

Lab - Create a Jenkins Job

This lab creates a freestyle Jenkins build job that runs every time a commit is made to your GitHub repository. Freestyle projects allow you to configure just about any sort of build job, they are highly flexible and very configurable.

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

1. Create a Jenkins Job that continuously integrates with a Git repository.

Part 1 - 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.

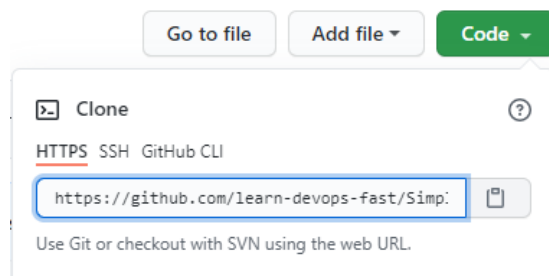
In this lab, you will log into your personal GitHub account and fork an existing GitHub repository. You will then create a Jenkins job that pulls source files from your fork. We will configure the Jenkins job to monitor your GitHub repository and trigger a build any time the GitHub repository changes.

We will then make changes to the files in GitHub, both in GitHub's online editor and in Eclipse, and watch Jenkins automatically run a build when commits are pushed to the remote GitHub repo.

__1. Log into GitHub with your personal account and fork the SimpleGreeting repository which can be found here:

`https://github.com/cameronmcnz/SimpleGreeting`

__2. Find the GitHub URL of your fork and copy it.

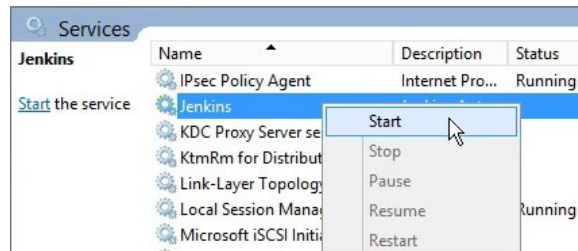


The URL should have the same structure as the original repository with your username in place of the text String cameronmcnz.

`https://github.com/<your username>/SimpleGreeting.git`

Part 2 - Create the Jenkins Job

__1. Make sure Jenkins is running by opening the Windows Services panel and checking to see if the Jenkins service is started. Start the Jenkins service if it is stopped.



If Jenkins will not start, you may have a conflict on port 8080. Ensure there are no other application servers, embedded Tomcat servers, containers or applications running inside your IDE that are blocking port 8080.

__2. Open the Jenkins administrative console in a web browser by navigating to:

<http://localhost:8080>

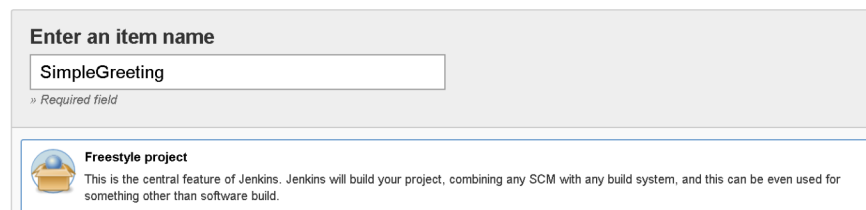
If you are prompted to login, the credentials are wasadmin/wasadmin.

__3. From the Jenkins home page, click on the **New Item** link.



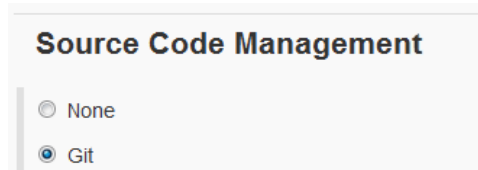
__4. Enter **SimpleGreeting** for the project name.

__5. Select **Freestyle project** as the project type.

A screenshot of the Jenkins 'Enter an item name' form. The form has a text input field with the value 'SimpleGreeting'. Below the input field, there is a small text '» Required field'. Below the form, there is a section for selecting the project type. The 'Freestyle project' option is selected, and it is highlighted with a blue border. The description for 'Freestyle project' is: '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.'

