

Getting Started with Continuous Delivery

Revision 3.2 – 8/19/21

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>)

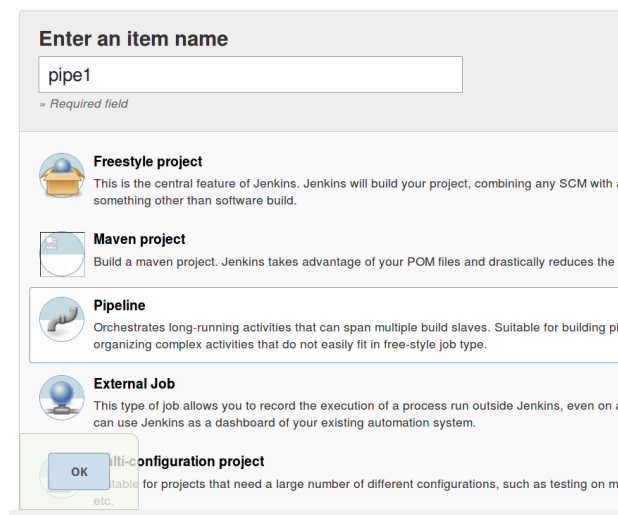
2. If you run into problems, double-check your typing!

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**.



Enter an item name

pipe1

Required field

Freestyle project
This is the central feature of Jenkins. Jenkins will build your project, combining any SCM with an something other than software build.

Maven project
Build a maven project. Jenkins takes advantage of your POM files and drastically reduces the cc

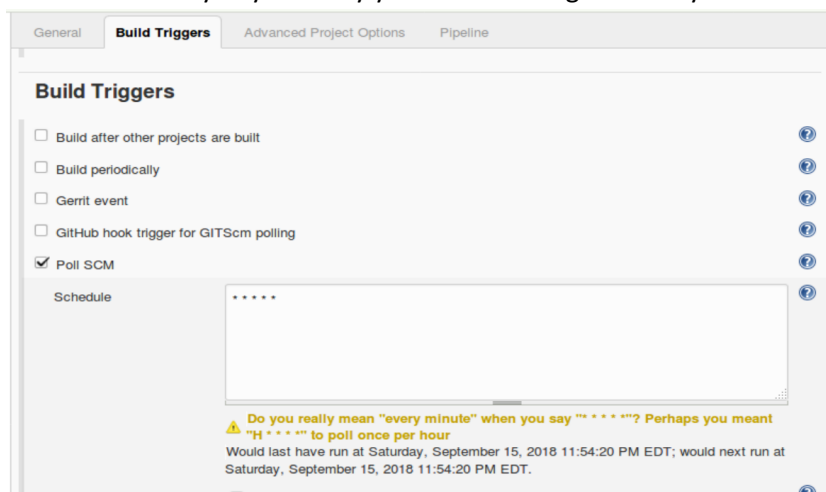
Pipeline
Orchestrates long-running activities that can span multiple build slaves. Suitable for building pipe organizing complex activities that do not easily fit in free-style job type.

External Job
This type of job allows you to record the execution of a process run outside Jenkins, even on a r can use Jenkins as a dashboard of your existing automation system.

Multi-configuration project
Suitable for projects that need a large number of different configurations, such as testing on mul etc.

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.



