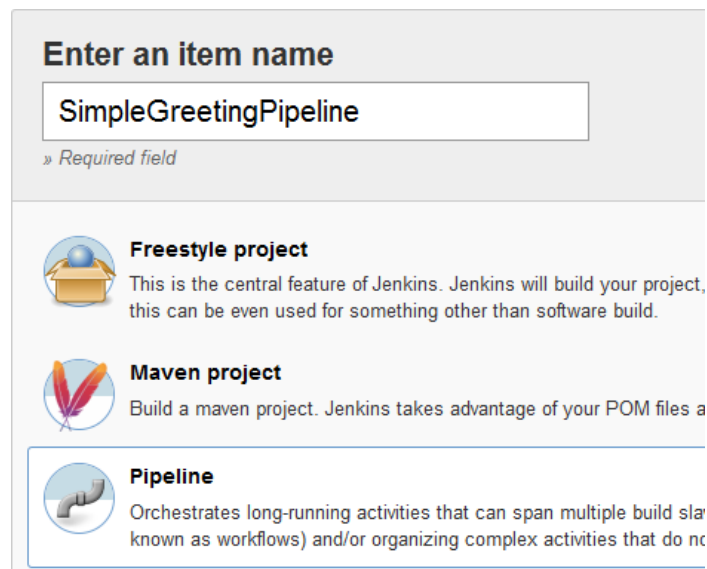
`http://localhost:8080/`

__2. Enter **wasadmin** as user and password and click **Log in**.

__3. Click on the **New Item** link.



___4. Enter 'SimpleGreetingPipeline' as the new item name, and select 'Pipeline' as the item type.



Enter an item name

SimpleGreetingPipeline

» Required field

Freestyle project
This is the central feature of Jenkins. Jenkins will build your project, this can be even used for something other than software build.

Maven project
Build a maven project. Jenkins takes advantage of your POM files a

Pipeline
Orchestrates long-running activities that can span multiple build slas known as workflows) and/or organizing complex activities that do nc

___5. When the input looks as above, click on **OK** to create the new item.

___6. Scroll down to the **Pipeline** section and enter the following in the **Script** text window.

```
node {  
    stage 'Checkout'  
    git url: 'C:\\Software\\repos\\SimpleGreeting.git'  
  
    stage 'Maven build'  
    bat 'mvn install'  
  
    stage 'Archive Test Results'  
    step([$class: 'JUnitResultArchiver',  
        testResults: '**/target/surefire-reports/TEST-*.xml'])  
}
```

This pipeline is divided into three stages. First, we checkout the project from our 'git' repository. Then we use the 'bat' command to run 'mvn install' as a Windows batch file. Finally, we use the 'step' command to utilize a step from a standard Jenkins plugin - in this case, the JUnitResultArchiver, to save and display the results of the unit tests.

All of the above is wrapped inside the 'node' command, to indicate that we want to run these commands in the context of a workspace running on one of Jenkins execution agents (or the master node if no agents are available).

___7. Click on **Save** to save the changes and return to the project page.

__8. Click on **Build Now** to startup a pipeline instance.



__9. After a few moments, you should see the **Stage View** appear, and successive stages will appear as the build proceeds, until all three stages are completed.

Stage View



Part 2 - Pipeline Definition in a 'Jenkinsfile'

For simple pipelines or experimentation, it's convenient to define the pipeline script in the web interface. But one of the common themes of modern software development is "If it isn't in version control, it didn't happen". The pipeline definition is no different, especially as you build more and more complex pipelines.

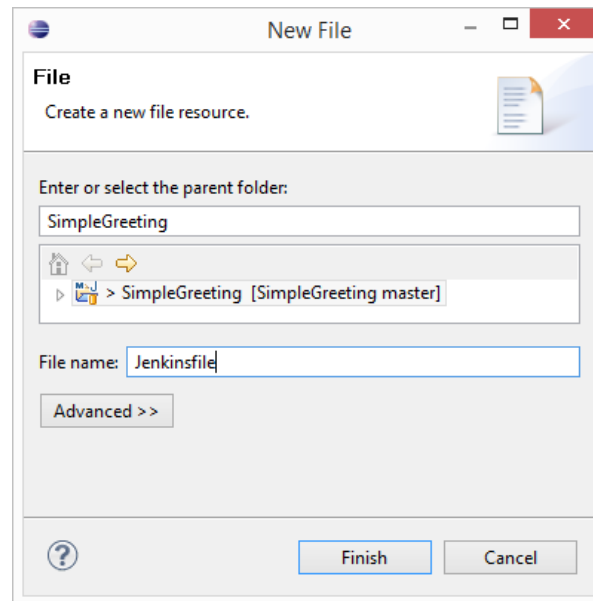
You can define the pipeline in a special file that is checked out from version control. There are several advantages to doing this. First, of course, is that the script is version-controlled. Second, we can edit the script with the editor or IDE of our choice before checking it in to version control. In addition, we can employ the same kind of "SCM Polling" that we would use in a more traditional Jenkins job.

In the following steps, we'll create a Jenkinsfile and create a pipeline job that uses it.

__1. Open the Eclipse editor. If this lab is completed in the normal sequence, you should have the 'SimpleGreeting' project already in Eclipse's workspace. If not, check out the project from version control (consult your instructor for directions if necessary).

__2. In the **Project Explorer**, right-click on the root node of the **SimpleGreeting** project, and then select **New** → **File**.