__6. Click **OK**, to create the new Jenkins job. After the job is created, you will be on the job configuration page.

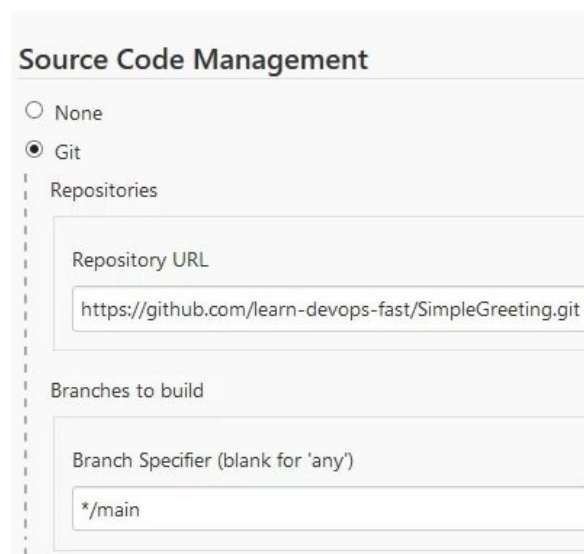
__7. Scroll down to the **Source Code Management** section and then select **Git**.



The screenshot shows a section titled "Source Code Management". Below the title, there are two radio button options: "None" and "Git". The "Git" option is selected, indicated by a filled blue circle next to it.

__8. Under Repositories, enter the path to your fork of the SimpleGreeting GitHub repository in the Repository URL field. The GitHub URL should take the form of:

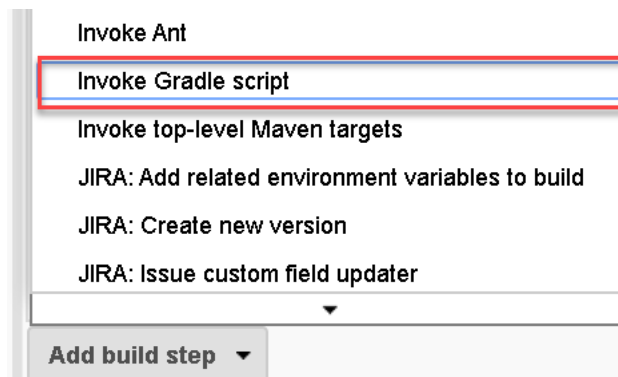
`https://github.com/<your username>/SimpleGreeting.git`



The screenshot shows the "Source Code Management" section with "Git" selected. Below this, there is a "Repositories" section. Inside it, the "Repository URL" field is filled with the text "https://github.com/learn-devops-fast/SimpleGreeting.git". Below the "Repositories" section is a "Branches to build" section. Inside it, the "Branch Specifier (blank for 'any')" field is filled with the text "*/main".

__9. Change the Branch Specifier to `*/main` if it has defaulted to `*/master`

__10. In **Build** section, click **Add build step > Invoke Gradle script**



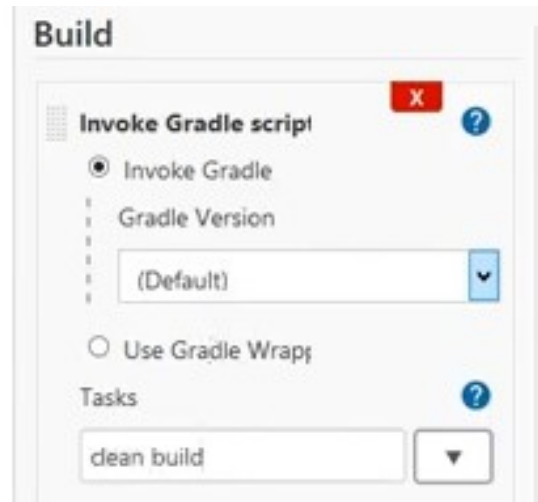
The screenshot shows a dropdown menu for adding a build step. The menu is open, showing several options. The option "Invoke Gradle script" is highlighted with a red rectangular border. Below the menu, there is a button labeled "Add build step" with a downward arrow.

__ 11. Ensure **Invoke Gradle** radio button is selected.

__ 12. In **Gradle Version**, select **Gradle if the tool has been configured**. If the Gradle option is not available, select (Default).

__ 13. In **Tasks**, enter **clean build**

The **Invoke Gradle script** configuration should look like this:

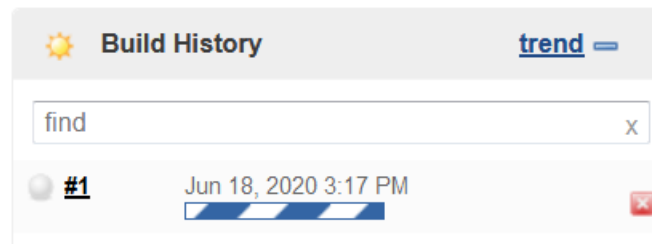


The screenshot shows a 'Build' configuration panel. Under the 'Invoke Gradle script' section, the 'Invoke Gradle' radio button is selected. Below it, the 'Gradle Version' dropdown menu is set to '(Default)'. The 'Use Gradle Wrapper' radio button is unselected. In the 'Tasks' section, the text 'clean build' is entered into the input field.