General **Build Triggers** Advanced Project Options Pipeline

Build Triggers

☐ Build after other projects are built

☐ Build periodically

☐ Gerrit event

☐ GitHub hook trigger for GITScm polling

☒ Poll SCM

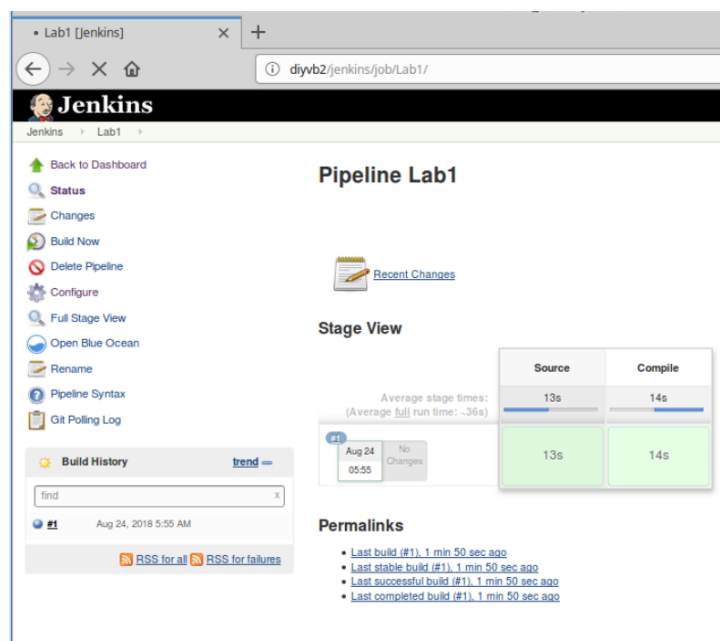
Schedule

* * * * *

Do you really mean "every minute" when you say "* * * * *"? Perhaps you meant "H * * * * *" to poll once per hour

Would last have run at Saturday, September 15, 2018 11:54:20 PM EDT; would next run at Saturday, September 15, 2018 11:54:20 PM EDT.

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. (You can just "Select All" and then "Copy" and then "Paste" to get the text from the file into Jenkins.)
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. 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.)



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.

```
$ 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
```

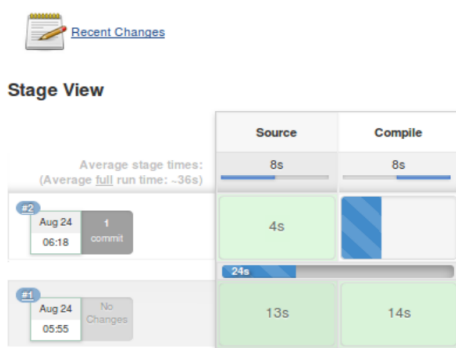
```
$ git commit -m "Add readme file"
```

```
$ 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 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.

© 2021 Tech Skills Transformations, LLC & Brent Laster

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.

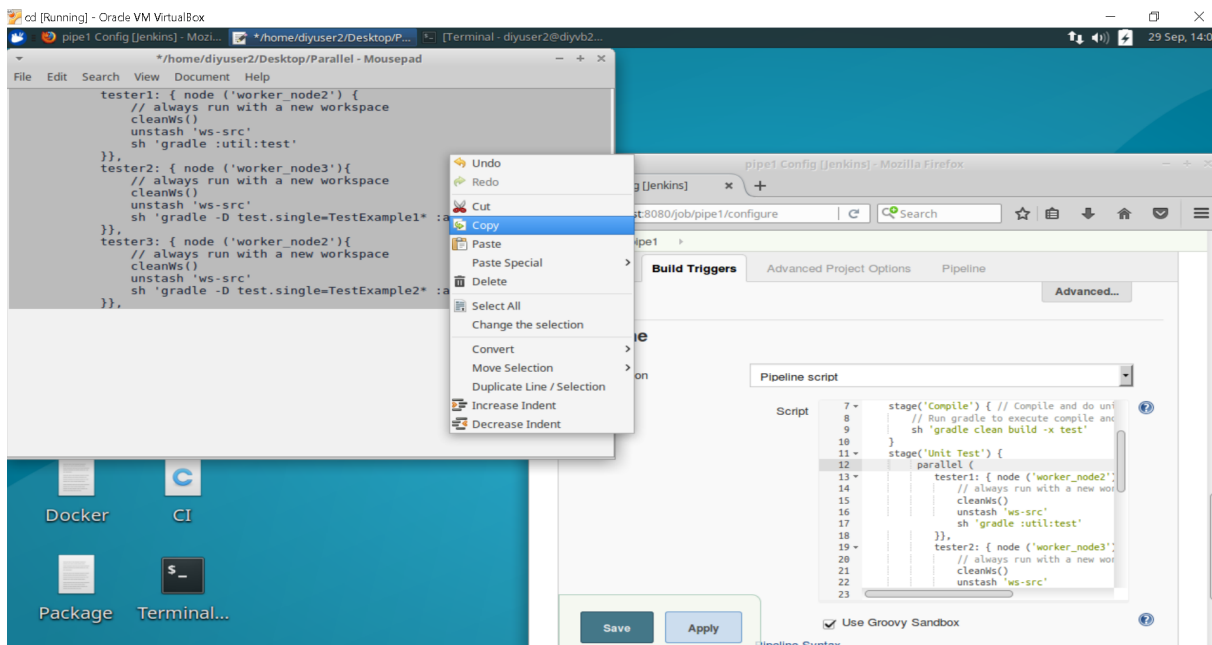
```
stage('Compile') { // Compile and do unit testing
    // Run gradle to execute compile and unit testing
    sh 'gradle clean compileJava -x test'
}
stage('Unit Test') {

    parallel (

)

