### **Getting Started with Continuous Delivery**

**Revision 3.1 – 8/18/21**

**Tech Skills Transformations / Brent Laster**

**IMPORTANT NOTES:**

**1. You should already have your VM image up and running per the setup doc.**

**(If you have not setup the VM yet, checkout the setup doc at** [**https://github.com/skilldocs/cd-intro/blob/main/cd-setup.pdf**](https://github.com/skilldocs/cd-intro/blob/main/cd-setup.pdf) **)**

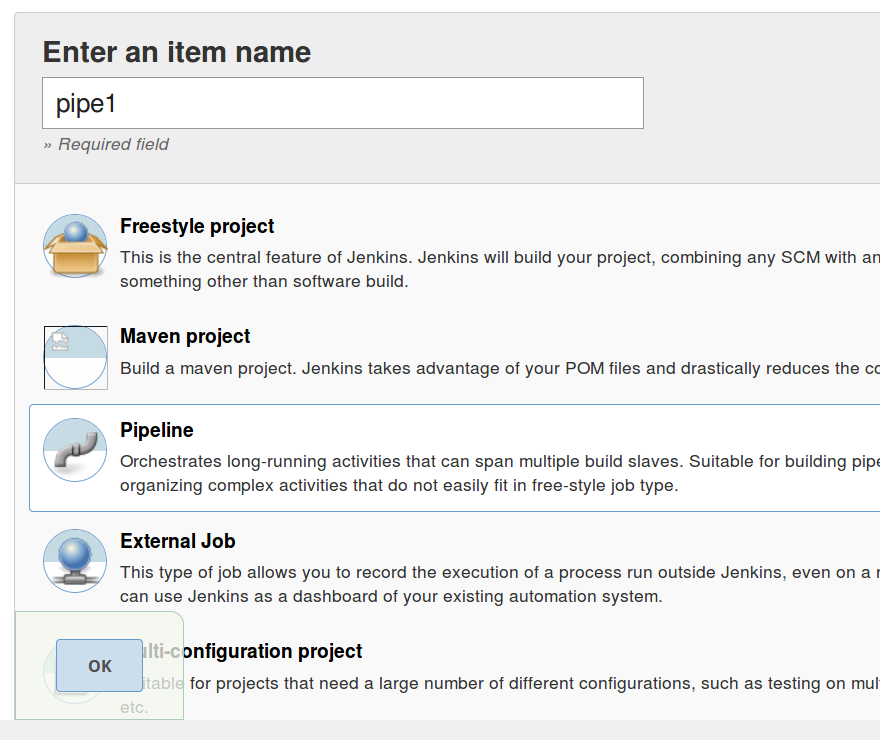
**2. If you run into problems, double-check your typing!**

**Lab 1 starts on the next page!**

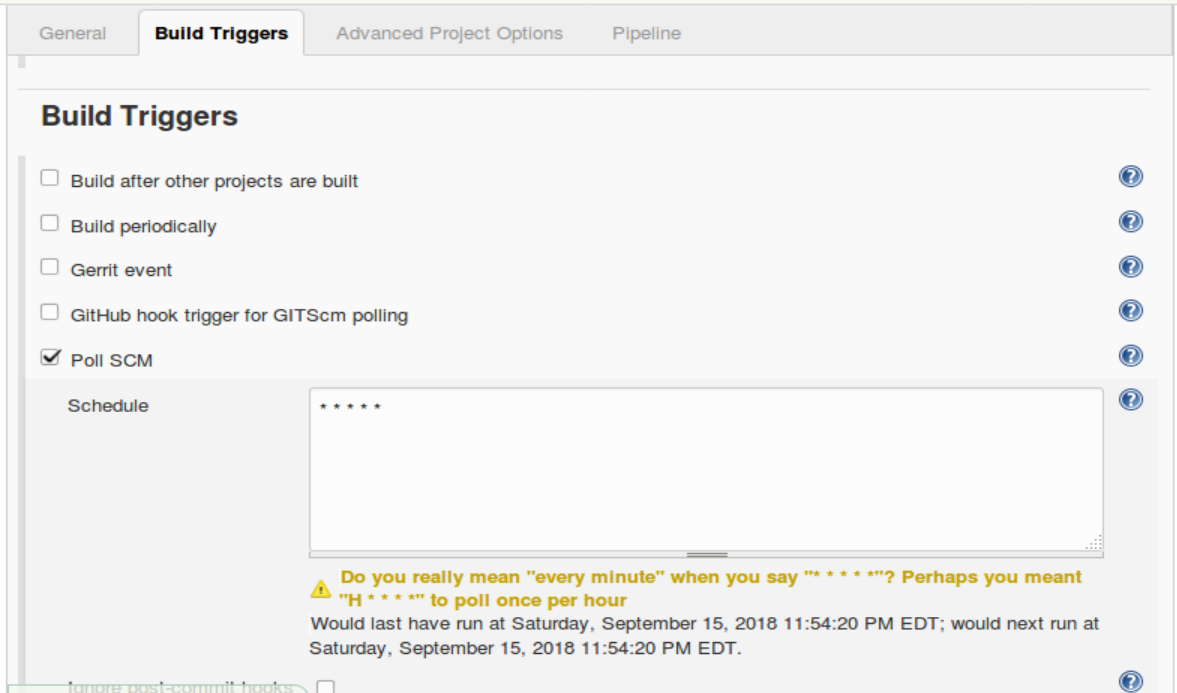
**Lab 1 – First CI Pipeline**

**Purpose: In this lab, we’ll get a quick start learning about CI by setting up a simple Jenkins pipeline job to build a project from our local repository on the VM.**