__ 14. Click **Save**. You will be sent to the landing page of the SimpleGreeting Job.

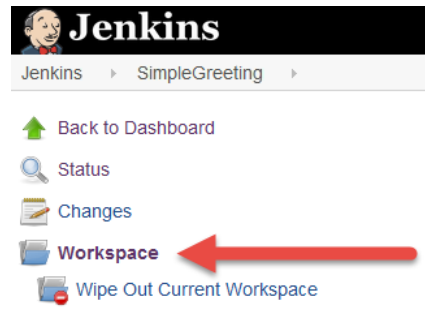
__ 15. On the right hand side, click **Build Now**.

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



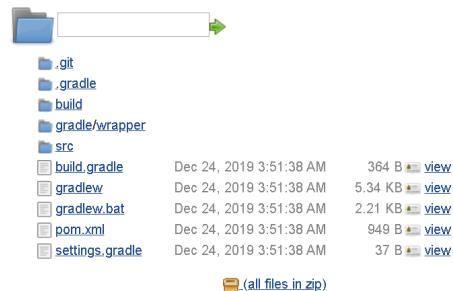
The screenshot shows the 'Build History' section with a search bar containing the text 'find'. Below the search bar, there is a single build entry labeled '#1' with a status icon. To the right of the entry, the timestamp 'Jun 18, 2020 3:17 PM' is displayed. A progress bar is shown below the timestamp, and a red 'x' icon is visible on the right side of the entry.

__16. After a few seconds the build will complete, the progress bar will stop. Click on **Workspace**.



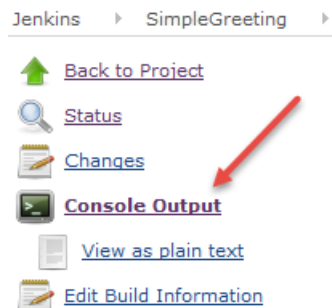
You will see that the directory is populated with the source code for our project.

Workspace of SimpleGreeting on master



__17. Find the **Build History** box, and click on the 'time' value for the most recent build. You should see that the build was successful.

__18. Click the **Console Output** from the left menu.



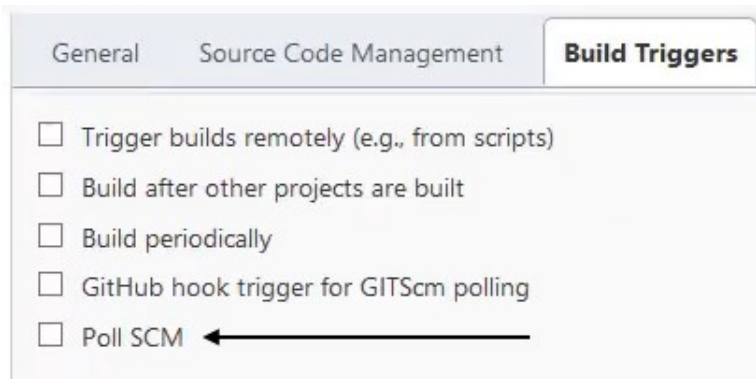
__19. At the end of the console output you there will be messages indicating the build was a success. You have created a project and built it successfully.

```
BUILD SUCCESSFUL in 11s
5 actionable tasks: 5 executed
Build step 'Invoke Gradle script' changed build result to SUCCESS
Finished: SUCCESS
```

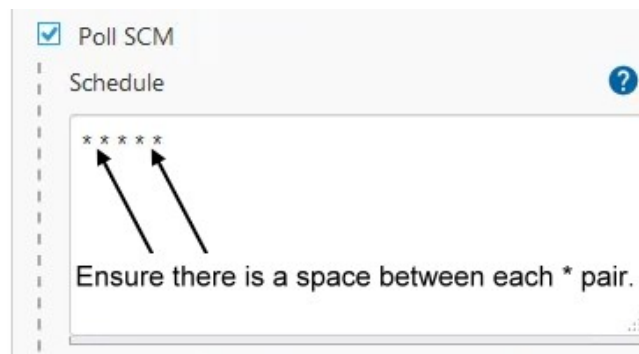
Part 3 - Enable Polling on the Repository

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 check box next to **Poll SCM**
- __5. With spaces between each asterisk, type '* * * * *' 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.