}
```

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.



Pipeline script

Script

```

10      }
11      stage('Unit Test') {
12          parallel (
13              tester1: { node ('worker_node2') {
14                  // always run with a new workspace
15                  cleanWs()
16                  unstash 'ws-src'
17                  sh 'gradle :util:test'
18              }},
19              tester2: { node ('worker_node3'){
20                  // always run with a new workspace
21                  cleanWs()
22                  unstash 'ws-src'
23                  sh 'gradle -D test.single=TestExample1* :api:test'
24              }},
25              tester3: { node ('worker_node2'){
26                  // always run with a new workspace

```

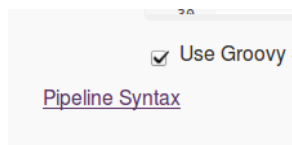
☒ Use Groovy Sandbox

[Pipeline Syntax](#)

- 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.
- 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.



- 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.

Steps

Sample Step `stash: Stash some files to be used later in the build`

Name

Includes

Excludes

Use Default Ant Excludes ☒

Allow empty stash ☐

Generate Pipeline Script

`stash includes: 'api/**, dataaccess/**, util/**, build.gradle, settings.gradle', name: 'ws-src'`

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      }
18    }
19  }
```

☒ Use Groovy Sandbox

[Pipeline Syntax](#)

7. **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.

	Source	Compile	Unit Test
Average stage times: (Average <u>full</u> run time: ~21s)	760ms	8s	25s
#9 Aug 13 20:36 No Changes	380ms	8s	26s

8. 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 assembles 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'
}
```


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

Stage View

Average stage times: (Average <u>full</u> run time: ~27s)				
	Source	Compile	Unit Test	Assemble
	724ms	8s	26s	5s
#10 Aug 13 20:55 No Changes	401ms	8s	32s	5s

3. 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.