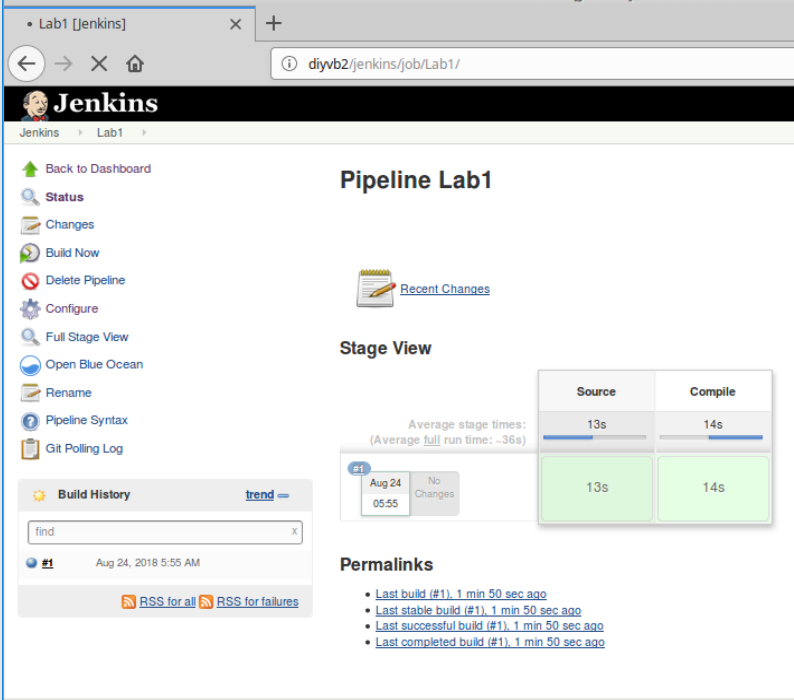
1. If you haven’t already, start up the CI virtual machine.
2. On the VM desktop, click on the Jenkins icon on the desktop and login to Jenkins with the userid and password supplied in the class.
3. Once logged in, click on New Item in the left menu. On the next screen, enter a name for the project such as “**pipe1**” and select a project type of “**Pipeline**”. Then select **OK**.



1. First, we will want to define what kind of events will trigger this build. For our purposes here, we’ll just scan the source repository periodically. Scroll down to the “**Build Triggers**” section and select “**Poll SCM**”.
2. This will open a text box labeled “**Schedule**”. This is where we’ll put in the representation for how often to scan the repository. Though we wouldn’t normally do this in production, we’ll tell it to scan every minute. In the field that pops up, enter **\* \* \* \* \***. (This is five asterisks separated by spaces.) This is short for “scan every minute of every day of the week of every day of every year”. You can ignore the yellow warning message.



1. With the build trigger setup, we can now put our pipeline in place to retrieve the source code and do a build when the build is triggered by a change in the area we are scanning for. The pipeline code you need is already saved in a file named “**CI**” on the desktop. Double-click on the icon for this file to open it. Then copy and paste the code from the file into the pipeline area “**Script**” box in your job.
2. Notice here that we have a “**Source**” stage with a **git** DSL step to get the code from our remote repository and then a “**Compile**” stage with a **sh** step to call the **gradle** build tool to attempt to build our code.
3. Save your changes to the pipeline code by clicking the “**Save**” button at the bottom of the screen. After a minute or two (be patient), Jenkins should try to automatically build your project. (You may need to refresh your browser to see the updates.)



1. Open a “**Terminal Emulator**” session (there is a shortcut on the desktop for that.)
2. We already have a copy of the repository referenced in our pipeline cloned down locally. In the terminal window change into the directory with that cloned copy.

**$ cd workshop**

1. Create a new file named readme.txt in the directory. You can put whatever content you want in it. To create it, you can either use the editor or just use a shortcut to redirect content in as shown below.

**$ gedit readme.txt**

**OR**

**$ echo some content > readme.txt**

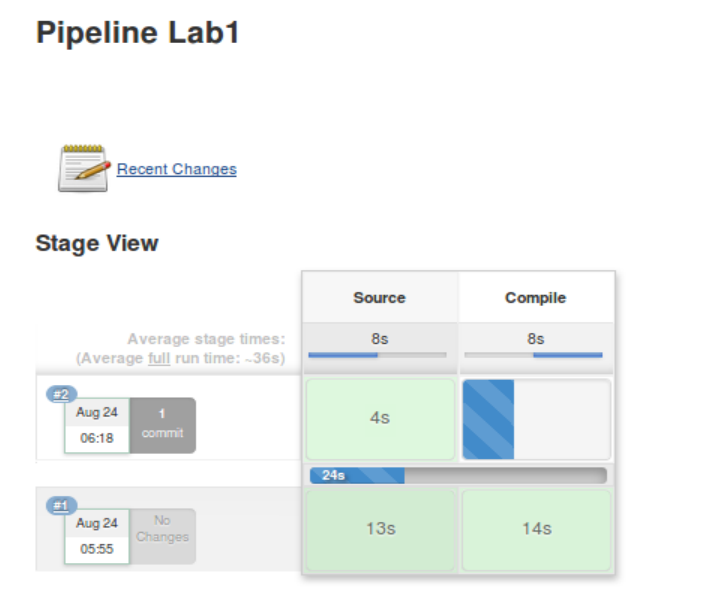
1. Now that we’ve made a change, we can stage it, commit it, and push it over into the repository. We’ll use the following Git commands to do this.

**git add readme.txt**

**git commit -m "Add readme file"**

**git push**

1. Now, switch back to the stage view page of the Jenkins job in the browser – if not already there (**http://localhost:8080/job/pipe1/**). WAIT FOR A MINUTE OR TWO. You do NOT need to do the “Build Now” here.
2. After a minute or two, Jenkins will detect your change and build the project. You can see the latest poll of the SCM by clicking on the “**Git Polling Log**” link in the left menu.



1. After it builds, you can click on the green circle with a checkmark in it next to #2 in the “**Build History**” window to see the console output where the project was built. Any further changes would be incorporated and built the same way.

**Optional:**

Normally, before we push our code, we would want to make sure that it builds cleanly in our local environment first. You can see this by going back to the terminal window, and in the **workshop** directory, run the gradle build command. (We add the “-x test” here to avoid running the unit tests right now.)

**gradle build -x test**

After a moment, you should see Gradle start up and run through the tasks to build the project. At the end, you should see a “BUILD SUCCESSFUL” message.

**Lab 2 – Adding in testing**

**Purpose: In this lab, we’ll add in some testing stages to our pipeline. We’ll also see how to run items in parallel.**

1. In the previous Compile stage in the pipeline script, we specifically told Gradle not to run the unit tests that it found by specifying the “-x test” target. Now we want to add in processing of several unit tests. Additionally, we want to run these in parallel. First, create a new stage for the unit testing with a parallel step to run our unit tests in parallel. (Note that **parallel** is lower-cased and those are **parentheses** after parallel, **NOT brackets**.)