___6. In **Post-build Actions** section, click **Add post-build action** and select **Publish JUnit test result report**.

This will allow you to graphically view unit test results.

___7. In **Test reports XMLs**, enter the following:

```
build\test-results\test\TEST-*.xml
```

Note: By default, Gradle stores test results in the folder mentioned above. In this case, the actual file name is TEST-com.simple.TestGreeting.xml but you can use * wildcard to specify the file name.

___8. Click **Save**.

Part 4 - Trigger a Jenkins Build from GitHub

The Jenkins build job is currently configured to query GitHub every minute for a code change that results in a new commit. To trigger a Jenkins build, we can edit a Java source file directly in GitHub

___1. Go to your fork of the SimpleGreetings repository in GitHub and open the Greeting.java file which can be found in the following location of the Gradle project:

```
SimpleGreeting/src/main/java/com/simple/Greeting.java
```

___2. Ensure you are on the `main` branch and not the `master` branch, as Jenkins has been configured to monitor the `main` branch.

___3. Click the pencil icon to edit the Greeting.java file with GitHub's online editor.

```
main SimpleGreeting / src / main / java / com / simple / Greeting.java Go to file ...  
12 lines (10 sloc) 243 Bytes Raw Blame Edit this file  
1 package com.simple;  
2  
3 public class Greeting {  
4     public static void main( String[] args ){  
5         Greeting msg = new Greeting();  
6         System.out.println(msg.getStatus());  
7     }  
8  
9     public String getStatus(){  
10         return "GOOD";  
11     }  
12 }
```

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

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

__5. To save the change, click the green 'Commit changes' button at the bottom of the online editor.

Commit changes

Update Greeting.java

Add an optional extended description...

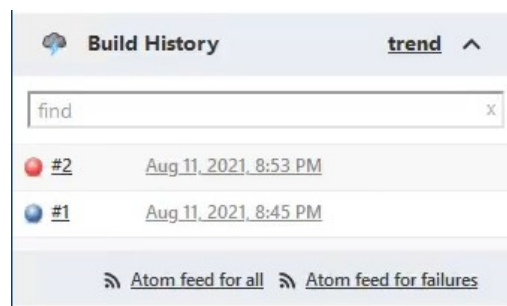
☒ Commit directly to the `main` branch.

☐ Create a new branch for this commit

Commit changes Cancel

__6. Return to the Jenkins administrative console in your web browser. If you happen to have closed it, open a new browser window and navigate to **<http://localhost:8080/SimpleGreeting>**. After 60 seconds, you should see a new build start as your local Jenkins server has detected a new.

__7. The unit test is expecting to see the text string "GOOD". The commit you just made causes the text string "BAD" to be returned, which will cause the build to fail. You may need to refresh the webpage if you do not see the build start after 60 seconds.



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 Gradle automatically runs the unit tests as part of a build, the unit test was run. It failed, and the failure results were logged.

Part 5 - Fix the Unit Test Failure

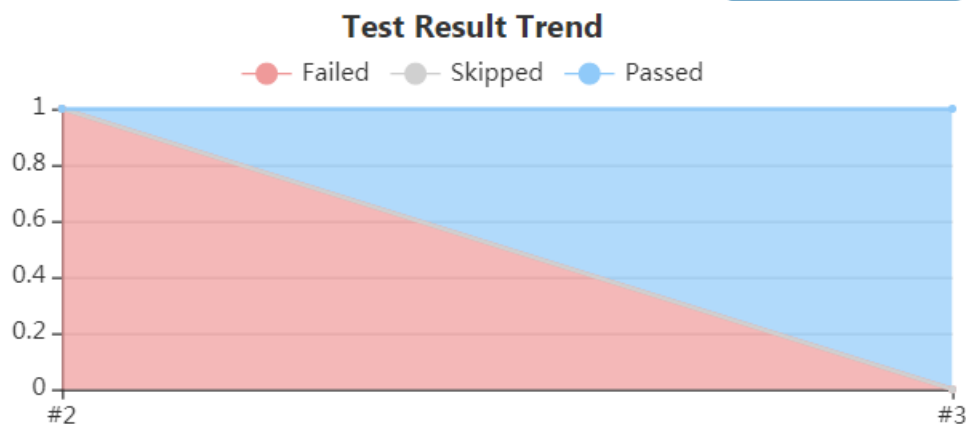
__1. Return to GitHub's online editor and again edit the **Greeting.java** so that the class once again returns 'GOOD'.