```

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$

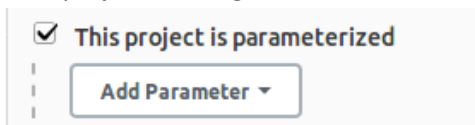
```

Lab 4 – Packaging

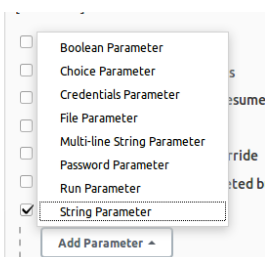
Purpose: In this lab, we'll see how to package up our application with a deployment identifier – ready to deploy.

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.

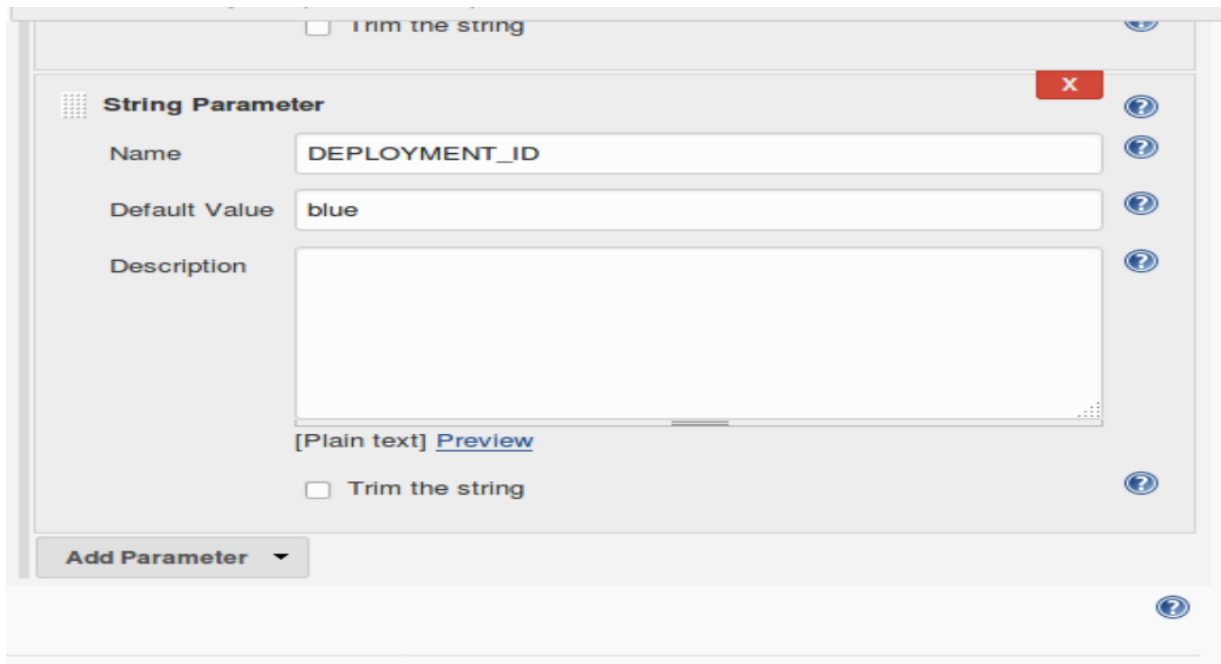
In the project's configuration, in the **General** section, click on the item that says "This project is parameterized"



Click on the "Add Parameter" button and, in the list that pops up, select **String Parameter**.



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



- 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()

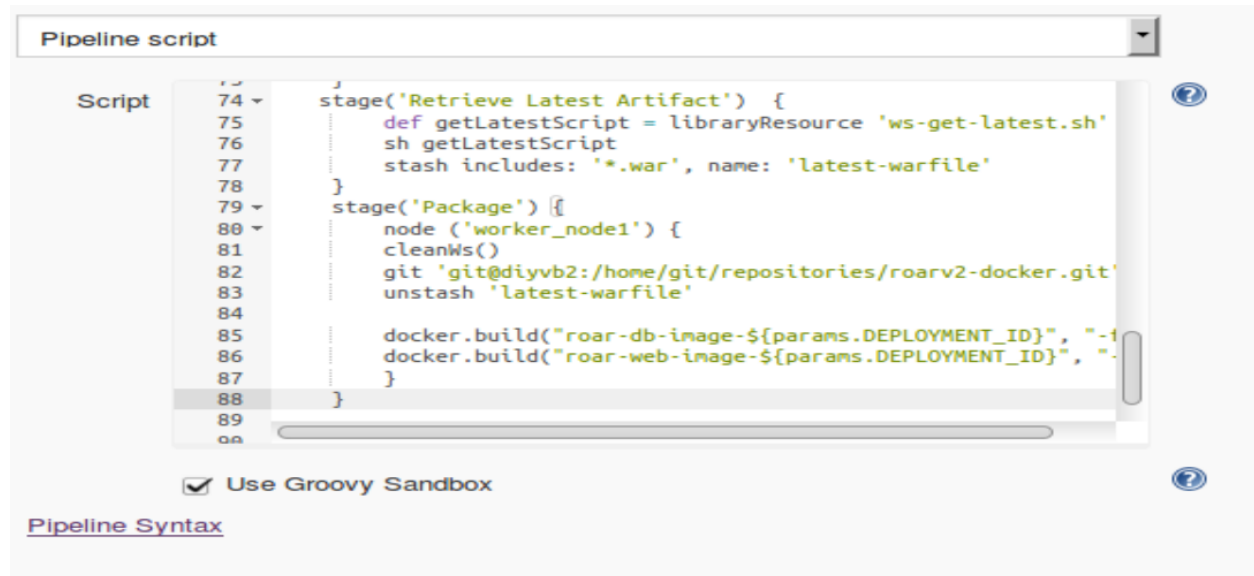
    // download the Dockerfiles
    git 'git@diyvb2:/home/git/repositories/roarv2-docker.git'

    // 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 ."
  }
}
```