If still in the **Stage View**, click **Configure** in the upper left menu. Add the lines in bold below to the configuration for your pipeline after the **Compile** stage.

stage('Compile') { // Compile and do unit testing

// Run gradle to execute compile and unit testing

sh 'gradle clean compileJava -x test'

}

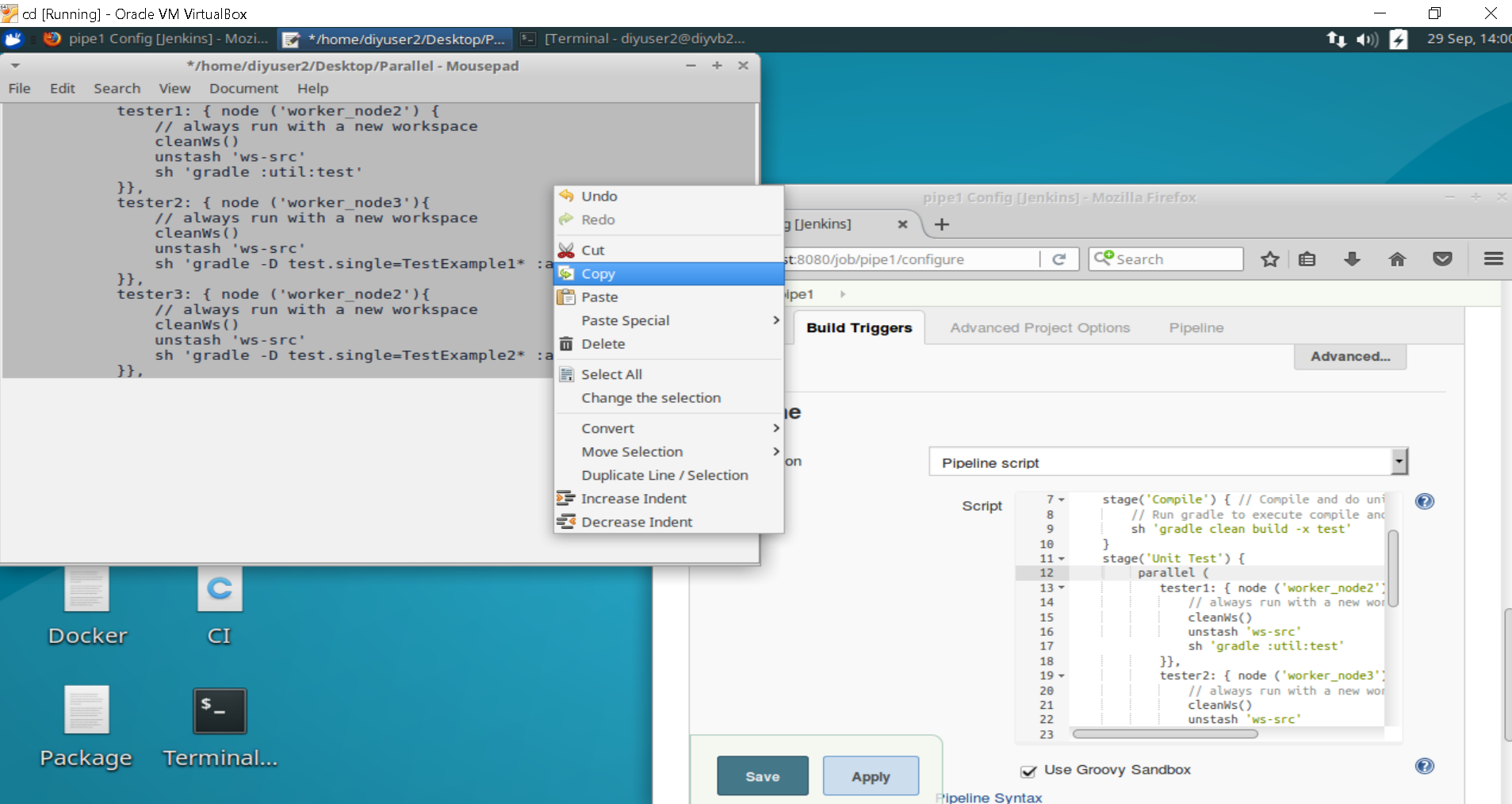
**stage('Unit Test') {**

**parallel (**

**)**

**}**

1. The parallel step takes a set of mappings with a key (name) and a value (code to execute in that parallel piece). To simplify setting this up, the code for the body of the parallel step (that runs the unit tests) is already done for you. It is in a text file on the VM desktop named **Parallel.** Open that file and copy and paste the contents between the opening and closing parentheses in the **parallel** step in the **Unit Test** stage.



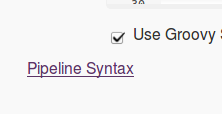


1. Look at the code in the parallel step. For each of the keys in the map (**tester1, tester2, and tester3**), we have a corresponding map value that consists of a block of code. The block of code has a node to run on (based on a selection by label), a step to clean the workspace, a step to “**unstash**”, and a call to the shared library Gradle command to run the particular test(s). The reason for the unstash command here is to get copies of the code onto this node for testing without having to pull it down again from source control (since we already did that.) This implies something was stashed. We’ll setup the stash next.
2. For purposes of having the necessary code to run the unit tests, we need to have the following pieces of our **workshop** project present on the testing nodes.

Subprojects **api**, **dataaccess**, and **util**

Project files **build.gradle** and **settings.gradle**

So we want to create a **stash** with them using the DSL’s **stash** command. To figure out the syntax, we’ll use the built-in **Snippet Generator** tool. Click on the “**Apply**” button to save your changes and then click on the **Pipeline Syntax** link underneath the editing window on the configuration page in Jenkins.



1. You’ll now be in the **Snippet Generator**. In the drop-down list of **Sample Steps**, find and select the **stash** command. A set of fields for the different named parameters associated with the stash command will pop up. You can click on the blue (with a ?) **help button** for any of the parameters to get more help for that one.