__3. Enter 'Jenkinsfile' as the file name.



__4. Click **Finish** to create the new file.

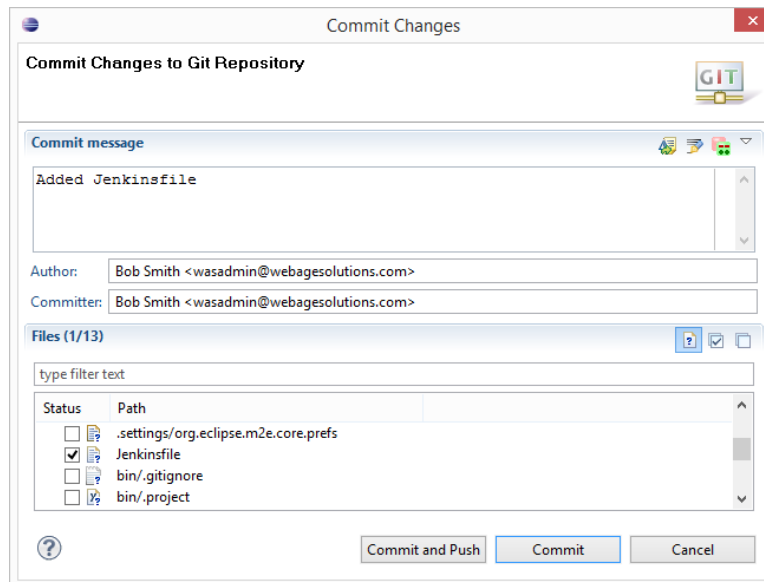
__5. Enter the following text into the new file (Note: this is the same script that we used above, so you could copy/paste it from the Jenkins Web UI if you want to avoid some typing):

```
node {  
    stage 'Checkout'  
    git url: 'C:\\Software\\repos\\SimpleGreeting.git'  
  
    stage 'Maven build'  
    bat 'mvn install'  
  
    stage 'Archive Test Results'  
    step([$class: 'JUnitResultArchiver',  
        testResults: '**/target/surefire-reports/TEST-*.xml'])  
}
```

__6. Save the Jenkinsfile by selecting **File** → **Save** from the main menu, or by hitting Ctrl-S.

__7. In the **Project Explorer**, right-click on the **SimpleGreeting** node, and then select **Team** → **Commit...**

__8. Eclipse will display the **Commit Changes** dialog. Click the checkbox next to **Jenkinsfile** (to include that file in the commit) and enter a commit message.

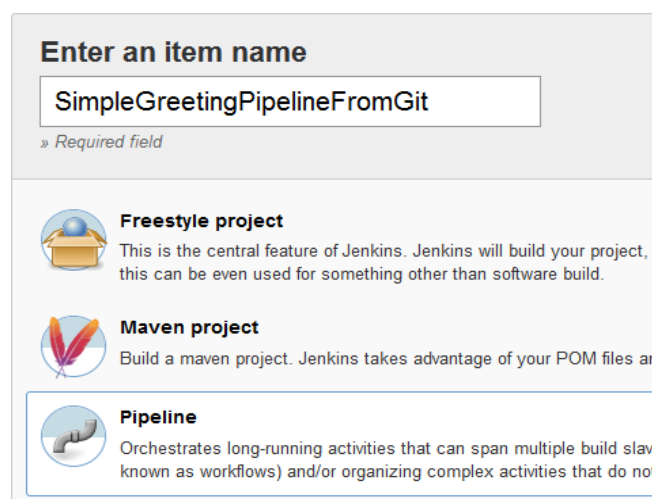


__9. Click **Commit and Push**, and then click **OK** to dismiss the status dialog.

Now we have a Jenkinsfile in our project, to define the pipeline. Next, we need to create a Jenkins job to use that pipeline.

__10. In the Jenkins user interface, navigate to the root page, and then click on **New Item**.

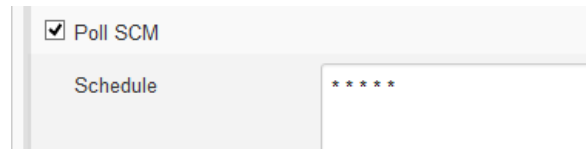
__11. Enter 'SimpleGreetingPipelineFromGit' as the name of the new item, and select **Pipeline** as the item type.



__12. Click **OK** to create the new item.

__13. Scroll down to the **Build Triggers** section.

__14. Click on **Poll SCM** and enter '* * * * *' as the polling schedule. This entry will cause Jenkins to poll once per minute.



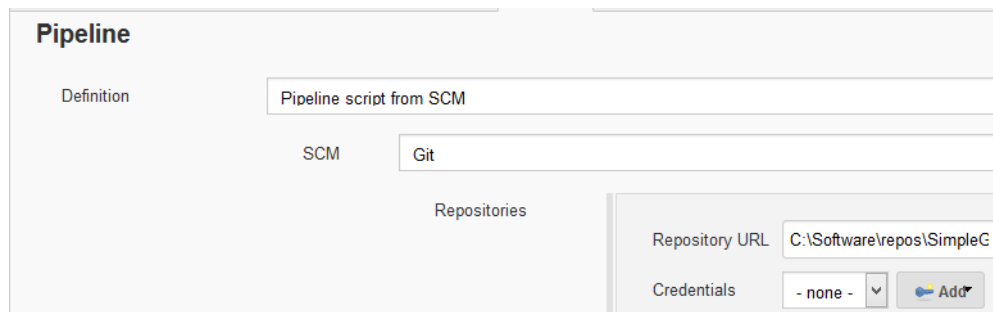
The screenshot shows a section of the Jenkins configuration interface. On the left, there is a vertical list of options, with 'Poll SCM' selected and checked. To the right of this list, there is a 'Schedule' label followed by a text input field containing the string '* * * * *'.

