Getting Started with Continuous Delivery

Revision 1.2 – 4/14/20

Brent Laster

IMPORTANT NOTES:

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

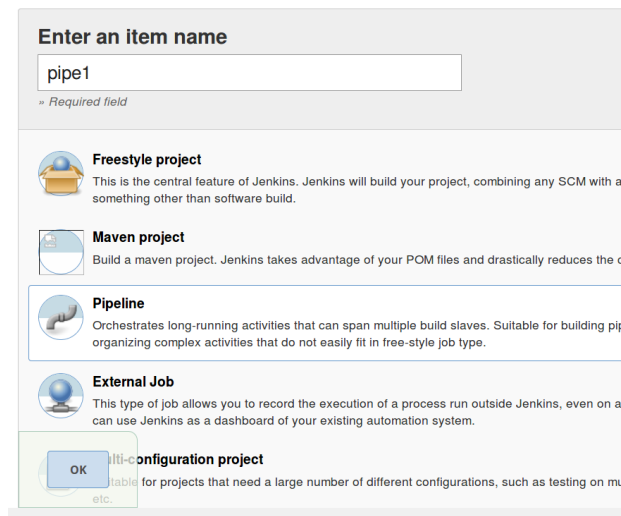**(If you have not setup the VM yet, checkout the setup doc at
https://github.com/brentlaster/safaridocs/blob/master/cd-labs.pdf)**

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**.

4. 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**".

5. 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.

© 2020 Brent Laster

6. 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.

7. 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.

8. Save your changes to the pipeline code by clicking the "**Save**" button at the bottom of the screen. Click the "**ENABLE AUTO REFRESH**" link in the upper right corner. After a minute or two (be patient), Jenkins should try to automatically build your project.
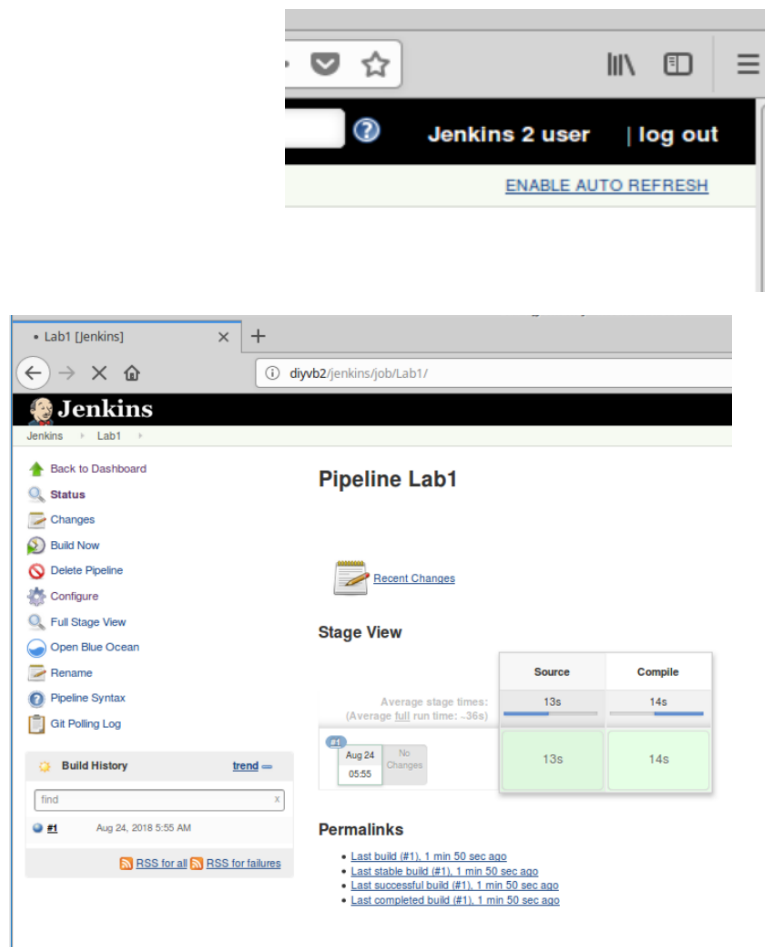
9. Open a "**Terminal Emulator**" session (there is a shortcut on the desktop for that.)

10. 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

11. 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.

© 2020 Brent Laster

gedit readme.txt

OR

echo   some content > readme.txt

12. 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

13. 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.

14. 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.

Pipeline Lab1
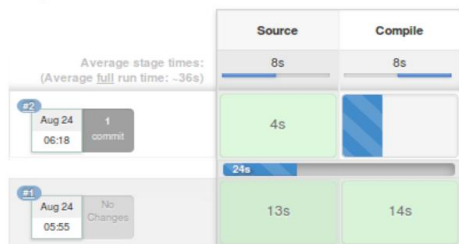
15. After it builds, you can click on the blue ball next to #2 in the "**Build History**" window to see the console output where the project was built.   Any further changes would be incorporated and built the same way.

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.

© 2020 Brent Laster

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. Also, we will go ahead and create a stage for the integration testing. (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 (

    )