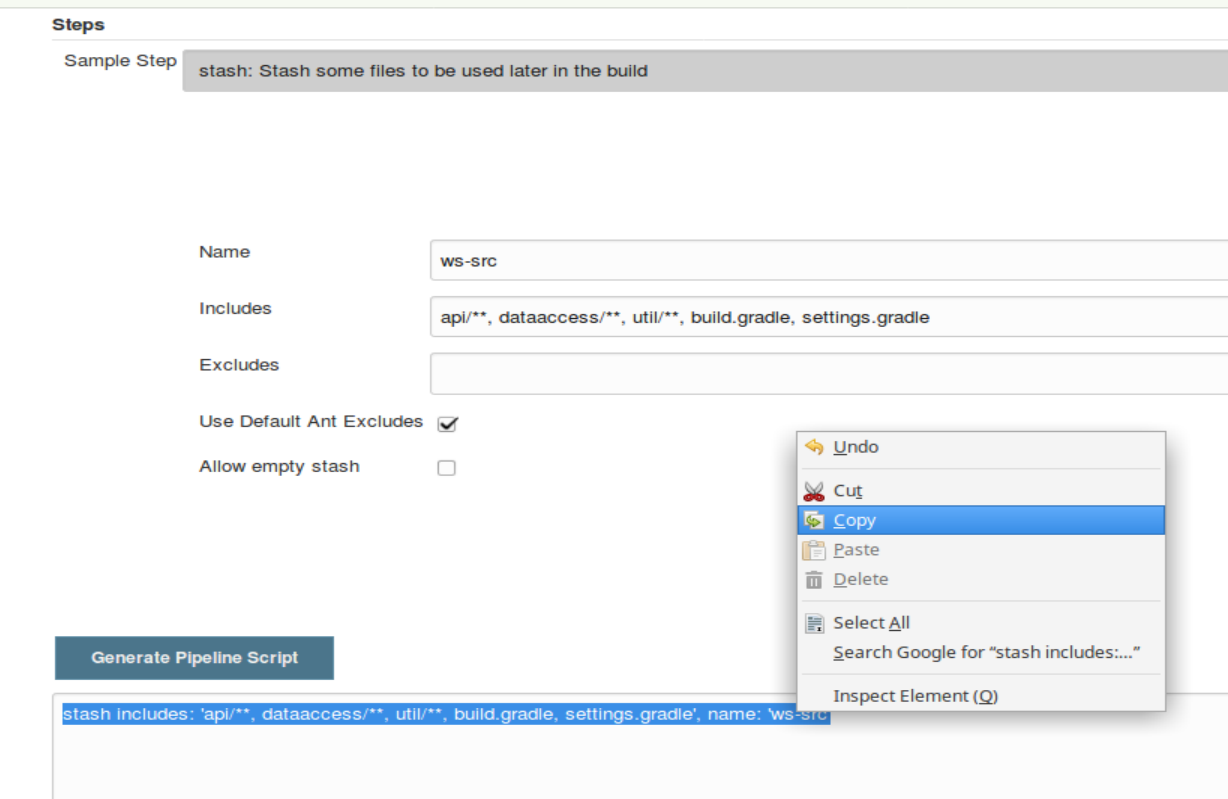
Type in the values for **Name** and **Includes** as follows:

Name: **ws-src**

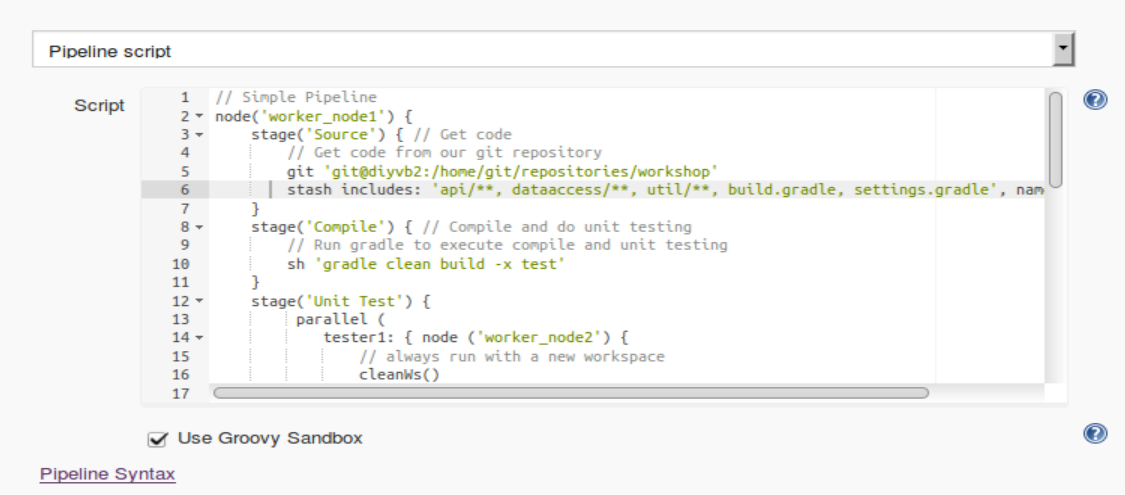
Includes: **api/\*\*, dataaccess/\*\*, util/\*\*, build.gradle, settings.gradle**

(The \*\* is a way to say all directories and all files under this area.)

Now, click the **Generate Pipeline Script** button. The generated code that you can use in your pipeline is shown in the box at the bottom of the screen.



1. Select and **copy** the text in the **Generate Pipeline Script** window. Switch back to the **configure** page for the **pipe1** job and **paste** the copied text directly under the **git** step in the **Source** stage.



1. **Save** your changes and select **Build Now** to execute a build of all the stages with the current code. In the Stage View, you’ll see the builds of our new stage.

Graphical user interface, table

Description automatically generated

1. Take a look at the **console output** for this run. In the **Build History** window to the left of the stage view, click on the green circle with the checkmark next to the latest run. Scroll down and look at the execution of the unit testing processes in parallel. This will be the lines starting with **[tester 1], [tester 2],** and **[tester 3**]. The output from the parallel processes will overlap each other in some spots.

**Lab 3 – Assembly**

**Purpose: In this lab, we’ll add the stage that assemble our artifact and adds in the semantic version**

1. Now, we'll add in code to our pipeline to assemble our target module from the pieces we have so far. We'll also build it with the semantic version numbers to version the resulting artifact.