__15. Scroll down to the **Pipeline** section, and change the **Definition** entry to 'Pipeline Script from SCM'

__16. Enter the following:

SCM:	Git
Repository URL:	C:\Software\repos\SimpleGreeting.git

__17. Press tab. The **Pipeline** section should look similar to:



The screenshot shows the 'Pipeline' configuration section in Jenkins. It includes a 'Definition' field with the value 'Pipeline script from SCM', an 'SCM' dropdown menu set to 'Git', and a 'Repositories' section. The 'Repository URL' is set to 'C:\Software\repos\SimpleG' and the 'Credentials' dropdown is set to '- none -' with an 'Add' button next to it.

__18. Click **Save** to save the new configuration.

__19. Click **Build Now** to launch the pipeline.

__20. You should see the pipeline execute, similar to the previous section.

Part 3 - Try out a Failing Build

The pipeline that we've defined so far appears to work perfectly. But we haven't tested it with a build that fails. In the following steps, we'll insert a test failure and see what happens to our pipeline.

__1. In Eclipse, go to the **Project Explorer** and locate the file 'Greeting.java'. It will be under **src/main/java** in the package 'com.simple'. Open 'Greeting.java'.

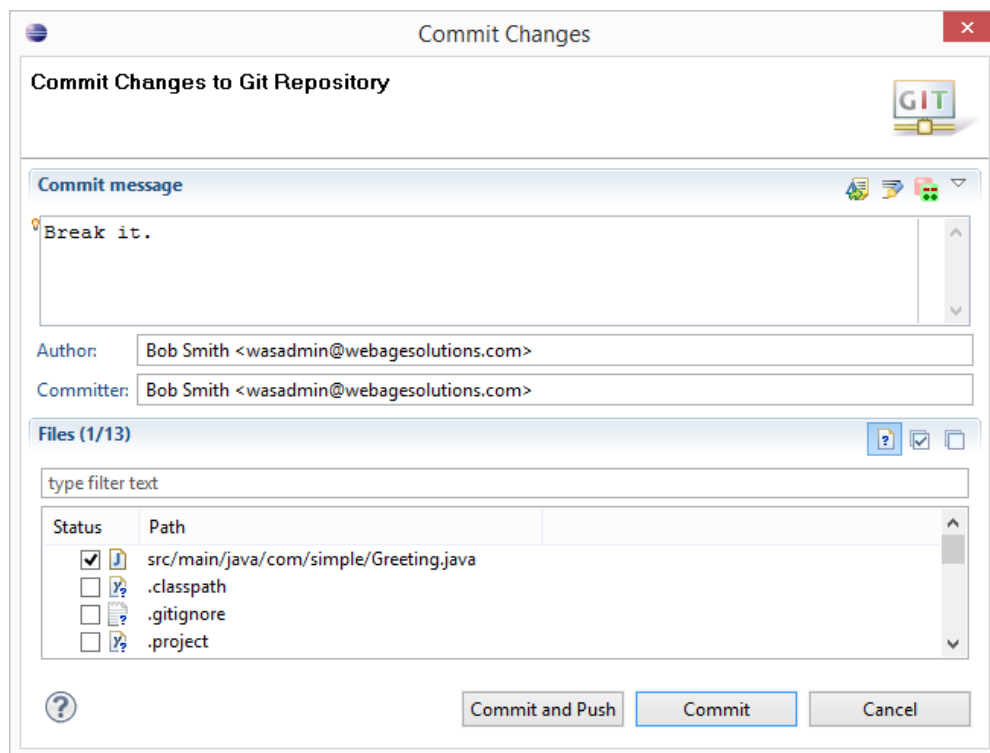
__2. Locate the line that reads 'return "GOOD";'. Change it to read 'return "BAD";'

```
public String getStatus() {  
  
    return "BAD";  
  
}
```

__3. Save the file.

__4. In the **Project Explorer**, right-click on **Greeting.java** and then select **Team** → **Commit...** (This is a shortcut for committing a single file).

__5. Enter an appropriate commit message and then click **Commit and Push**.



__6. Click **OK** in the results box, to close it.

__7. Switch back to Jenkins.

__8. In a minute or so, you should see a build launched automatically. Jenkins has picked up the change in the 'Git' repository and initiated a build. If nothing happens then click **Build Now**.

Stage View



This time, the results are a little different. The 'Maven Build' stage is showing a failure, and the 'Archive Test Results' stage was never executed.

What's happened is that the unit tests have failed, and Maven exited with a non-zero result code because of the failure. As a result, the rest of the pipeline was canceled. This behavior probably isn't what you want or what you expect in most cases. We'd like to go ahead and archive the test results, even when there's a failure. That way, we can see the trend including failed tests.

The solution here is to add a command-line parameter to the Maven invocation. If we add '-Dmaven.test.failure.ignore' to the Maven command line, then Maven will continue with the build even if the tests fail.