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

__2. Commit the the change when the edit is complete.

__3. Watch the Jenkins web browser window. After a minute or two you should see the build start automatically. You could optionally click **Build now** in Jenkins.

Click on the trends result for the SimpleGreeting project and you will see a graph of your build history that will look similar to the graph below.



Part 6 - Developer Workflow

In a typical software project, a developer will clone a GitHub repository, edit code locally, and push back to the project's GitHub repository when a feature is complete. In this part we will replicate the software development workflow.

In order to make changes to the source code, we'll clone a copy of the Git repository into an Eclipse project. You will need the GitHub URL of your fork of SimpleGreeting to perform this lab.

__1. Return to GitHub and copy the GitHub URL of your fork of the SimpleGreeting repository. The URL should follow this format:

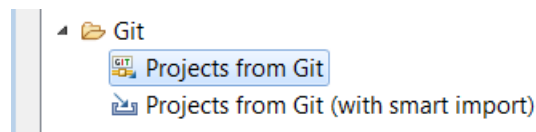
`https://github.com/<your username>/SimpleGreeting.git`

__2. Start Eclipse. There should be a link to Eclipse on your desktop. Alternatively, you can open Eclipse by running `C:\Software\eclipse\eclipse.exe`.

__3. Use `C:\Workspace` as Workspace.

__4. From the main menu, select **File** → **Import...**

__5. Select **Git** → **Projects from Git**.

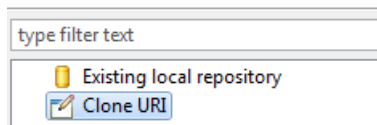


__6. Click **Next**.

__7. Select **Clone URI** and then click **Next**.

Select Repository Source

Select a location of Git Repositories



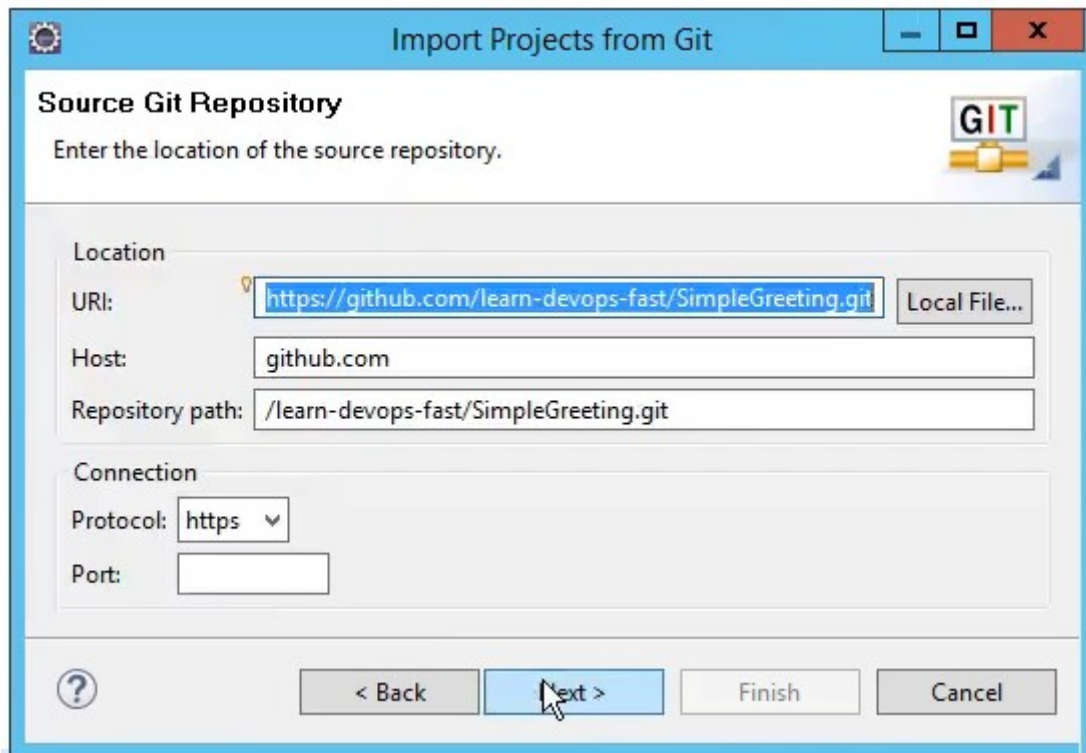
You might think that 'Existing local repository' would be the right choice, since we're cloning from a folder on the same machine. Eclipse, however, expects a "local repository" to be a working directory, not a bare repository. On the other hand, Jenkins will complain if we try to get source code from 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.