}
stage('Integration Test')
{

}
```

2. 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.

© 2020 Brent Laster
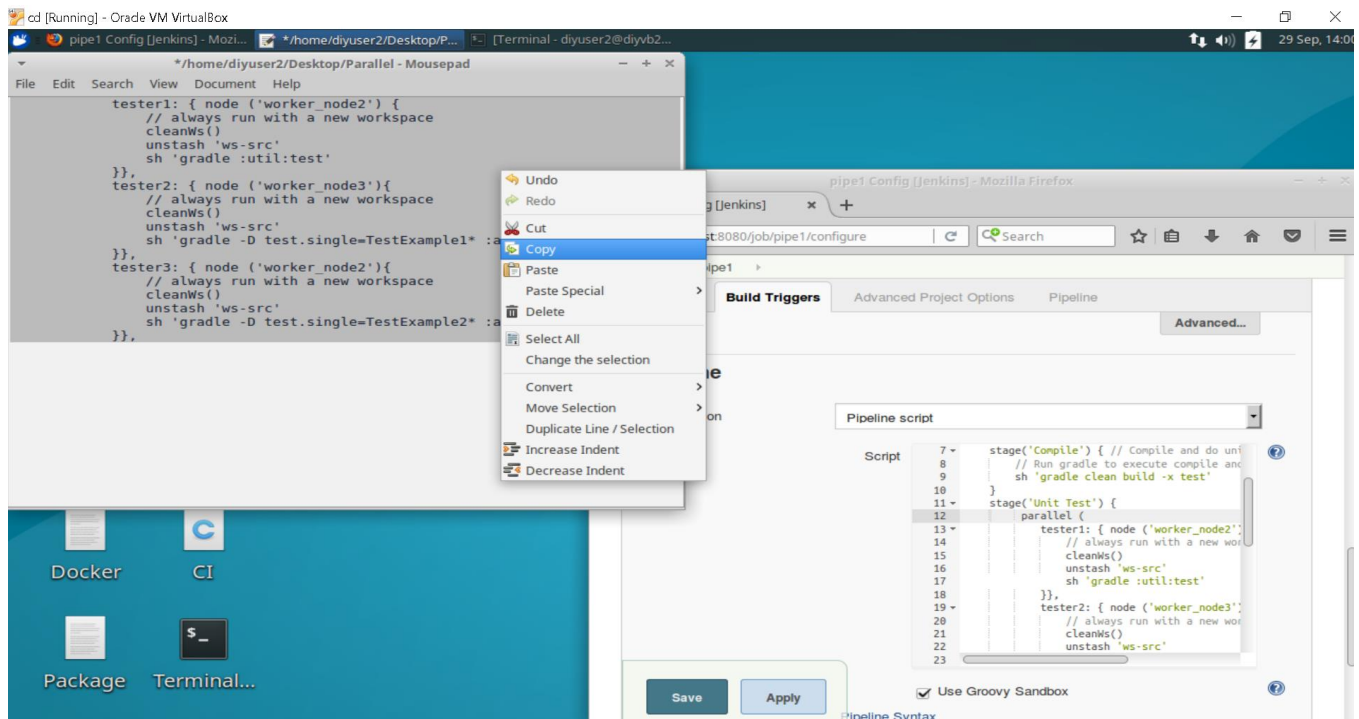
3. Look at the code in the parallel step. For each of the keys in the map (**tester1, tester2, and tester3**), we have a corresponding map value that consists of a block of code. The block of code has a node to run on (based on a selection by label), a step to clean the workspace, a step to "**unstash**", 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.

4. 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.
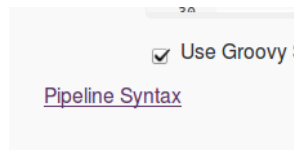
5. You'll now be in the **Snippet Generator**. In the drop-down list of **Sample Steps**, find and select the **stash** command. A set of fields for the different named parameters associated with the stash command will pop up. You can click on the blue (with a ?) **help button** for any of the parameters 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.

6. 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.


```
Pipeline script

Script
        1    // Simple Pipeline
        2 ▾  node('worker_node1') {
        3 ▾      stage('Source') { // Get code
        4              // Get code from our git repository
        5              git 'git@diyvb2:/home/git/repositories/workshop'
        6            | stash includes: 'api/**, dataaccess/**, util/**, build.gradle, settings.gradle', nam
        7          }
        8 ▾      stage('Compile') { // Compile and do unit testing
        9              // Run gradle to execute compile and unit testing
       10              sh 'gradle clean build -x test'
       11          }
       12 ▾      stage('Unit Test') {
       13            | parallel (
       14 ▾              tester1: { node ('worker_node2') {
       15                    // always run with a new workspace
       16                    cleanWs()
       17

☑ Use Groovy Sandbox

Pipeline Syntax
```

7. Finally, we'll add the commands to run the integration tests. We'll add a line to setup the test database we use for the integration testing and then invoke Gradle to run the **integrationTest** task that we have defined in our **build.gradle** file. Enter or copy and paste the lines in bold below into the **Integration Test** stage in the pipeline code.

```
stage('Integration Test') { // setup and run integration testing
    // set up integration database
    sh "mysql -uadmin -padmin registry_test < registry_test.sql"
    sh 'gradle integrationTest'
}
```

8. **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 stages.

Stage View

	Source	Compile	Unit Test	Integration Test
Average stage times:	2s	18s	52s	16s
#5 Apr 11 22:45 No Changes	1s	10s	58s	16s

9. 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 blue ball 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 create the stages that assemble our artifact. We'll also see how to parameterize our job and use the parameters in a function that is pulled in from GitHub to modify the version on an artifact that we produce.

1. Start out by adding the empty stage definition for an **Assemble** stage in our pipeline code. This should come after the **Integration Test** stage but before the closing bracket. Lines to add are shown in bold below.

```
      sh "mysql -uadmin -padmin registry_test < registry_test.sql"
       sh 'gradle integrationTest'
  }
     }
     stage('Assemble')
     {