__9. Go back to Eclipse and open the 'Jenkinsfile' if necessary.

__10. Alter the 'bat "mvn..." line to read as follows:

```
bat 'mvn -Dmaven.test.failure.ignore install'
```

__11. Save the 'Jenkinsfile'.

__12. Commit and push the changes using the same technique as above.

__13. After a minute or so, you should see a new Pipeline instance launched. If nothing happens then click **Build Now**.

Stage View



This time, the pipeline runs to completion, and the test results are archived as expected. Notice that the build is now flagged as 'unstable' (indicated by the yellow color and the icon). The JUnit archiver noticed the failures and flagged the build unstable, even though Maven exited normally.

Part 4 - Add a Manual Approval Step

One of the interesting features of the Pipeline functionality is that we can include manual steps. This is very useful when we're implementing a continuous deployment pipeline. For example, we can include a manual approval step (or any other data collection) for cases like 'User Acceptance Testing' that might not be fully automated.

In the steps below, we'll add a manual step before a simulated deployment.

- __1. Go to 'Eclipse' and open the 'Jenkinsfile' if necessary.
- __2. Add the following to the end of the file:

```
stage 'User Acceptance Test'

def response= input message: 'Is this build good to go?',
    parameters: [choice(choices: 'Yes\nNo',
        description: '', name: 'Pass')]

if(response=="Yes") {
    node {
        stage 'Deploy'
        bat 'mvn -Dmaven.test.failure.ignore install'
    }
}
```

This portion of the script creates a new stage called 'User Acceptance Test', then executes an 'input' operation to gather input from the user. If the result is 'Yes', the script executes a deploy operation in a new 'node' step. (In this case, we're repeating the 'mvn install' that we did previously. Only because we don't actually have a deployment repository setup)

__3. Save and commit 'Jenkinsfile' as previously.

__4. When the pipeline executes, watch for a "paused" stage called 'User Acceptance Test'. If you move your mouse over this step, you'll be able to select "Yes" or "No", and then click **Proceed**.

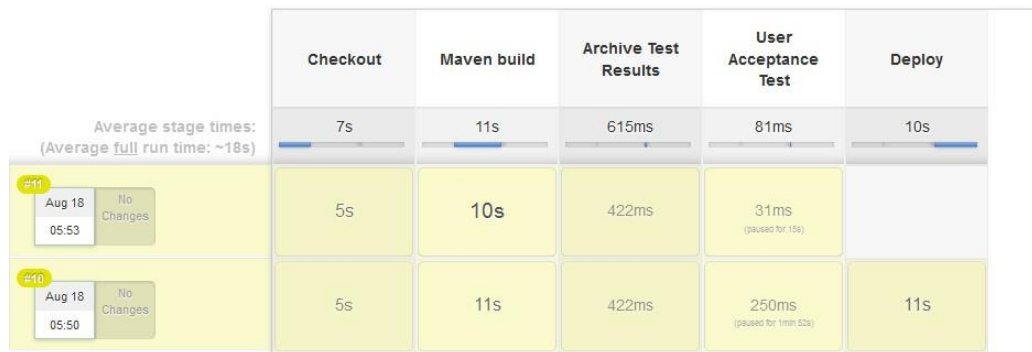
Stage View



__5. Select **Yes** and click **Proceed**, you should see the job run to completion.

__6. If you run the pipeline again (click **Build Now**), but this time, select **No** on the 'User Acceptance Test', you'll see that the pipeline exits early and doesn't run the 'deploy' stage.

Stage View



What's happened is that the final 'Deploy' stage was only executed when we indicated that the 'User Acceptance Test' had passed.

__7. Close all.

Part 5 - Review

In this lab, we explored the Pipeline functionality in Jenkins. We built a simple pipeline in the Jenkins web UI, and then used a 'Jenkinsfile' in the project. Lastly, we explored how to gather user input, and then take different build actions based on that user input.

Lab 7 - Advanced Pipeline with Groovy DSL

In this lab you will utilize Groovy DSL and Shared Library to create an advanced pipeline in Jenkins.

Part 1 - Explore Groovy DSL basics

In this part you will explore Groovy DSL basics. You will use Jenkins built-in Script Console for testing the Groovy script.

__1. In the web browser enter following URL to access Jenkins.

`http://localhost:8080/`

__2. Login as wasadmin for user and password.

__3. Click **Manage Jenkins**.



__4. Click **Manage Nodes**.



[Manage Nodes](#)

Add, remove, control and monitor the various nodes that Jenkins runs jobs on.

__5. If you see a **Linux Build** like below click on it.

S	Name ↓	Architecture	Clock Difference	Free Disk
	Linux Build	Linux (amd64)	In sync	6
	master	Windows 8.1 (x86)	In sync	24
Data obtained		33 sec	33 sec	

__6. Then click **Delete Agent** and click **Yes** to confirm deletion.



[Delete Agent](#)

__7. Click **Manage Jenkins**.

__8. Click **Script Console**.



Script Console

Executes arbitrary script for administration/trouble-shooting/diagnostics.

Notice it displays a text box where you can write script and run it to see the result. Alternatively, you can access the Script Console by accessing the URL <http://localhost:8080/script>

__9. Enter following text.

```
int square(int a) {  
    return a * a;  
}  
  
println "Square of 5 is: ${square(5)}"
```

__10. Click **Run** button.

Notice it displays "Square of 5 is: 25".