__8. If your GitHub URL is on the clipboard, the URI, Host and Repository Path properties will be pre-populated. Otherwise, set these properties.

URI: `https://github.com/<GitHub username>/SimpleGreeting.git`

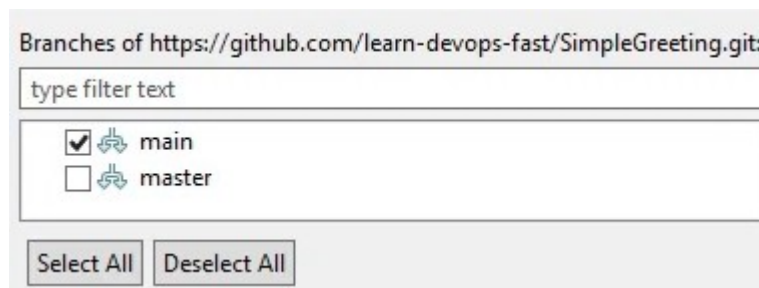
Host: `github.com`

Repository path: `/<GitHub username>/SimpleGreeting.git`

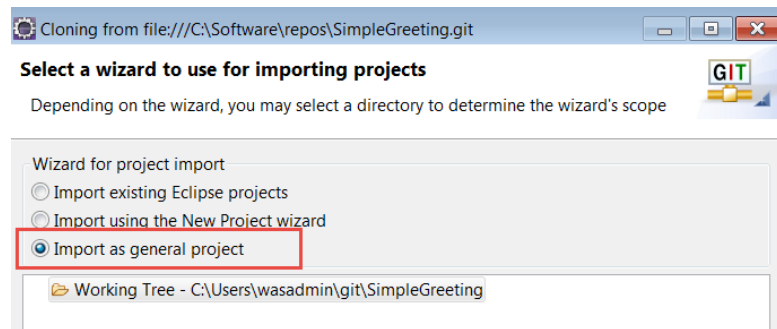


__9. When the Git Repository fields are set, click **Next**.

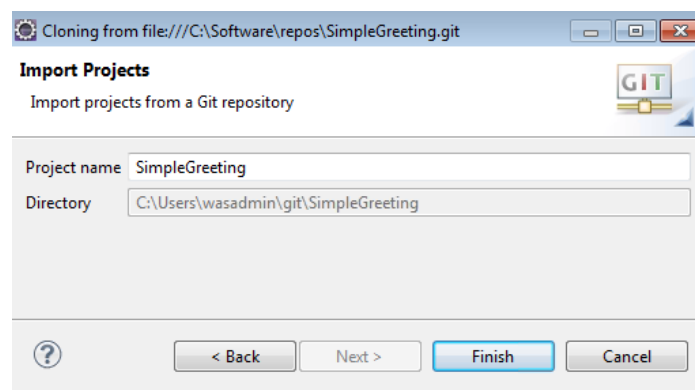
__10. Choose to only import the main branch and then click **Next**.



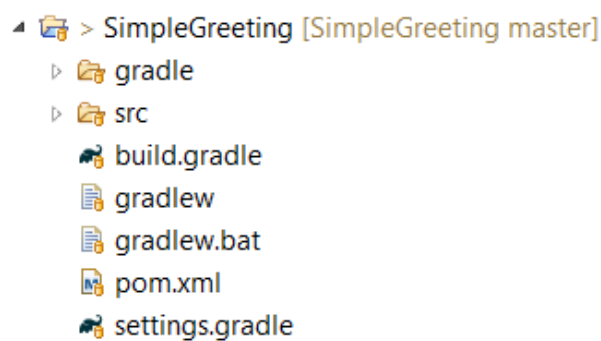
__11. Select **Import as a general project** and click **Next**.