Add the lines below as an **Assemble** stage in your pipeline configuration. (For convenience, there is also a **file on the desktop** named **Assemble** that you can **copy and paste** from.)

stage('Assemble')

{

// define semantic version values for built object

def workspace = env.WORKSPACE

def majorVersion = 1

def minorVersion = 0

def patchVersion = env.BUILD\_NUMBER

def buildStage = "SNAPSHOT"

def propertiesFile = "${workspace}/gradle.properties"

// write values out to gradle.properties file so gradle can pick them up

sh "sed -i '/MAJOR\_VERSION/c\\MAJOR\_VERSION='${majorVersion} ${propertiesFile}"

sh "sed -i '/MINOR\_VERSION/c\\MINOR\_VERSION='${minorVersion} ${propertiesFile}"

sh "sed -i '/PATCH\_VERSION/c\\PATCH\_VERSION='${patchVersion} ${propertiesFile}"

sh "sed -i '/BUILD\_STAGE/c\\BUILD\_STAGE='${buildStage} ${propertiesFile}"

// run the gradle assemble task to actually package up the war file

sh 'gradle clean -x test build assemble'

// stash built artifact to easily get to it in next stage

stash includes: 'web/build/libs/\*.war', name: 'latest-warfile'

}

1. **Save** your changes and start a build. After a few moments, you should have a successful build with another stage in your pipeline.

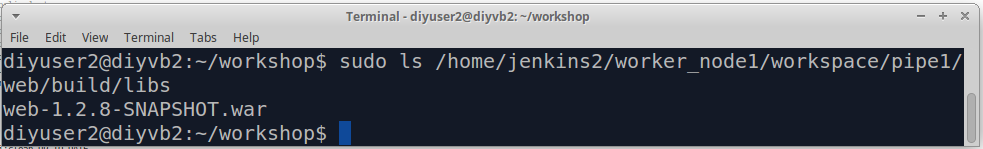
Graphical user interface

Description automatically generated

1. Look at the version of the war file that was produced. Go to a terminal window and run the following command (This assumes your workspace is **/home/jenkins2/worker\_node1/workspace/pipe1**. You can find your workspace path near the top of the console log. Look for **Running on** ..)

**$ sudo ls /home/jenkins2/worker\_node1/workspace/pipe1/web/build/libs**

The output will show the war file named with the version you specified.



**Lab 4 – Packaging**

**Purpose: In this lab, we’ll see how to package up our application with a deployment identifier – ready to deploy.**

1. First, since we may want to create multiple instances of our built application to swap in and out of our deployments, let’s add a new string parameter to our **pipe1** project.

In the project's configuration, in the **General** section, click on the item that says "This project is parameterized"

Text

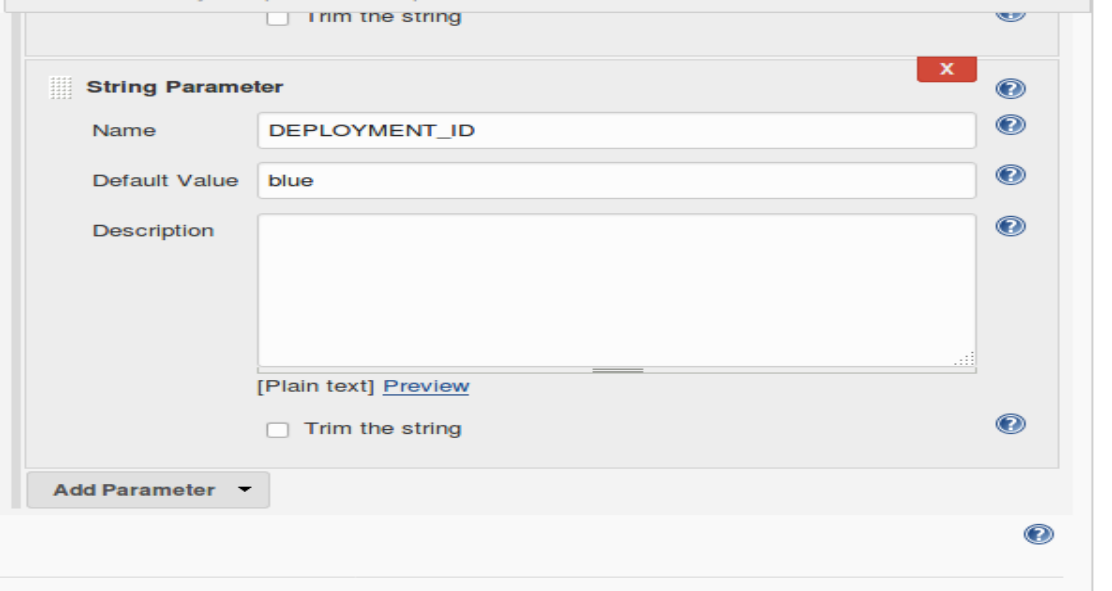
Description automatically generated with medium confidence

In the list that pops up select **String Parameter**.

Graphical user interface, application

Description automatically generated

The parameter should be called **DEPLOYMENT\_ID** and have a default value of “**blue**” (lower-case).



1. Next, we’ll create a stage block for the new stage. We’ll call this one **Package**. This stage will run on a node that has Docker and will do several things for us. It will get our Dockerfiles from source control, get the war we previously stashed, and then build the two Docker images (one for the web pieces and one for the database piece). Notice that we also include our new DEPLOYMENT\_ID in the image name. We’ll also clean out the workspace to make sure we don’t have any leftover files in it. Add the lines shown below into the stage within the node block. For your convenience, they are already typed in a file on the desktop named “**Package**”.

stage('Package')

{

// ensure we run on a node that has Docker

node('worker\_node1') {

// clean the workspace

cleanWs()

// download the Dockerfiles

git 'git@diyvb2:/home/git/repositories/roarv2-docker.git'

// retrieve the warfile we previously built

unstash 'latest-warfile'