3. Now, your pipeline script should look something like this. **Save** your changes.

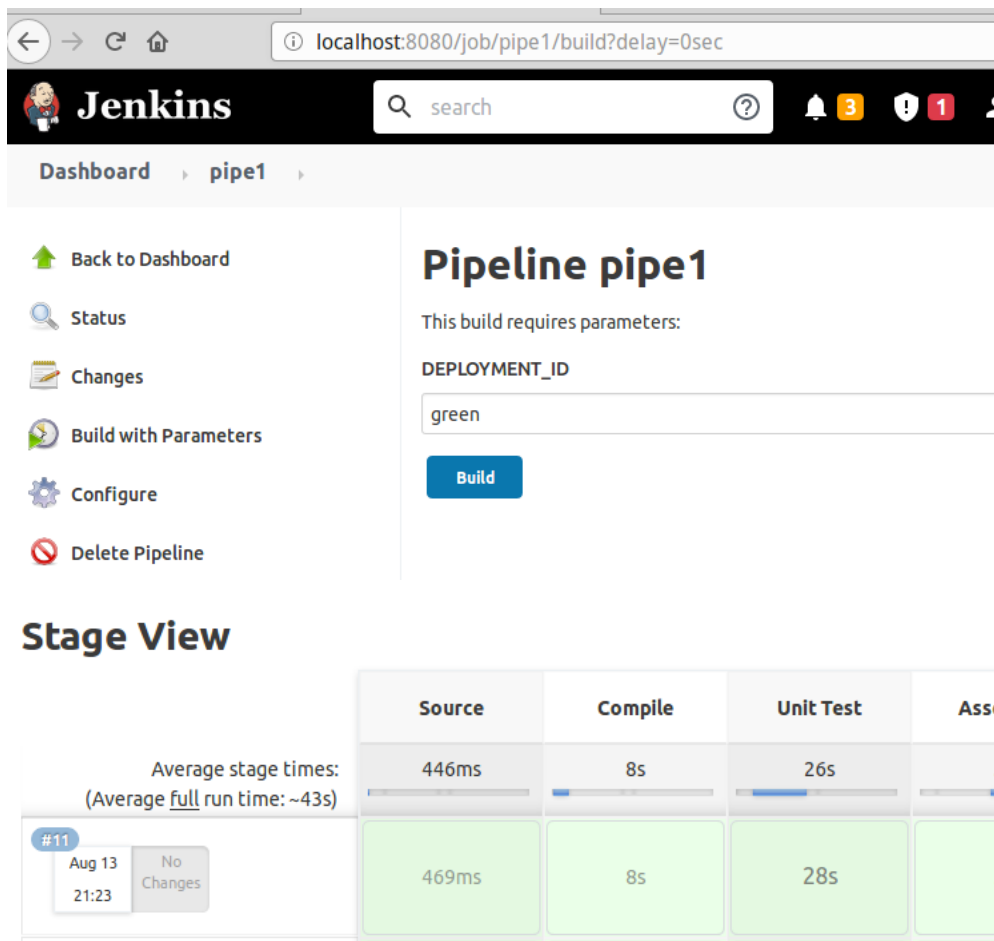


```
Script
74 stage('Retrieve Latest Artifact') {
75     def getLatestScript = libraryResource 'ws-get-latest.sh'
76     sh getLatestScript
77     stash includes: '*.war', name: 'latest-warfile'
78 }
79 stage('Package') {
80     node ('worker_node1') {
81         cleanWs()
82         git 'git@diyvb2:/home/git/repositories/roarv2-docker.git'
83         unstash 'latest-warfile'
84
85         docker.build("roar-db-image-${params.DEPLOYMENT_ID}", "-f Dockerfile.db")
86         docker.build("roar-web-image-${params.DEPLOYMENT_ID}", "-f Dockerfile.web")
87     }
88 }
89
90
```

☒ Use Groovy Sandbox

[Pipeline Syntax](#)

4. 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”.**



localhost:8080/job/pipe1/build?delay=0sec

Jenkins

Dashboard > pipe1

- Back to Dashboard
- Status
- Changes
- Build with Parameters
- Configure
- Delete Pipeline

Pipeline pipe1

This build requires parameters:

DEPLOYMENT_ID

green

Build

Stage View

	Source	Compile	Unit Test	Assemble	Package
Average stage times: (Average full run time: ~43s)	446ms	8s	26s	5s	38s
#11 Aug 13 21:23 No Changes	469ms	8s	28s	5s	38s