__12. Click **Next** and the wizard lets you edit the name of the imported project. Leave the name as SimpleGreeting and click **Finish**.



__13. You should see the new project in the **Project Explorer**, expand it.



In real-world, after this step, we should convert our project in Eclipse so it understands Gradle project layout. However, it's not required to understand Jenkins continuous integration so we will leave the project layout as it is.

Part 7 - 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.

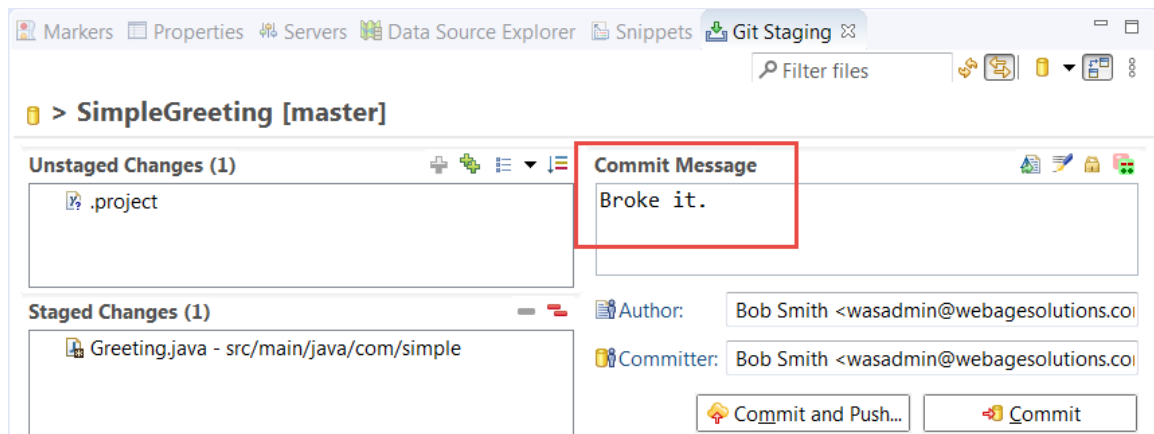
- ___ 1. In the **Project Explorer**, expand the **src/main/java/com/simple** tree node to reveal the **Greeting.java** file.
- ___ 2. Double-click on **Greeting.java** to open the file.
- ___ 3. Find the line that says 'return "GOOD";'. Edit the line to read 'return "BAD";'

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

- ___ 4. 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.

- ___ 5. Right-click on **SimpleGreeting** in the **Project Explorer** and then select **Team** → **Commit**.
- ___ 6. Eclipse will open the **Git Staging** tab. Enter a few words as a commit message, and then click **Commit and Push**.



- ___ 7. Click **Close** in the status dialog that pops up.
- ___ 8. 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 or you can click **Build now** to launch a new build if it's taking too long.

__9. This build should fail. (red circle)

Part 8 - Fix the Unit Test Failure

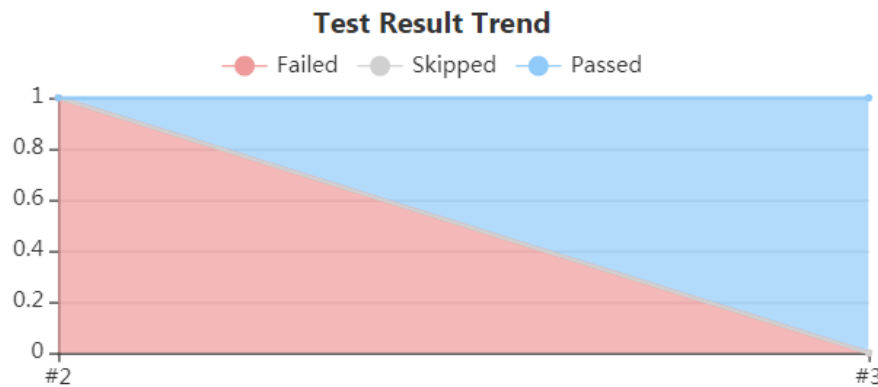
__1. Back in eclipse, edit the file **Greeting.java** so that the class once again returns 'GOOD'.

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

__2. As above, save the file, commit and then 'Commit and Push' the change.

__3. Watch the Jenkins web browser window. After a minute or two you should see the build start automatically or you can click **Build now**, this time the build will pass, when the build is done then refresh the page.

View your build results trend to see how Jenkins has charted your build history.



Part 9 - 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.