// build the docker image for our database piece

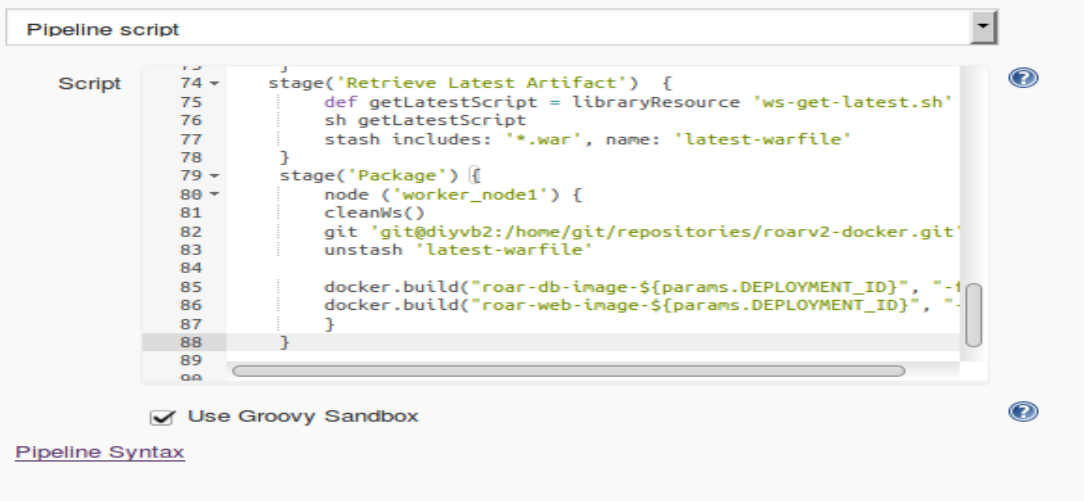
sh "docker build -t roar-db-image-${params.DEPLOYMENT\_ID} -f Dockerfile\_roar\_db\_image ."

// build the docker image for our webapp pulling in the war file

sh "docker build -t roar-web-image-${params.DEPLOYMENT\_ID} --build-arg warFile=web/build/libs/web\*.war -f Dockerfile\_roar\_web\_image ."

}

}

1. Now, your pipeline script should look something like this.  **Save** your changes. 
2. Then go ahead and do a “**Build**” to get a packaged version of the images out there. **Change the “Deployment ID” to be “green”.**

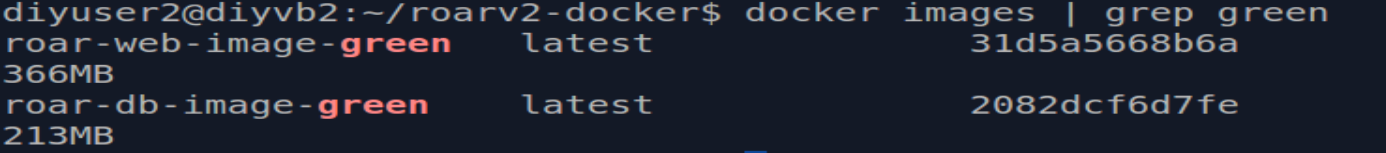
Graphical user interface, text, application, email

Description automatically generated

Graphical user interface, table

Description automatically generated

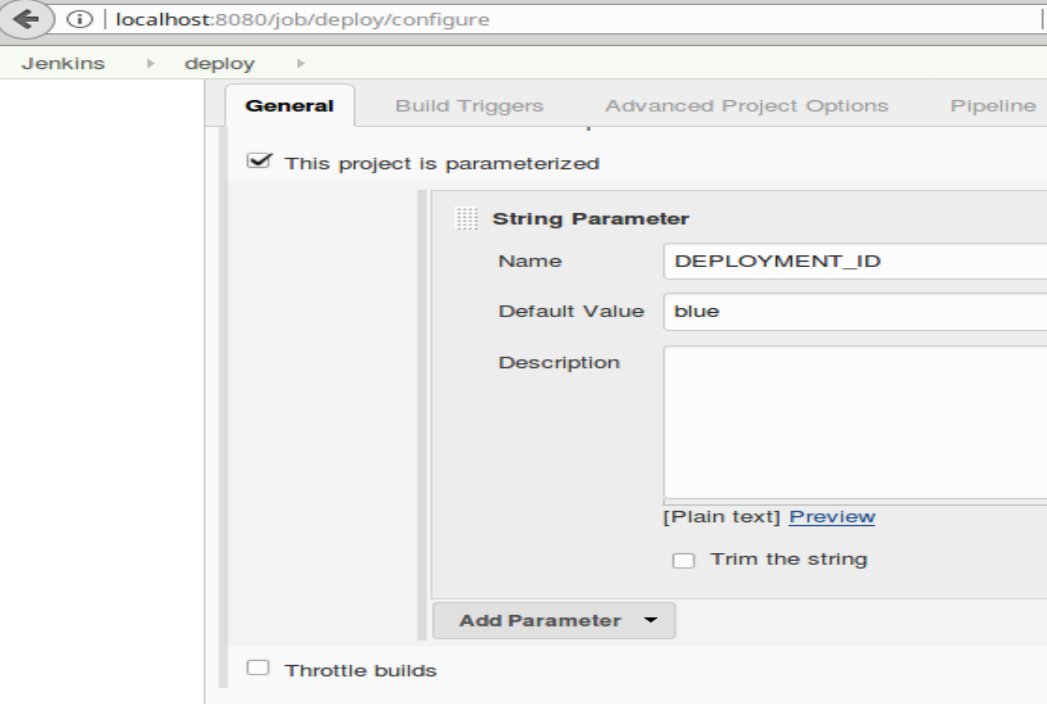
1. After this, you should be able to go to a **Terminal Emulator** window, issue the command “docker images” and see the “green” images listed (those with “green” in their name).



**Lab 5 – Deployment**

**Purpose: In this lab, we’ll look at a separate Jenkins job that we can run to deploy the appropriate version of our app.**

1. We’ve already setup a separate Jenkins job to help us deploy instances of our app. On the dashboard, open up the “Deploy” job and go to the “Configure” screen for it.
2. If you scroll down, you’ll see the parameters section. We want to be able to specify which version of our app we deploy, so we’ve added a string parameter called **DEPLOYMENT\_ID** with a default value again of “blue”.



1. Further down, in the Pipeline section, we have the actual deployment code. Since our instances are packaged as Docker containers, this code will do several things when we deploy:
   1. Stop any other running instances of our application.
   2. Remove any stopped containers.
   3. Get references to the two images we need for our application – selected based on the deployment id that is passed in as a parameter.
   4. Start the images linked together into a single app.
   5. Find and print the ip address where the newly deployed image is running.
2. The code is shown below (it is already in the job):

node('worker\_node1') {

stage('Deploy') {

sh "docker stop `docker ps -a --format '{{.Names}} \n\n'` || true"

sh "docker rm -f `docker ps -a --format '{{.Names}} \n\n'` || true"

dbImage = docker.image("roar-db-image-${params.DEPLOYMENT\_ID}")

webImage = docker.image("roar-web-image-${params.DEPLOYMENT\_ID}")

def dbContainer = dbImage.run("-p 3308:3306 -e MYSQL\_DATABASE='registry' -e MYSQL\_ROOT\_PASSWORD='root+1' -e MYSQL\_USER='admin' -e MYSQL\_PASSWORD='admin'")