     }
    }
```

2. Since artifacts should be versioned, we'll add some parameters to our Jenkins job, so we can modify the versions of the war file that we produce if we want.

   Click on the **General** tab or scroll up to the top of the configure page. Check (click on) the box that says, **"This project is parameterized."**

   Click on the **Add Parameter** button and select **String Parameter** from the drop-down. Fill in the values as follows:

   Name: MAJOR_VERSION
   Default Value: 1

   Repeat the process to add 3 more **String Parameters** with the following settings:

   Name: MINOR_VERSION
   Default Value: 1

   Name: PATCH_VERSION
   Default Value: $BUILD_NUMBER

   Name: BUILD_STAGE
   Default Value: SNAPSHOT

General   Build Triggers   Advanced Project Options   Pipeline

☑ This project is parameterized

String Parameter   X

Name   MAJOR_VERSION

Default Value   1

Description

[Plain text] Preview

String Parameter   X

Name   MINOR_VERSION

Default Value   1

Description

[Plain text] Preview

String Parameter   X

Name   PATCH_VERSION

Default Value $BUILD_NUMBER

Description

[Plain text] Preview

String Parameter   X

Name   BUILD_STAGE

Default Value   SNAPSHOT

Description

Save   Apply

3. We now have these values and can change them if needed. We want to incorporate them as the version number for the war file that we produce out of this project. We can do that indirectly by updating the **gradle.properties** file in our project with the values. This will update the file with the values we pass in and the Gradle build tool will then version our artifact with those values.

We can use some shell commands and the tool **sed** to accomplish this easily. For demonstration purposes, we've put those commands into a function and put that code into a repository on **GitHub.** To see the code, open a new browser tab/window and go to the URL below:

https://github.com/brentlaster/utilities

Then, go into the **jenkins/pipeline** folder.   Click on the **updateGradleProperties.groovy** file.

© 2020 Brent Laster

Brent Laster

Notice here that we are using the pipeline DSL step **sh** as part of our code. Any valid DSL step can be used in such a function.

4. Next, we'll add the code in our pipeline stage to bring this in via the **fileLoad** step. There are a couple of parts to this.

First, we define a variable that points to our project's workspace in Jenkins. This is so we can make sure to update the gradle.properties file in the correct path.

Next we define a variable to point to the function we load from GitHub.

Finally, we invoke the function, passing in references to our parameters.

Add the lines below in bold into the **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') {
    // assemble war file
    def workspace = env.WORKSPACE
    def setPropertiesProc = fileLoader.fromGit('jenkins/pipeline/updateGradleProperties',
            'https://github.com/brentlaster/utilities.git', 'master', null, '')

    setPropertiesProc.updateGradleProperties("${workspace}/gradle.properties",
            "${params.MAJOR_VERSION}",
            "${params.MINOR_VERSION}",
            "${params.PATCH_VERSION}",
            "${params.BUILD_STAGE}")
}
```

© 2020 Brent Laster

5. There is one last step for this stage. We need to add a call to Gradle to invoke the **assemble** task. To get everything ready, we will also call the **build** task. But we will tell Gradle explicitly to not run the **test** task (via the -x switch).

   Add the line in bold at the bottom of your task, after the **setPropertiesProc** call.

```
setPropertiesProc.updateGradleProperties("${workspace}/gradle.properties",

        "${params.MAJOR_VERSION}",
        "${params.MINOR_VERSION}",
        "${params.PATCH_VERSION}",
        "${params.BUILD_STAGE}")

        sh 'gradle -x test build assemble'

}
```

6. **Save** your changes and click on the **Build with Parameters** button in the left-hand menu. Note that since we have parameters, we must tell Jenkins whether we want to use the defaults or enter new values.
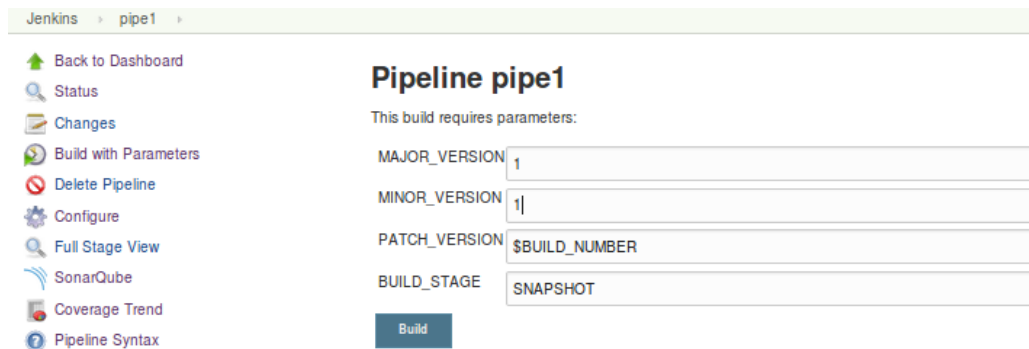
If you want to change a value, you can. Just remember for the future stages, always make the values the same or higher for future runs. (i.e. If you change **MINOR_VERSION** to a **2** now, then set it as a **2 or higher** in subsequent runs.)

When you are ready to build, click the **Build** button.

7. You should have a successful build with another stage in your pipeline.
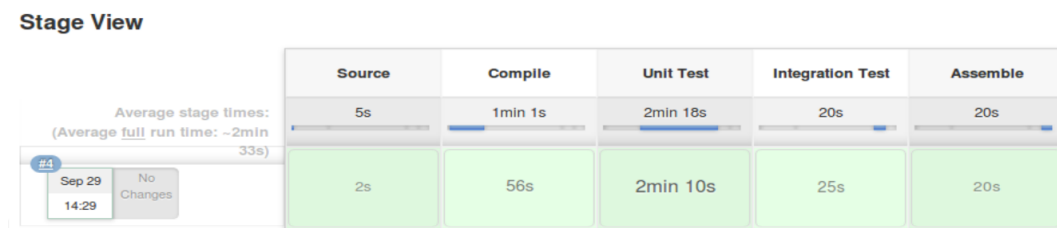
You can open the console log for the most recent build and find the place where the code from github was pulled in.
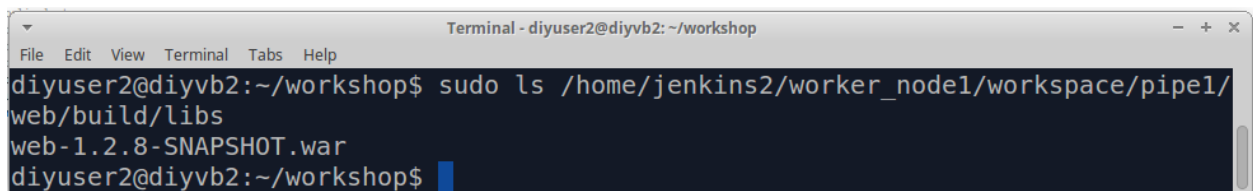
© 2020 Brent Laster

```
[Pipeline] deleteDir
[Pipeline] echo
Checking out https://github.com/brentlaster/utilities.git, branch=master
[Pipeline] checkout
Cloning the remote Git repository
Cloning repository https://github.com/brentlaster/utilities.git
 > git init /home/jenkins2/worker_node3/workspace/pipe1/libLoader # timeout=10
Fetching upstream changes from https://github.com/brentlaster/utilities.git
 > git --version # timeout=10
 > git fetch --tags --progress https://github.com/brentlaster/utilities.git +refs/heads/*:refs/remotes/origin/*
 > git config remote.origin.url https://github.com/brentlaster/utilities.git # timeout=10
 > git config --add remote.origin.fetch +refs/heads/*:refs/remotes/origin/* # timeout=10
 > git config remote.origin.url https://github.com/brentlaster/utilities.git # timeout=10
```

Optional:

If you want to see the version of the war file that was produced, you can 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.

```
Terminal - diyuser2@diyvb2:~/workshop                          – + ×
File  Edit  View  Terminal  Tabs  Help
diyuser2@diyvb2:~/workshop$ sudo ls /home/jenkins2/worker_node1/workspace/pipe1/
web/build/libs
web-1.2.8-SNAPSHOT.war
diyuser2@diyvb2:~/workshop$
```

Lab 4 – Storing and Tracking Artifacts

Purpose: In this lab, we'll see how to integrate Jenkins and Gradle with Artifactory to store our war file artifact into an Artifactory repository.