__11. In the **Script Console** text box, remove previous code and enter the following code.

```
def printFile(location) {  
    pub = new File(location)  
    if (pub.exists()){  
        println "Location ${location}"  
        pub.eachLine{line-> println line}  
    }  
    else{  
        println "${location} does not exist"  
    }  
}  
  
printFile("C:/Windows/System32/drivers/etc/hosts")
```

__12. Click **Run**.

Notice it displays file contents.

Part 2 - Create a directory structure for storing Shared Library content and initialize the Git repository.

In this part you will you create a simple library of reusable functions and add it to Jenkins as a shared library.

__1. Open **Command Prompt**.

__2. Switch to the **Workspace** directory.

```
cd c:\workspace
```

__3. Create a directory for storing Groovy libraries and switch to it.

```
mkdir groovy && cd groovy
```

__4. Create a directory for storing your first library's scripts and switch to the directory.

```
mkdir MyFirstLibrary.git && cd MyFirstLibrary.git
```

__5. Initialize Git repository.

```
git init .
```

__6. Configure user name and email address required for committing changes to the repository.

```
git config user.name "Bob"
```

```
git config email.address "bob@abcinc.com"
```

__7. Create directory structure for storing the Groovy source code.

```
mkdir src\com\abcinc
```

Note: **src** is where source code must reside. In the **src** directory you can organize source code in the form of packages. **com.abcinc** package corresponds to **com\abcinc** directory structure.

Part 3 - Create a simple Groovy script, commit it to the repository, and add it to Jenkins as a shared library.

In this part you will create a simple Groovy script, commit it to the repository, add it to Jenkins as a shared library.

__1. Create a simpleClass.groovy file using the Notepad. Click Yes to create the file.

```
notepad src\com\abcinc\simpleClass.groovy
```

__2. Enter following text.

```
package com.abcinc

import java.text.SimpleDateFormat

class simpleClass {
    int square(int a) {
        return a * a;
    }

    def sayHello(def name) {
        return "Hi, ${name}";
    }

    def getDateTime() {
        def date = new Date()
        def sdf = new SimpleDateFormat("MM/dd/yyyy HH:mm:ss")
        return sdf.format(date);
    }
}
```

__3. Save the file.

__4. Add the content to the Git repository.

```
git add .
```

__5. Commit changes.

```
git commit -m "Added simpleClass"
```

__6. In the web browser open Jenkins.

`http://localhost:8080`

__7. Click **Manage Jenkins**.

__8. Click **Configure System**.

__9. Scroll down the page, locate **Global Pipeline Libraries**, and click **Add** button.

Global Pipeline Libraries

Sharable libraries available to any Pipeline jobs running on this system. These libraries will be trusted, meaning they run without "sandbox" restrictions and may use @Grab.

Add

__10. In **Name** text box, enter "my-shared-lib" (without quotes).

__11. In **Default Version** text box, enter "master" (without quotes).

__12. Under **Retrieval Method**, click "Modern SCM".

__13. Under **Source Code Management** click **Git**.

__14. In **Project Repository** text box, enter "c:\workspace\groovy\MyFirstLibrary.git" (without quotes).

Library

Name

my-shared-lib

Default version

master

Cannot validate default version until after saving and reconfiguring.

Load implicitly

☐

Allow default version to be overridden ☒

Retrieval method

☒ Modern SCM

Source Code Management

☒ Git

Project Repository

c:\workspace\groovy\MyFirstLibrary.git

__15. Click **Apply** button.

__16. Click **Save** button.

Note: Next you will verify you are connected to the repository properly.

__17. Click **Manage Jenkins**.

__18. Click **Configure System**.

__19. Scroll down the page and notice it shows the repository revision number.

Global Pipeline Libraries

Sharable libraries available to any Pipeline jobs running on this system. These libraries will be trusted, meaning they run without "sandbox" restrictions and may use @Grab.

Library	
Name	my-shared-lib
Default version	master
	Currently maps to revision: 2ea0ae0b71303c9300a92c795297473bf2113e49
Load implicitly	<input type="checkbox"/>
Allow default version to be overridden	<input checked="" type="checkbox"/>

Note: The number will most likely be different on your side.

Part 4 - Create a Jenkins Job and utilize the shared library you created previously

In this part you will create a Jenkins Job and utilize the shared library you created previously.

__1. In the web browser ensure you have Jenkins web site open.

http://localhost:8080

__2. Click **New Item**.

__3. In **Enter an item name** enter "Shared Library Pipeline" (without quotes).

__4. Select **Pipeline**.

__5. Click **OK** button.

__6. Scroll through the page, locate **Pipeline** section, and in **Script** enter following text.

```
@Library('my-shared-lib')
import com.abcinc.simpleClass;

def m = new simpleClass();

println m.sayHello("Bob");
println "Square is: ${m.square(5)}";
println "Date is: ${m.getDateTime()}";
```

__7. Click **Apply** button.

Note: Ignore red icon(s), if there are any.

__8. Click **Save** button.

__9. Click **Build Now**.

Notice a build shows up under **Build History** with blue colored icon.