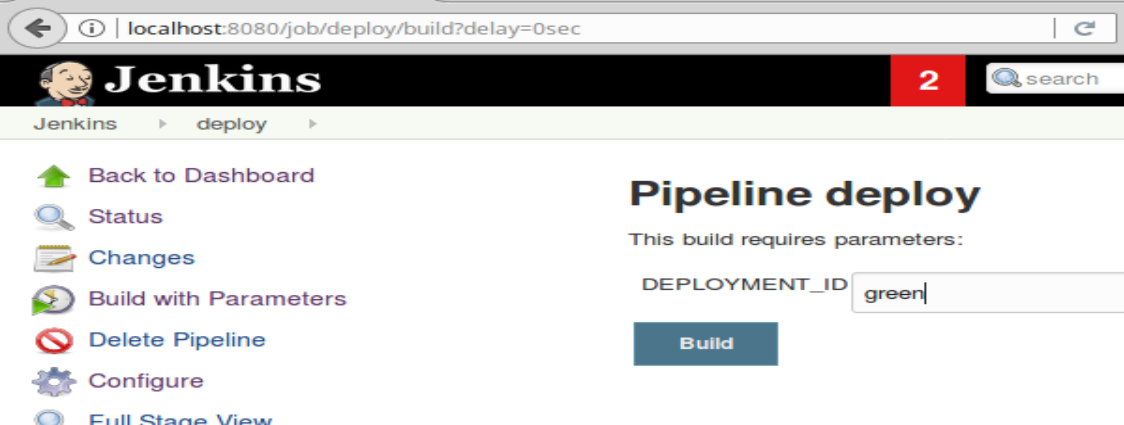
def webContainer = webImage.run("--link ${dbContainer.id}:mysql -p 8089:8080")

sh "docker inspect --format '{{.Name}} is available at http://{{.NetworkSettings.IPAddress }}:8080/roar' \$(docker ps -q -l)"

}

}

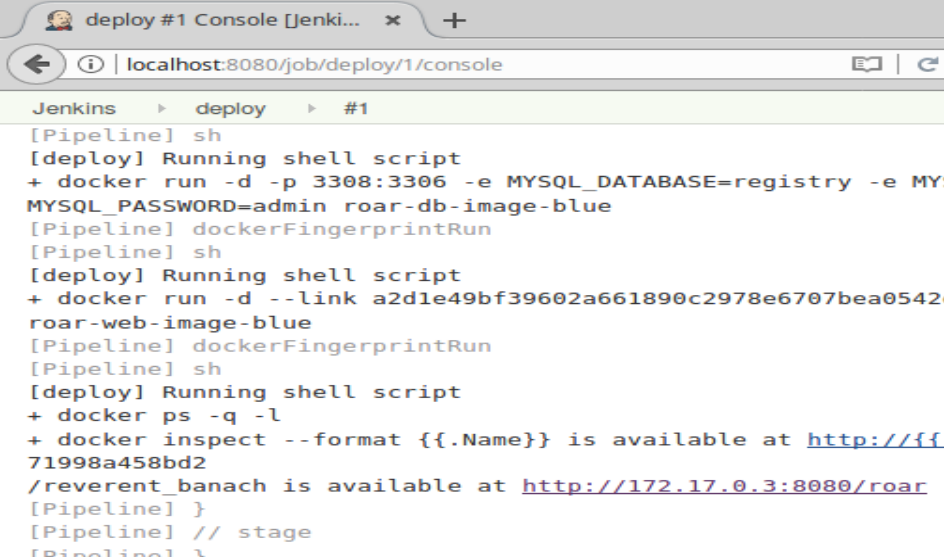
1. Let’s deploy an instance. Select“**Build with Parameters**”. Change the parameter value to “green”. This will deploy our last built set of images.



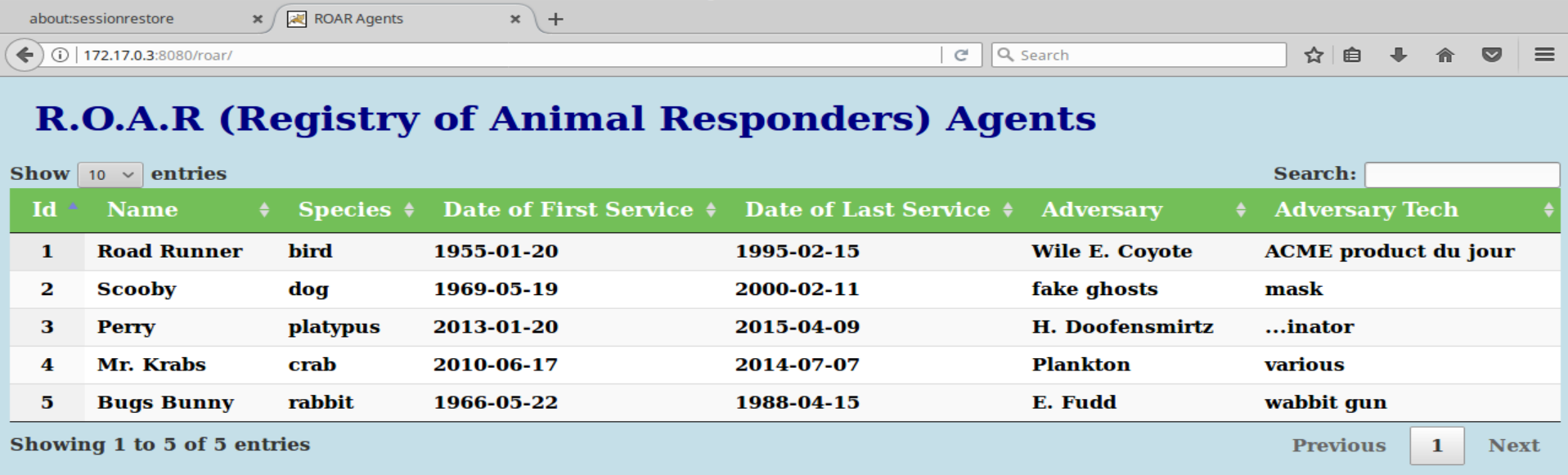
1. After the build runs, click on the Console Output link or the blue ball next to the build to open the Console Output screen.