1. First, as we always do, create a stage block for the new stage.   We'll call this one **Publish Artifacts**.

```
        sh 'gradle -x test build assemble'
    }
    stage ('Publish Artifacts')
    {

    }
```

2. The **Artifactory plugin** provides a default **Artifactory** object that we can use, as well as several steps for working with Jenkins and various build systems, such as **Gradle** and **Maven**.   We next look at the steps you need to setup and execute the publish to Artifactory using Gradle.

© 2020 Brent Laster

Go back to the browser tab where you were configuring your **pipe1** job (where you added the new **Publish Artifacts** stage).

Add these commands to your new stage by copying them from the **file on your desktop named Publish.**

The explanations below are just fyi on what they are doing.

a. We point the Artifactory object to our installed instance. (Here, **LocalArtifactory** is the **Server ID** we gave it in the global configuration.)

   def server = Artifactory.server "LocalArtifactory"

b. We create a new Gradle object that knows about the Artifactory instance and point it to the gradle version we have defined in our global configuration.

   def artifactoryGradle = Artifactory.newGradleBuild()

   artifactoryGradle.tool = "gradle4"

c. We tell Artifactory which Artifactory repositories to use for **deploying** (storing) objects into (**libs-snapshot-local**) and which one to **resolve** (retrieve) dependencies from (**remote-repos**).

   artifactoryGradle.deployer repo:'libs-snapshot-local', server: server

   artifactoryGradle.resolver repo:'remote-repos', server: server

d. We tell Artifactory where to publish info about each build.

   def buildInfo = Artifactory.newBuildInfo()

   buildInfo.env.capture = true

e. We want the publish to use Maven type descriptors and not to publish the individual jar files we create (just the war file we end up with).

   artifactoryGradle.deployer.deployMavenDescriptors = true

   artifactoryGradle.deployer.artifactDeploymentPatterns.addExclude("*.jar")

f. We tell Jenkins that we aren't already loading the related Gradle plugin (com.jfrog.artifactory) in our Gradle script.
   artifactoryGradle.usesPlugin = false

g. We run the Artifactory Gradle build and invoke Gradle's built-in **artifactoryPublish** task.

   artifactoryGradle.run buildFile: 'build.gradle', tasks: 'clean artifactoryPublish', buildInfo: buildInfo

h. We tell Artifactory to actually publish the build info as well.

   server.publishBuildInfo buildInfo

Now, you should have your completed **Publish Artifacts** stage.

```
Pipeline script

Script    75        stage ('Publish Artifacts')
          76 ▾      {
          77            def  server = Artifactory.server "LocalArtifactory"
          78            def artifactoryGradle = Artifactory.newGradleBuild()
          79            artifactoryGradle.tool = "gradle4"
          80            artifactoryGradle.deployer repo:'libs-snapshot-local', se
          81            artifactoryGradle.resolver repo:'remote-repos', server: s
          82
          83            def buildInfo = Artifactory.newBuildInfo()
          84            buildInfo.env.capture = true
          85            artifactoryGradle.deployer.deployMavenDescriptors = true
          86            artifactoryGradle.deployer.artifactDeploymentPatterns.add
          87            artifactoryGradle.usesPlugin = false
          88
          89            artifactoryGradle.run buildFile: 'build.gradle', tasks:
          90            server.publishBuildInfo buildInfo
          91
```

3. Now **Save** your updates and **Build with Parameters**. Remember that if you changed any parameter values last time, make them the same or higher so this will be the latest.

    After this runs, you'll see the newest stages in the stage view.

Stage View

	Source	Compile	Unit Test	Integration Test	Assemble	Publish Artifacts
Average stage times: (Average full run time: ~2min 50s)	4s	58s	2min 8s	19s	19s	42s
#5 Sep 29 14:42 · No Changes	2s	44s	1min 49s	17s	18s	42s

Optional:

   We already have an Artifactory instance set up and running on this machine and it is integrated with Jenkins. You can see how Jenkins is integrated with it by going to **Manage Jenkins**, then to **Configure System**, and scroll down until you find the **Artifactory** section.

   You can see that we have it running at http://localhost:8081/artifactory .

   If you want, you can click the **Test Connection** button to see it test the connection to the system.

© 2020 Brent Laster

Lab 5 – Packaging – part 1

Purpose: In this lab, we'll see how to retrieve the latest version from Artifactory and one way to include code from an external source.

1. For the first part of packaging, we'll want to retrieve the desired version of our artifact from Artifactory.   For simplicity here, we'll just get the latest version. We may not always want to retrieve the latest version, but we'll do it here to show how to use an external script here since the free version of Artifactory doesn't support this.

   We again are storing this code in an external file – in this case in the resources directory of a Shared Library. To be able to access it, we need to import the shared library into our script.