__10. Click the build under **Build History**.

__11. Click **Console Output**.

Notice the pipeline execution checkouts the Groovy script(s) from the repository then displays the results. The result should be similar to this:

```
Started by user Administrator
Loading library my-lib@master
> git.exe rev-parse --is-inside-work-tree # timeout=10
Setting origin to C:\Workspace\groovy\MyGroovyLibrary
> git.exe config remote.origin.url C:\Workspace\groovy\MyGroovyLibrary # timeout=10
Fetching origin...
Fetching upstream changes from origin
> git.exe --version # timeout=10
> git.exe fetch --tags --progress origin +refs/heads/*:refs/remotes/origin/*
> git.exe rev-parse "master^{commit}" # timeout=10
> git.exe rev-parse "origin/master^{commit}" # timeout=10
> git.exe rev-parse --is-inside-work-tree # timeout=10
Fetching changes from the remote Git repository
> git.exe config remote.origin.url C:\Workspace\groovy\MyGroovyLibrary # timeout=10
Fetching upstream changes from C:\Workspace\groovy\MyGroovyLibrary
> git.exe --version # timeout=10
> git.exe fetch --tags --progress C:\Workspace\groovy\MyGroovyLibrary +refs/heads/*:refs/remotes/origin/*
Checking out Revision 56c3b67d784ee6d41060d65e093435e8d2009e26 (master)
> git.exe config core.sparsecheckout # timeout=10
> git.exe checkout -f 56c3b67d784ee6d41060d65e093435e8d2009e26
> git.exe rev-list 56c3b67d784ee6d41060d65e093435e8d2009e26 # timeout=10
[Pipeline] echo
Hi, Bob
[Pipeline] echo
Square is: 25
[Pipeline] echo
Date is: 02/17/2017 07:32:49
[Pipeline] End of Pipeline
Finished: SUCCESS
```

Part 5 - Create a standardized software engineering process shared library

In this part you will define a standardized software engineering process as a shared library. It will clean, build, verify, await approval, and deploy the project.

__1. In the **Command Prompt** switch to the directory where you will store scripts for the shared library.

```
cd c:\workspace\groovy\MyFirstLibrary.git
```

__2. Open **Notepad** and create file for the library. Click Yes to create the file.

```
notepad src\com\abcinc\utils.groovy
```

__3. Enter following text.

```
package com.abcinc;

def checkout() {
    node {
        stage 'Checkout'
        git url: 'C:\\Software\\repos\\SimpleGreeting.git'
    }
}

def mvn_install() {
    node {
        stage 'Install'
        bat 'mvn install'
    }
}

def mvn_clean() {
    node {
        stage 'Clean'
        bat 'mvn clean'
    }
}

def mvn_verify() {
    node {
        stage 'Verify'
        bat 'mvn verify'
    }
}

def archive_reports() {
    node {
        stage 'Archive Reports'
        step([$class: 'JUnitResultArchiver', testResults:
'**/target/surefire-reports/TEST-*.xml'])
    }
}

def user_acceptance(wkdir) {
    node {
        stage 'User Acceptance Test'
```

```

def response= input message: 'Is this build good to go?',
                    parameters: [choice(choices: 'Yes\nNo',
                    description: '', name: 'Pass')]

    if(response=="Yes") {
        node {
            stage 'Deploy'

            bat "xcopy \"C:\\Program Files\\Jenkins\\workspace\\
$wsdir\\target\\SimpleGreeting*.jar\" C:\\workspace\\dev\\ /y"
        }
    }
}

```

IMPORTANT NOTE:

Make sure Jenkins is installed under **C:\\Program Files** if not then change the path in the last line above if Jenkins is in another folder.

Checkout function utilizes git to checkout from an existing repository. Ensure c:\\software\\repos\\SimpleGreeting.git directory is present. In case if SimpleGreeting.git is not present, duplicate the SimpleGreeting directory as SimpleGreeting.git, then run **git add .** followed by **git commit -m "added SimpleGreeting.git"**

__4. Save and close the file.

__5. In **Command Prompt** stage the new file.

```
git add .
```

__6. Commit the changes to the repository.

```
git commit -m "added utils.groovy"
```

Note: You have already configured the c:\\workspace\\groovy\\MyFirstLibrary as a shared library in previous parts of this lab. It is configured as shared library with the name "my-shared-lib".

Part 6 - Create a Jenkins Job and utilize the shared library

In this part you will create a Jenkins Job and utilize the shared library created in the previous part of this lab.

__1. In the web browser ensure you have Jenkins web site open.

`http://localhost:8080`

__2. Click **New Item**.

__3. In **Enter an item name** enter "ABCInc" (without quotes).

__4. Select **Pipeline**.

__5. Click **OK** button.

__6. Scroll through the page, locate **Pipeline** section, and in **Script** enter following text.

```
@Library('my-shared-lib')
import com.abcinc.utils;

def u = new utils();

u.checkout();
u.mvn_clean();
u.mvn_install();
u.mvn_verify();
u.user_acceptance("ABCInc");
```

Note: utils is an implicit class. A class having same name as groovy script file is automatically created.

__7. Click **Apply** button.

Note: Ignore red icon(s), if there are any.

__8. Click **Save** button.

Before Build we need to fix the code that we broke in a previous lab.

__9. Open eclipse in the same workspace.

__10. Go to the **Project Explorer** and locate the file 'Greeting.java'. It will be under **src/main/java** in the package 'com.simple'. Open 'Greeting.java'.

__11. Locate the line that reads 'return "BAD";'. Change it to read 'return "GOOD";'

__12. Save the file.

__13. In the **Project Explorer**, right-click on **Greeting.java** and then select **Team** → **Commit...** (This is a shortcut for committing a single file).