1. Scroll down in the Console Log output until you see the link for the deployed app. Click on that link to see it running.



1. After clicking on it, you should see something like this:



1. Go back to Jenkins.

**Lab 6 – Optional but recommended: Full CD example**

**Purpose: In this lab, we’ll make a change in our app, and then deploy the new version to replace the existing one.**

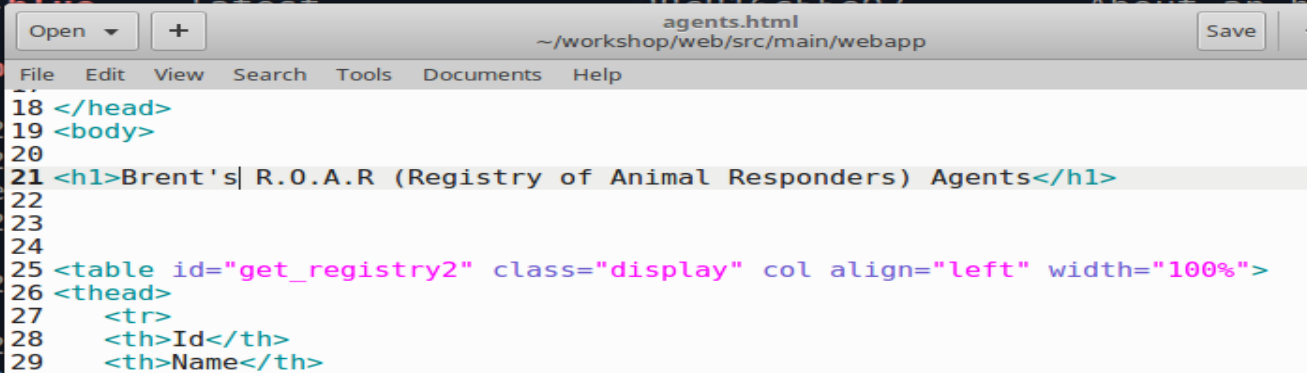
1. We’re going to modify a file and see the whole CD process in action. Go to a Terminal Emulator window and change into the “workshop” directory.

**cd workshop**

1. Edit the agents html file.

**gedit web/src/main/webapp/agents.html**

1. In the editor, update the title to be something different, such as adding your name (in the line that starts with “**h1**”).



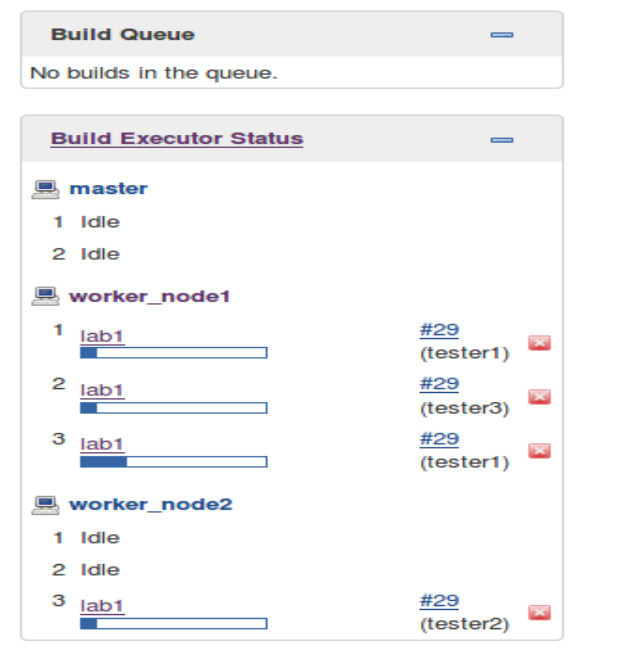
1. **Save** your changes and exit the editor.
2. Stage, commit, and push your update.

**git add web/src/main/webapp/agents.html**

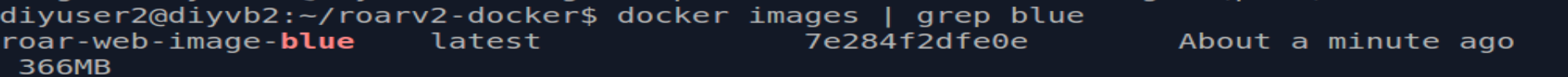
**git commit -m “Update”**

**git push**

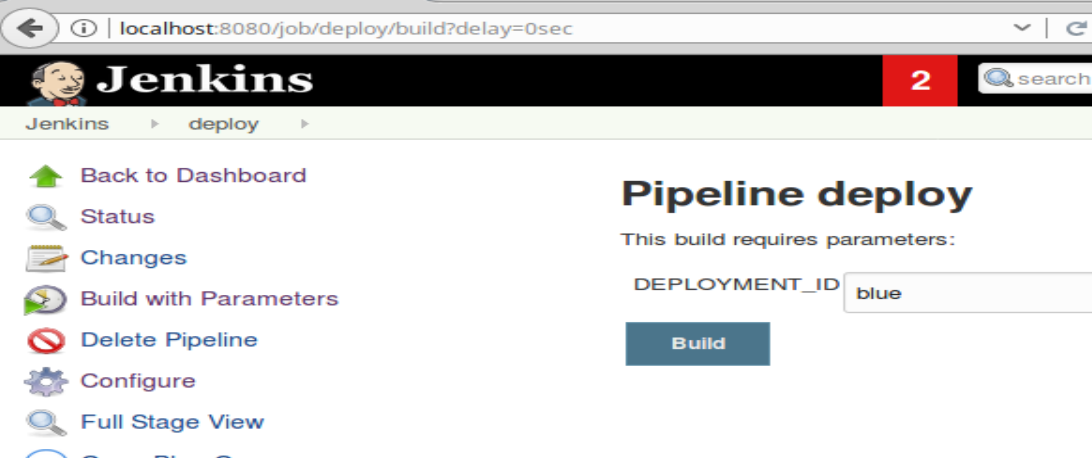
1. Wait. Be patient. After a minute or two, you should see the CI process kick in and your build start.



1. When this completes, we will have a new web docker image (with a deployment type of “blue” since that was our default).



1. Now, let’s deploy the “blue” version with our new change. Run the Jenkins “deploy” job and enter (or accept) the default “blue” value.



1. Once again, you can go to the Console Output, find the link that is displayed and view the updated version of the app. (Note that you may need to force a refresh of the cached page.)