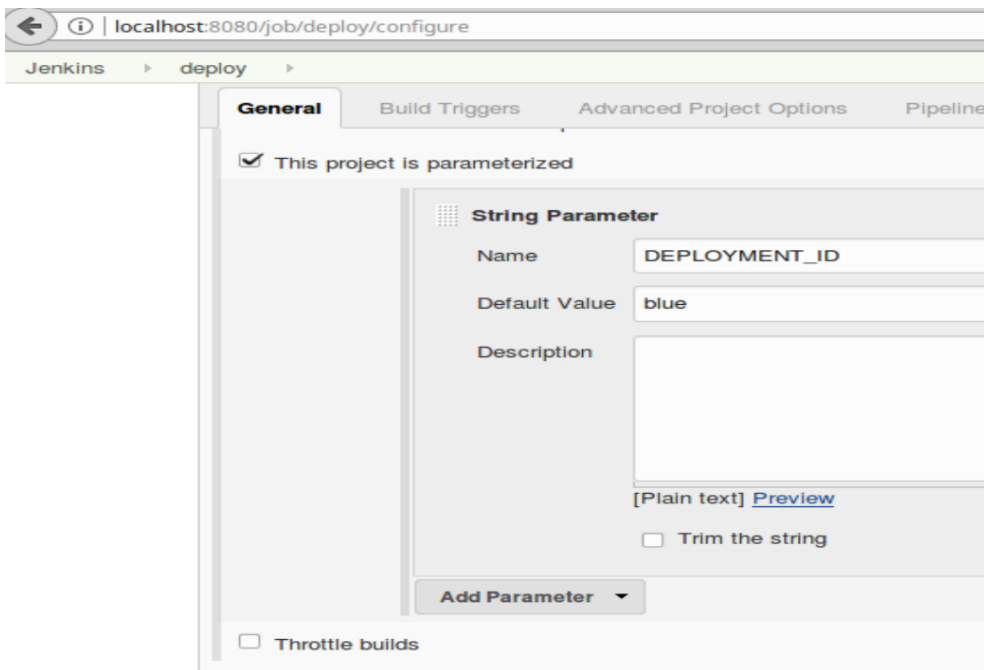
- After this, you should be able to go to a **Terminal Emulator** window, issue the command “docker images” and see the “green” images listed (those with “green” in their name).

```
diyuser2@diyvb2:~/roarv2-docker$ docker images | grep green
roar-web-image-green      latest          31d5a5668b6a
366MB
roar-db-image-green       latest          2082dcf6d7fe
213MB
```

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.

- 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.
- 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”.

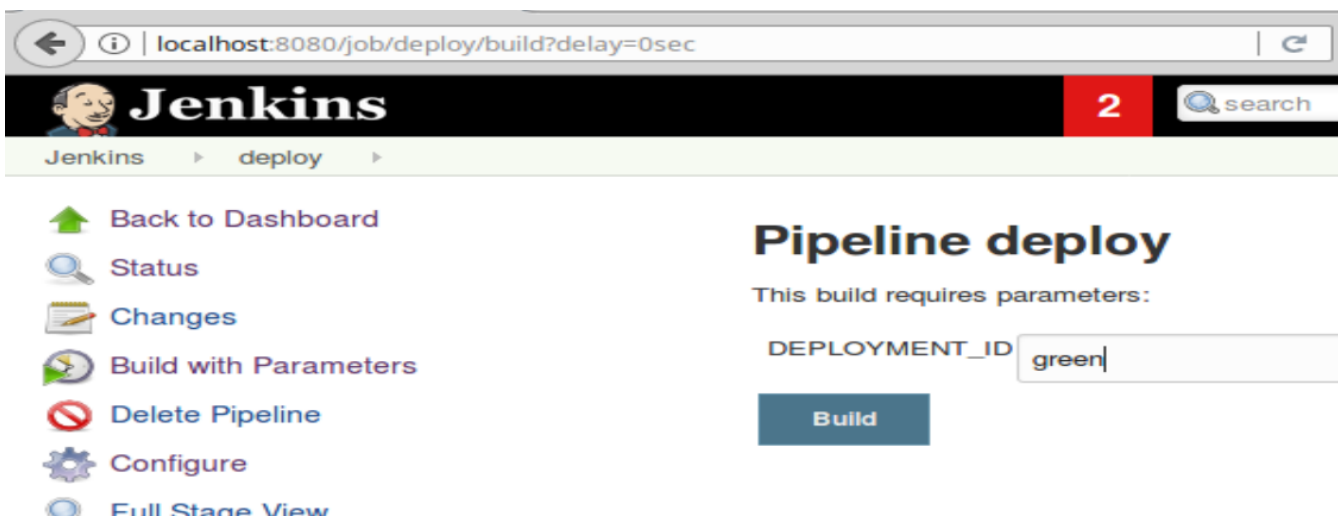


- Further down, in the Pipeline section, we have the actual deployment code. Since our instances are packaged as Docker containers, this code will do several things when we deploy:
 - Stop any other running instances of our application.
 - Remove any stopped containers.
 - Get references to the two images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