- __14. Enter an appropriate commit message and then click **Commit and Push**.
- __15. Click **OK** in the results box, to close it.
- __16. Go back to Jenkins.
- __17. Click **Build Now**.
- __18. When the stage progresses to "User Acceptance", hover the mouse over "User Acceptance Test", select "Yes", and click "Proceed" button.

The build will complete all stages.

Checkout	Clean	Install	Verify	User Acceptance Test	Deploy
582ms	1s	3s	2s	150ms	445ms
718ms	1s	3s	2s	120ms (paused for 9s)	642ms

- __19. After all stages are completed, notice a new build shows up under **Build History**.
- __20. Click the latest build.
- __21. Click **Console Output**.
- __22. Notice it shows all stages and their details.
- __23. In **File Explorer** go to **c:\workspace\dev** and notice SimpleGreeting-*.jar is copied here by the shared library script upon user acceptance.
- __24. Close all.

Part 7 - Review

In this lab you utilized Groovy DSL and Shared Library to create an advanced pipeline in Jenkins.

Lab 8 - Configure Jenkins Security

In this lab we will configure Jenkins to authenticate users against its internal user database, and enforce capability limitations on users.

At the end of this lab you will be able to:

1. Enable security and select the internal user database
2. Create users
3. Assign global privileges to users
4. Assign project-specific privileges to users.

Part 1 - Enable Jenkins Security

___1. Make sure Jenkins is started. Since we configured as windows service it will be started every time you start the machine.

___2. Go to the Jenkins console:

`http://localhost:8080`

___3. Enter **wasadmin** as user and password and click **Log in**.

___4. On the Menu click **Manage Jenkins**.



___5. Click **Configure Global Security**.


__6. Jenkins will display the **Configure Global Security** page.





☒ Enable security

__7. Under the **Security Realm** heading, select **Jenkins own user database**. Select **Allow users to sign up**.

Security Realm

☐ Delegate to servlet container 

☒ Jenkins' own user database 

☒ Allow users to sign up 

☐ LDAP

__8. Click the **Save** button at the bottom of the page.

At this point, we have enabled security, but we haven't created any users, nor have we actually applied an authorization requirement. Let's go ahead and add a couple of users.

Part 2 - Create an Administrative User

There are two ways to add users: We can do it through Jenkins' management console, or we can allow users to sign up themselves. Let's first use the management console to add an administrative user so we can lock down the security.

__1. In the "Manage Jenkins" page (the last part of the lab should have left you here), click on **Manage Users**.



[Manage Users](#)

Create/delete/modify users that can log in to this Jenkins

__2. Click on **Create User**.



__3. The system displays the **Sign Up** page. Enter the following information in the appropriate fields:

Username:	admin
Password:	password
Confirm Password:	password
Full name:	Administrative User
E-mail address:	admin@localhost.com

Note that the e-mail address is not actually verified to be a valid email address, but Jenkins will reject it unless it is in the usual email format.

__4. When the page looks like below, click on the **Create User** button.

Create User



Username:	<input type="text" value="admin"/>
Password:	<input type="password" value="password"/>
Confirm password:	<input type="password" value="password"/>
Full name:	<input type="text" value="Administrative User"/>
E-mail address:	<input type="text" value="admin@localhost.com"/>

Create User

__5. The system will display the list of current users, including the 'admin' user that you just created.

Users

These users can log into Jenkins. This is a sub set of [this list](#), which some commits on some projects and have no direct Jenkins access.

User Id	
 admin	Administrative User
 wasadmin	Administrator

__6. Click on **Manage Jenkins** to return to the management console.

Part 3 - Enable Authentication

__1. Click on the link for **Configure Global Security**.



[Configure Global Security](#)
Secure Jenkins; define who is

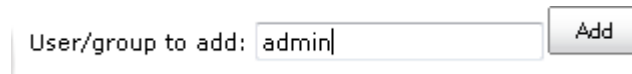
__2. Under the **Authorization** heading, select **Project-based Matrix Authorization Strategy**.

Authorization

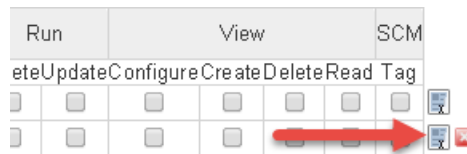
- ☐ Anyone can do anything
- ☐ Legacy mode
- ☐ Logged-in users can do anything
- ☐ Matrix-based security
- ☒ Project-based Matrix Authorization Strategy

When you click on the radio button above, Jenkins will display a list of global authorizations. We need to enter the 'admin' user here with full permissions, and then we'll add other users to individual projects.

___3. In the field labeled **User/group to add:**, enter **admin**, and then click **Add**.

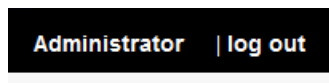


___4. Jenkins displays the newly-added user in the list of users. Now we need to select all the permissions. You could click each permission box listed for 'admin' individually, but to save a little time, if you scroll the window horizontally all the way over to the right-hand side, you'll find a button that will select all the permissions in one operation. Find that button and click it.

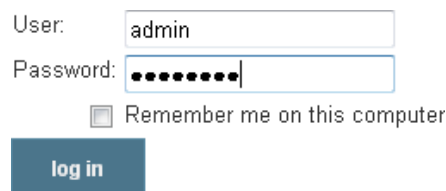


___5. All check boxes will be selected. Click **Save**.

___6. Since we have altered the authorization strategy, Jenkins resets its security system, which requires us to log in again. You will see that wasadmin access has been denied. Click the **log out** link at the upper-right corner of Jenkins' home page.