   Add this line in bold below before the **node** line at the top of your pipeline: (the line is "@Library('Utilities2')_")

   ```
   // Simple Pipeline
   @Library('Utilities2')_
   node('worker_node1') {
      stage('Source') { // Get code
   ```


2. To invoke this code in our stage, we will use the **libraryResource** step to load it. Then we can just pass the code to the **sh** step and have it executed. Add the stage below in bold into your pipeline.
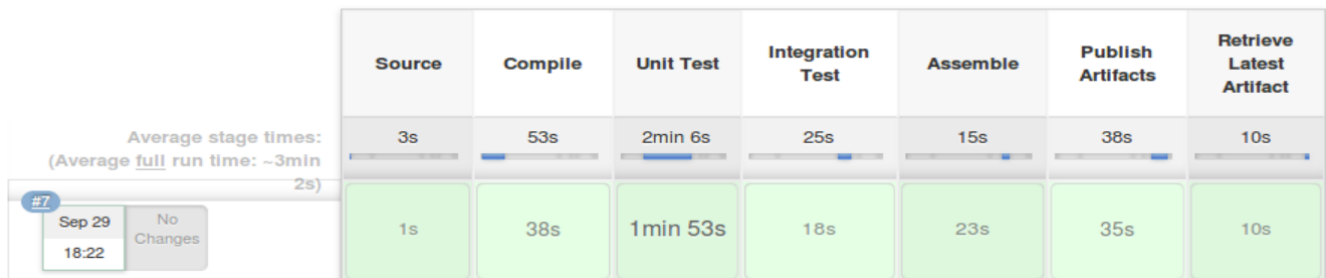
```
stage('Retrieve Latest Artifact') {
     def getLatestScript = libraryResource 'ws-get-latest.sh'
     sh getLatestScript
}
```

3. There's one other piece we want to add here. Since this stage is retrieving the latest artifact, we want to keep that to pass on to later stages. As we've seen, one way to do this is with the **stash** step. Add the stash step in bold below into your stage.

```
stage('Retrieve Latest Artifact') {
     // get the latest artifact out
     def getLatestScript = libraryResource 'ws-get-latest.sh'
     sh getLatestScript
     stash includes: '*.war', name: 'latest-warfile'
}
```

4. **Save** your changes and **Build With Parameters** as before**.**

Stage View

	Source	Compile	Unit Test	Integration Test	Assemble	Publish Artifacts	Retrieve Latest Artifact
Average stage times: (Average full run time: ~3min 2s)	3s	53s	2min 6s	25s	15s	38s	10s
#7 Sep 29 No 18:22 Changes	1s	38s	1min 53s	18s	23s	35s	10s

You can also go into the console log to see the output and what it pulled from Artifactory. (The links generated in the log are actually usable. If you click on one, you can download the artifact as well.)

© 2020 Brent Laster

```
[Pipeline] { (Retrieve Latest Artifact)
[Pipeline] libraryResource
[Pipeline] sh
[pipe1] Running shell script
+ [ -e *.war ]
+ server=http://localhost:8081/artifactory
+ repo=libs-snapshot-local
+ name=web
+ artifact=com/demo/pipeline/web
+ path=http://localhost:8081/artifactory/libs-snapshot-local/com/demo/pipeline/web
+ sed s/.*<latest>\([^<]*\)<\/latest>.*/\1/
+ grep latest
+ curl -s http://localhost:8081/artifactory/libs-snapshot-local/com/demo/pipeline/web/maven-metadata.xml
+ version=1.4.13-SNAPSHOT
+ sed s/.*<value>\([^<]*\)<\/value>.*/\1/
+ head -1
+ sort -t- -k2,2nr
+ grep <value>
+ curl -s http://localhost:8081/artifactory/libs-snapshot-local/com/demo/pipeline/web/1.4.13-SNAPSHOT/maven-metadata.xml
+ build=1.4.13-20170413.040403-1
+ war=web-1.4.13-20170413.040403-1.war
+ url=http://localhost:8081/artifactory/libs-snapshot-local/com/demo/pipeline/web/1.4.13-SNAPSHOT/web-1.4.13-20170413.040403-1.war
+ echo http://localhost:8081/artifactory/libs-snapshot-local/com/demo/pipeline/web/1.4.13-SNAPSHOT/web-1.4.13-20170413.040403-1.war
http://localhost:8081/artifactory/libs-snapshot-local/com/demo/pipeline/web/1.4.13-SNAPSHOT/web-1.4.13-20170413.040403-1.war
+ wget -q -N http://localhost:8081/artifactory/libs-snapshot-local/com/demo/pipeline/web/1.4.13-SNAPSHOT/web-1.4.13-20170413.040403-1
[Pipeline] stash
Stashed 1 file(s)
[Pipeline] }
```
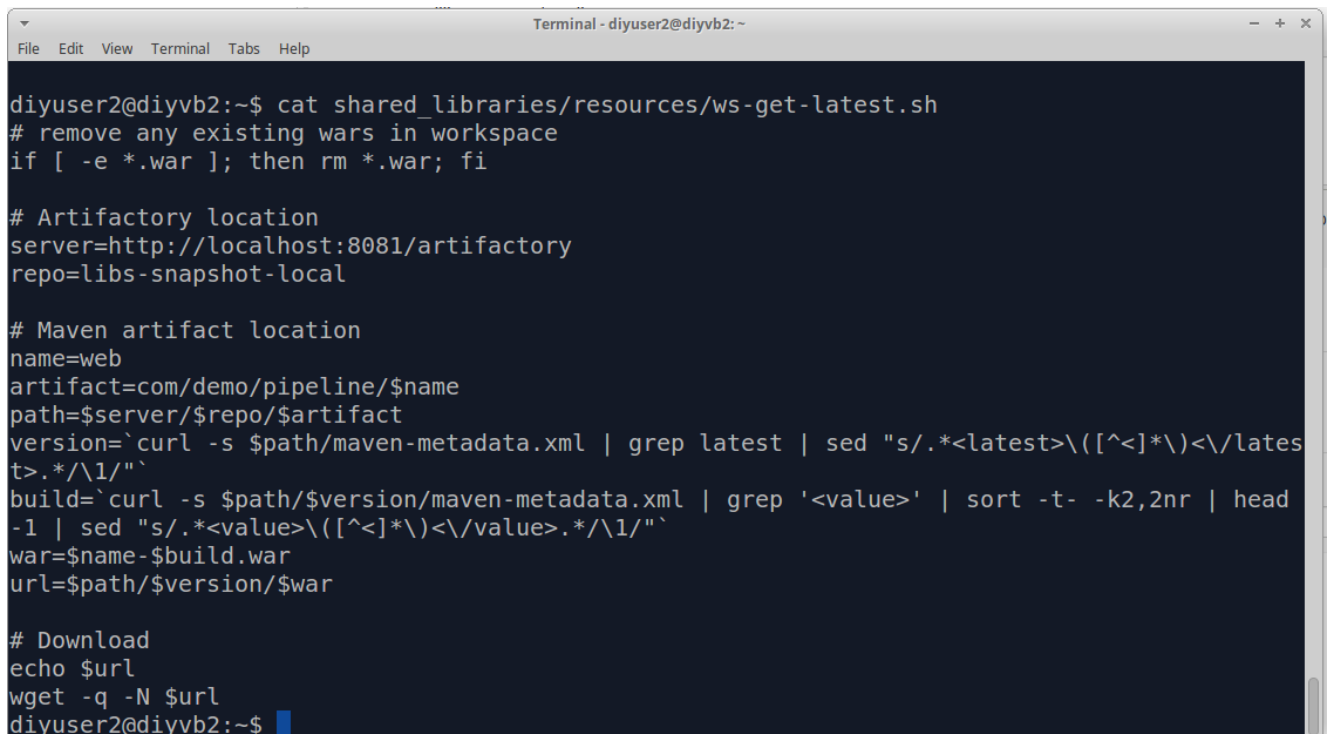
Optional:

Look at the code we're using for this task by running the following command in a terminal session:

cat shared_libraries/resources/ws-get-latest.sh

Essentially, we have a set of shell commands that parse pom files in Artifactory and extract the details about what version is latest, and then we drill down into Artifactory with that information, find the latest version and retrieve it.

```
diyuser2@diyvb2:~$ cat shared_libraries/resources/ws-get-latest.sh
# remove any existing wars in workspace
if [ -e *.war ]; then rm *.war; fi

# Artifactory location
server=http://localhost:8081/artifactory
repo=libs-snapshot-local

# Maven artifact location
name=web
artifact=com/demo/pipeline/$name
path=$server/$repo/$artifact
version=`curl -s $path/maven-metadata.xml | grep latest | sed "s/.*<latest>\([^<]*\)<\/latest>.*/\1/"`
build=`curl -s $path/$version/maven-metadata.xml | grep '<value>' | sort -t- -k2,2nr | head -1 | sed "s/.*<value>\([^<]*\)<\/value>.*/\1/"`
war=$name-$build.war
url=$path/$version/$war

# Download
echo $url
wget -q -N $url
diyuser2@diyvb2:~$
```

© 2020 Brent Laster

Lab 6 – Packaging – Part 2

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. The parameter should be called **DEPLOYMENT_ID** and have a default value of "**blue**" (lower-case).

2. Next, we'll create a stage block for the new stage.   We'll call this one **Package**.

```
            stash includes: '*.war', name: 'latest-warfile'
    }
    stage ('Package')
    {

    }
```

3. We're going to create Docker images to package up our application. To do this, we'll use a node that we're sure can run Docker on it. Add a node definition to use 'worker_node1'.

```
    stage ('Package')

    {
        node ('worker_node1') {

        }

    }
```

4. To package up our app, we will want to 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**".

```
node ('worker_node1') {
    cleanWs()

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

    unstash 'latest-warfile'

    sh "docker build -t roar-db-image-${params.DEPLOYMENT_ID} -f Dockerfile_roar_db_image ."

    sh "docker build -t roar-web-image-${params.DEPLOYMENT_ID} --build-arg warFile=web*.war -f Dockerfile_roar_web_image ."

}
```

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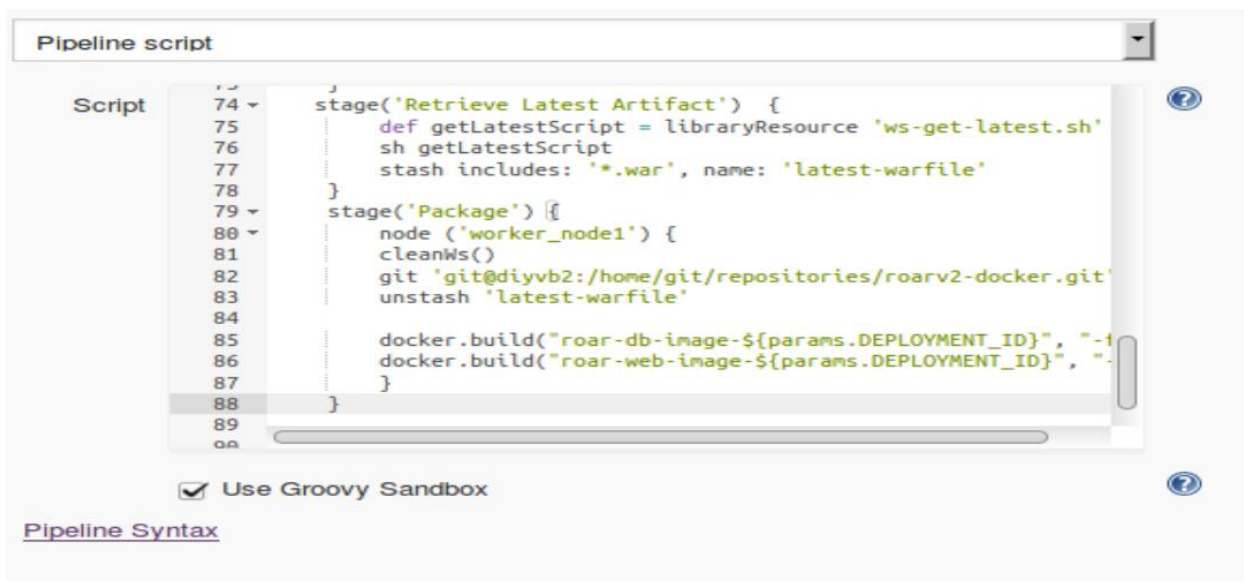

6. After this, you should be able to go to a **Terminal Emulator** window, issue the command "docker images" and see the "green" images listed (those with "green" in their name).