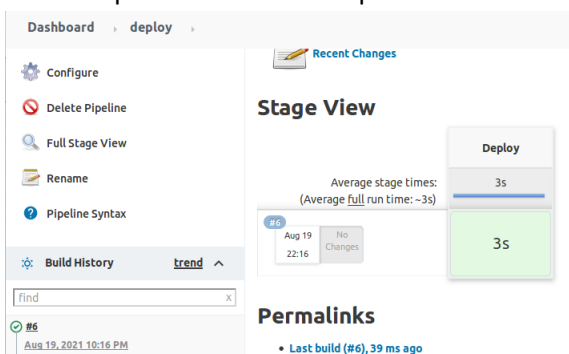
```
node('worker_node1') {
  stage('Deploy') {
    sh "docker stop `docker ps -a --format '{{.Names}}' \n\n` || true"
    sh "docker rm -f `docker ps -a --format '{{.Names}}' \n\n` || true"
    dbImage = docker.image("roar-db-image-${params.DEPLOYMENT_ID}")
    webImage = docker.image("roar-web-image-${params.DEPLOYMENT_ID}")
    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)`"
  }
}
```

- Let's deploy an instance. Go back out of the config screen. Select **"Build with Parameters"**. Change the parameter value to **"green"**. Click on the **"Build"** button. This will deploy our last built set of images.



- After the build runs, click on the Console Output link or the green circle with the checkmark next to the build to open the Console Output screen.



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

```

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://ff
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:

about:sessionrestore x ROAR Agents x +

172.17.0.3:8080/roar/

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 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

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").

```

agents.html
~/workshop/web/src/main/webapp
Save
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.

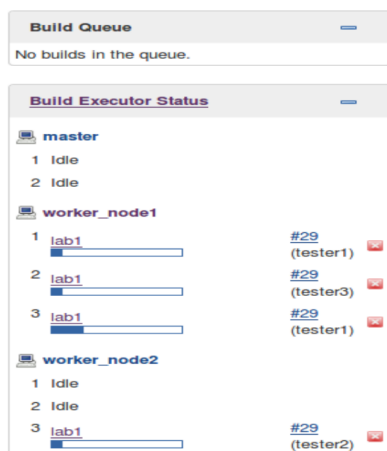
5. Stage, commit, and push your update.

```
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.



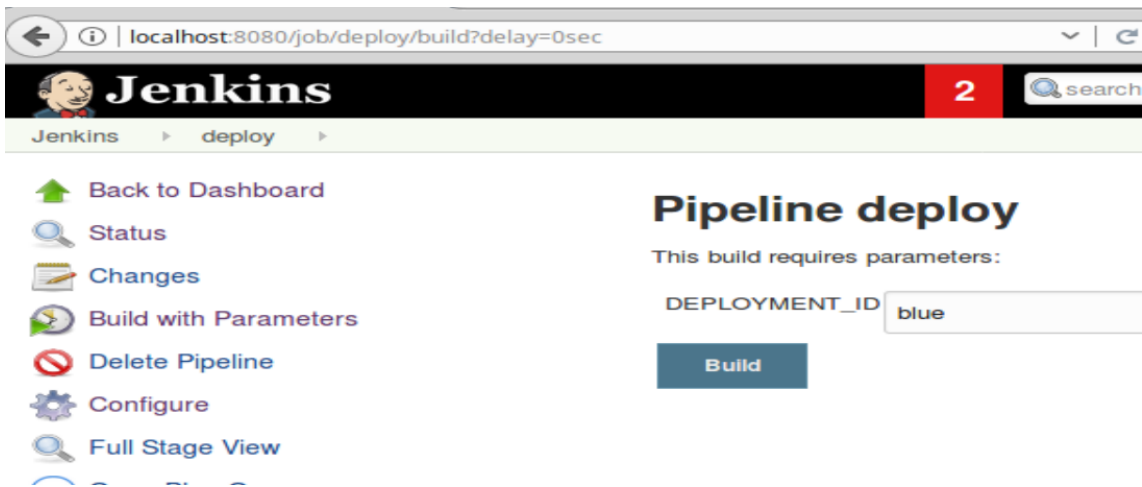
7. When this completes, we will have a new web docker image (with a deployment type of “blue” since that was our default).

```

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.



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.)

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

To clear the cache, you can go to the **Preferences** menu, then to **Privacy & Security**, select **Cookies and Site Data**, then **Clear Data** and refresh the browser.