___7. At the login screen, enter **admin** as the userid and **password** as the password, then click **log in**.



If for whatever reason, Jenkins doesn't let you log in, check with your instructor – you may need to reset the security system by editing the configuration file manually and then start over with the security setup.

___8. Do not save the password.

___9. Click the **log out** link at the upper-right corner of Jenkins' home page.

Jenkins should display the login page again.

Part 4 - Create a Self-Signed-Up User

When we enabled Jenkins security, we left the checkbox selected for “Allow users to sign up”. As a result, there is a **Create an Account** link on the login page. Let's create a user that way, and then we'll go back and grant them privileges on a project.

__1. Click on **Create an account** in the login page.



[Create an account](#) if you are not a member yet.

__2. Jenkins will display the **Sign up** page. Enter the following information in the appropriate fields:

Username:	jane
Password:	password
Confirm Password:	password
Full name:	Non-Administrative User
E-mail address:	jane@localhost.com

Note that the e-mail address is not actually verified to be a valid email address, but Jenkins will reject it unless it is in the usual email format.

__3. When the page looks like below, click on the **Sign up** button.

Sign up

Username:	<input type="text" value="jane"/>
Password:	<input type="password" value="password"/>
Confirm password:	<input type="password" value="password"/>
Full name:	<input type="text" value="Non-Administrative User"/>
E-mail address:	<input type="text" value="jane@localhost.com"/>

Sign up

__4. Jenkins will display the **Success** window. Do not save the password.

Success

You are now logged in. Go back to [the top page](#).

__5. Click on the link to go to **the top page**.

__6. Since the new user has no permissions, access is denied.

Access Denied

 jane is missing the Overall/Read permission

__7. Click on the **log out** link, and then log back in using 'admin/password'.

__8. We should see the main dashboard page. Click on the **SimpleGreeting** job (we created this job in an earlier lab).

__9. Click **Configure**.

__10. In the configuration page, click on the checkbox marked **Enable project-based security**, to select it.


☒ Enable project-based security

__11. When you select the checkbox, Jenkins will display a matrix of users and permissions. In the field marked **User/group to add:**, enter **jane**, and then click **Add**.

User/group to add:

Add

__12. In the row labeled **jane**, select the checkboxes for **Discover**, **Read** and **Workspace**.

User/group	Credentials					Job								
	Create	Delete	Manage	Domains	Update	View	Build	Cancel	Configure	Delete	Discover	Move	Read	Workspace
Anonymous	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
 jane	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>

__13. Click **Save**.



__14. We also need to grant jane the 'overall read' permission. Click **Back to Dashboard**.

__15. Click the **Manage Jenkins** link, and then select **Configure Global Security**.

__16. Under the **Authorization** heading, in the field marked **User/group to add:** enter **jane** and then click **Add**.

User/group to add:

__17. In the row labeled **jane**, select the checkbox for **Read** under **Overall**.

User/group	Overall				
	Administer	Configure	UpdateCenter	Read	Run
 admin	<input checked="" type="checkbox"/>		<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
Anonymous	<input type="checkbox"/>		<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
 jane	<input type="checkbox"/>		<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>

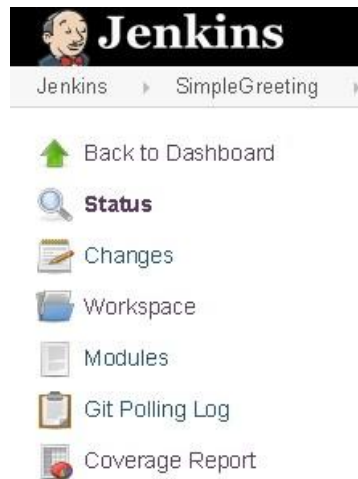
__18. Click **Save**.

__19. Log out, and then log back in as 'jane/password'.

__20. Note that Jenkins only displays the 'SimpleGreeting' job. This is because Jane only has read and discover access to this project.

__21. Click on the **SimpleGreeting** job.

__22. Notice that Jane doesn't have full privileges – there is no **Build Now** or **Configure** option on the project.



__23. Log out of Jenkins.

Next we will give full permission to the user wasadmin.

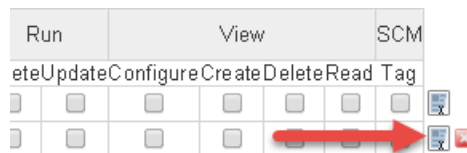
__24. Login back as admin/password.

__25. Click **Manage Jenkins**.

__26. Click on the link for **Configure Global Security**.

__27. In the field labeled **User/group to add:**, enter **wasadmin**, and then click **Add**.

__28. Jenkins displays the newly-added user in the list of users. Now we need to select all the permissions. You could click each permission box listed for 'admin' individually, but to save a little time, if you scroll the window horizontally all the way over to the right-hand side, you'll find a button that will select all the permissions in one operation . Find that button and click it.



__29. All check boxes will be selected. Click **Save**.

__30. Logout.

__31. Login back as wasadmin and make sure you have full access.

__32. Close all.

Part 5 - Review

In this lab, we went through a series of steps to enable and configure Jenkins security. We saw how to create users administratively, and how to configure users who sign up using the self-sign-up screen.