© 2020 Brent Laster

Lab 7 – 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".
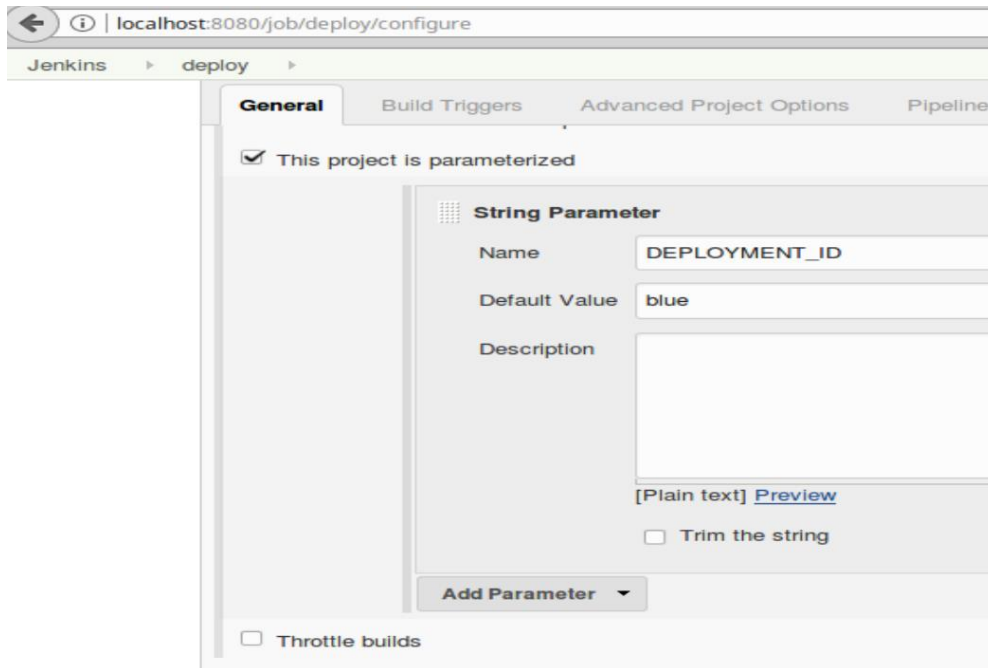
3. Further down, in the Pipeline section, we have the actual deployment code. Since our instances are packaged as Docker containers, this code will do several things when we deploy:
    a. Stop any other running instances of our application.
    b. Remove any stopped containers.
    c. Get references to the two images we need for our application – selected based on the deployment id that is passed in as a parameter.
    d. Start the images linked together into a single app.
    e. Find and print the ip address where the newly deployed image is running.
4. 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}")
```

© 2020 Brent Laster

```
    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)"
  }
}
```

5. Let's deploy an instance.   Select "**Build with Parameters**". Change the parameter value to "green". This will deploy our last built set of images.
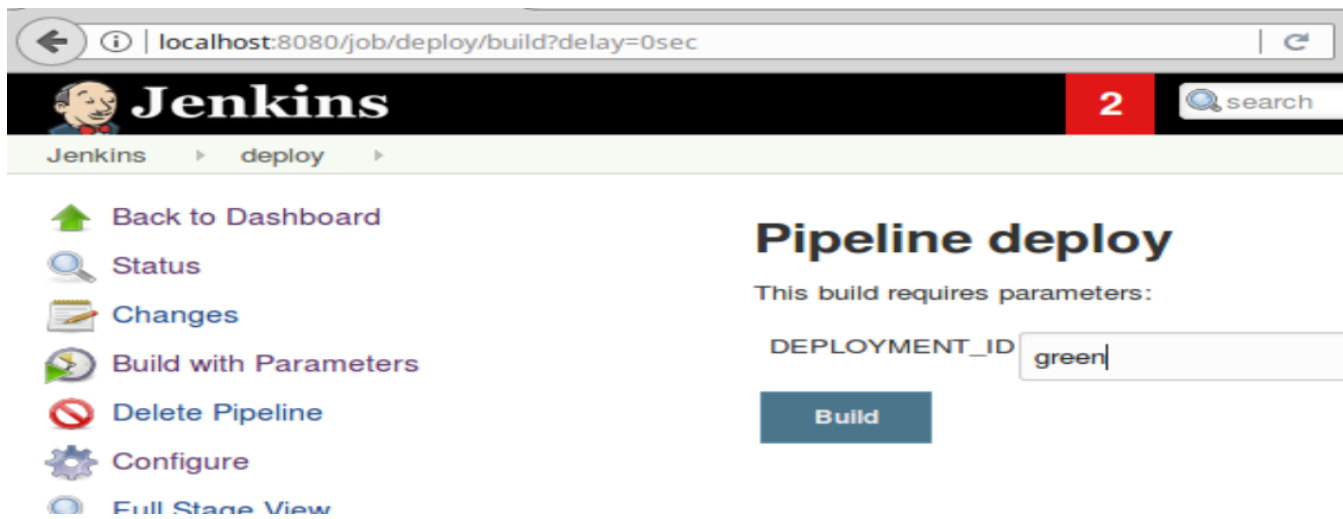
6. After the build runs, click on the Console Output link or the blue ball next to the build to open the Console Output screen.

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

© 2020 Brent Laster

Browser window showing:

```
Jenkins    ▸    deploy    ▸    #1
[Pipeline] sh
[deploy] Running shell script
+ docker run -d -p 3308:3306 -e MYSQL_DATABASE=registry -e MY
MYSQL_PASSWORD=admin roar-db-image-blue
[Pipeline] dockerFingerprintRun
[Pipeline] sh
[deploy] Running shell script
+ docker run -d --link a2d1e49bf39602a661890c2978e6707bea0542
roar-web-image-blue
[Pipeline] dockerFingerprintRun
[Pipeline] sh
[deploy] Running shell script
+ docker ps -q -l
+ docker inspect --format {{.Name}} is available at http://{{
71998a458bd2
/reverent_banach is available at http://172.17.0.3:8080/roar
[Pipeline] }
[Pipeline] // stage
[Pipeline] ]
```

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

R.O.A.R (Registry of Animal Responders) Agents

Show 10 ∨ entries   Search: _____

Id ▲	Name	Species	Date of First Service	Date of Last Service	Adversary	Adversary Tech
1	Road Runner	bird	1955-01-20	1995-02-15	Wile E. Coyote	ACME product du jour
2	Scooby	dog	1969-05-19	2000-02-11	fake ghosts	mask
3	Perry	platypus	2013-01-20	2015-04-09	H. Doofensmirtz	…inator
4	Mr. Krabs	crab	2010-06-17	2014-07-07	Plankton	various
5	Bugs Bunny	rabbit	1966-05-22	1988-04-15	E. Fudd	wabbit gun

Showing 1 to 5 of 5 entries   Previous 1 Next

9. Go back to Jenkins.

Lab 8 – 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

2. Edit the agents html file.

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

3. In the editor, update the title to be something different, such as adding your name (in the line that starts with "**h1**").
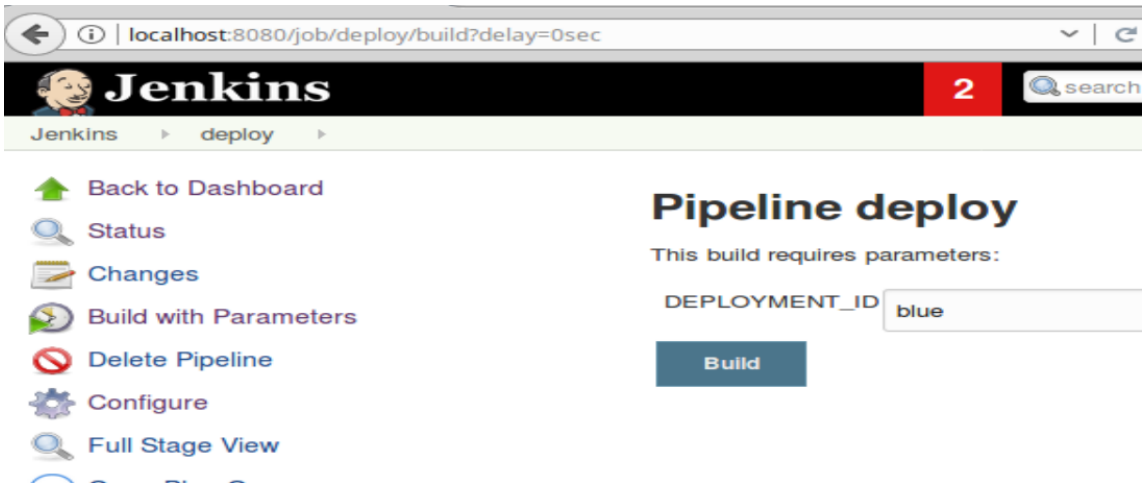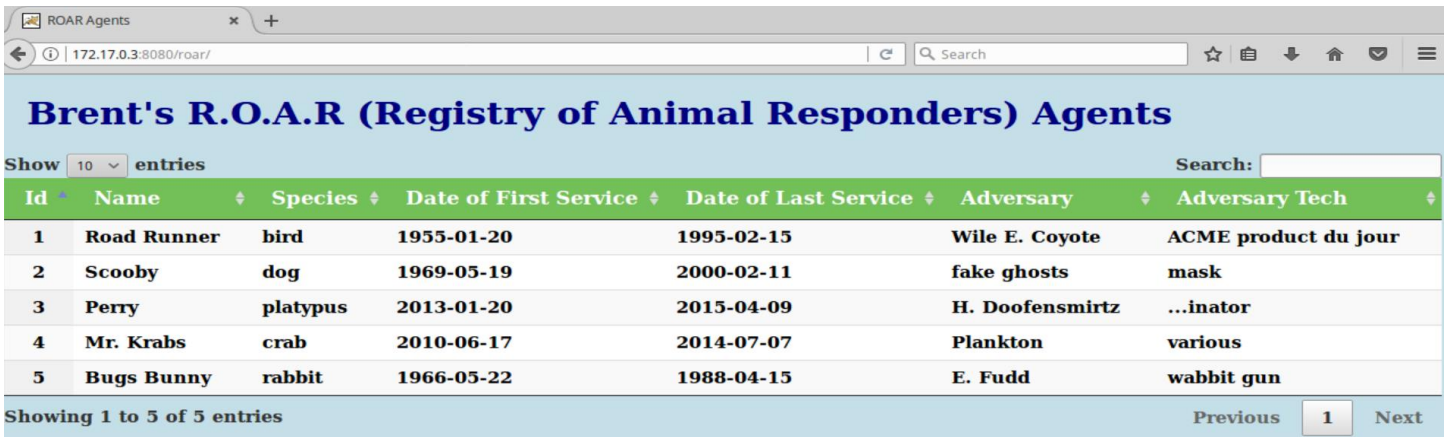
```
Open  ▼        +                          agents.html                              Save
                                ~/workshop/web/src/main/webapp

File   Edit   View   Search   Tools   Documents   Help
18 </head>
19 <body>
20
21 <h1>Brent's| R.O.A.R (Registry of Animal Responders) Agents</h1>
22
23
24
25 <table id="get_registry2" class="display" col align="left" width="100%">
26 <thead>
27     <tr>
28     <th>Id</th>
29     <th>Name</th>
```
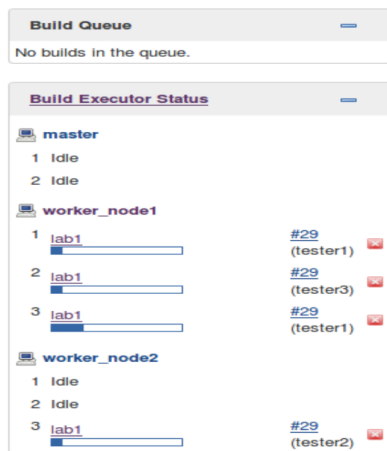
4. **Save** your changes and exit the editor.

5. Stage, commit, and push your update.

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

    git commit -m "Update"

    git push

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

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


```
diyuser2@diyvb2:~/roarv2-docker$ docker images | grep blue
roar-web-image-blue      latest         7e284f2dfe0e      About a minu
 366MB
```

8. Now, let's deploy the "blue" version with our new change. Run the Jenkins "deploy" job and enter (or accept) the default "blue" value.

© 2020 Brent Laster

Brent Laster

9. 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.)