

# Workshop: Creating a Deployment Pipeline with Jenkins 2

## DevOps World/Jenkins World 2018

### Workshop Labs

Version 1.0 – 8/26/18

Brent Laster

#### Setup

In this workshop, we will be creating a pipeline on a VM that you can run on your system using Virtualbox. The applications that the pipeline uses (including Jenkins) are installed in the VM and all work will be done in the VM.

**IMPORTANT! Prior to beginning the labs, please make sure you have followed the steps to get the VM up and running from the setup document at:**

<https://github.com/brentlaster/conf/blob/master/dwjwt2018/dw-jdp-setup.pdf>

The labs are typically divided into two parts – one part to update in the Jenkinsfile for the pipeline definition, and one part to update in the Jenkins application to be able to run the pipeline.

On the VM, the editing of the Jenkinsfile can be done in a terminal session via an editor (gedit, nano, etc).

Within the Jenkinsfile, steps to complete will be marked as numbered comments – of the form

`// * # instructions`

In each branch a completed Jenkinsfile can be found under Jenkinsfile-lab#.solution.

**Note: Before doing the labs, make sure to have the ova file loaded and the image up and running in VirtualBox on your system. All steps in the labs are intended to be done in the VM (DW-JDP image).**

#### Lab 1 – Adding a stage in the Jenkinsfile

**Purpose:** In this first lab, we'll see what the basic structure of a scripted pipeline looks like, how to reference and invoke a Gradle installation in a pipeline, how to update a Jenkinsfile, and how to define a multibranch job in Jenkins to reference a Jenkinsfile.

In a terminal window/editor:

1. Open a terminal session. (You can use the **Terminal Emulator icon** on the desktop). Change into the pipeline directory and checkout the lab 1 branch:

```
cd pipeline
```

```
git checkout lab1
```

2. Rename the file “Jenkinsfile.start” to just “Jenkinsfile” using Git.

```
git mv Jenkinsfile.start Jenkinsfile
```

3. Using whatever editor you want, open the Jenkinsfile for editing. (Note gedit and nano editors are available on the VM.)

```
gedit Jenkinsfile
```

4. Note that we already have a **Source** stage here that checks out our source. We are going to add a **Build** stage. Add the 3 lines **in bold** below in the file under “// \* 1. Add build stage ...”. (Note that the set of characters after the **sh** command is double quote, single quote, dollar sign, left curly brace.)

```
node ('worker_node1') {  
    stage ('Pull Source') {  
        checkout scm  
    }  
// * 1. Add build stage ...  
    stage('Compile') {  
        sh "${tool 'gradle4'}/bin/gradle' -g /home/jenkins2/.gradle clean  
        compileJava -x test"  
    }  
}
```

Note: The “-g /home/jenkins/.gradle” is optional, but recommended. This tells Gradle to use this directory as the home directory. We do that because that directory already contains a cache of dependencies we need and this will cut down on the downloads.

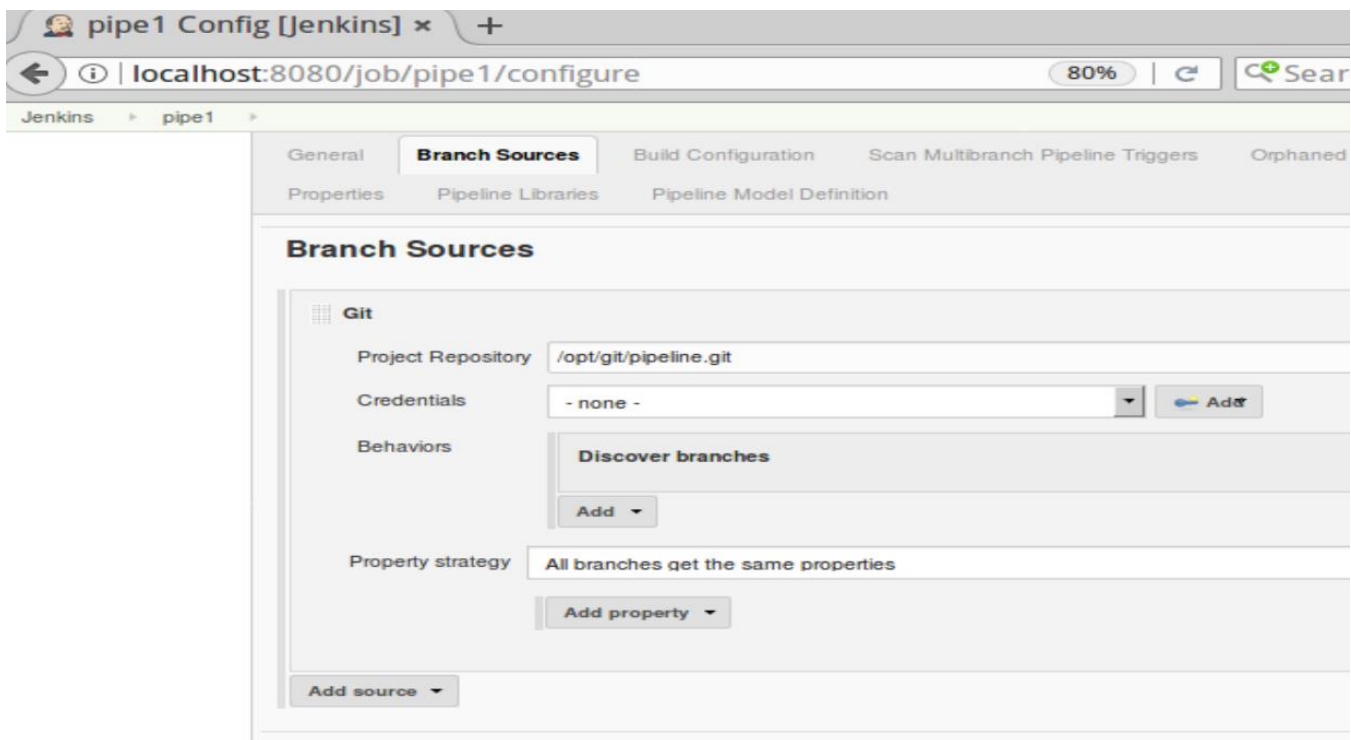
5. Save your changes and Quit the editor. Then, still in the terminal window, update the Jenkinsfile in the remote Git repository. (You can use whatever commit message/comment you want instead of “Update for lab 1” if you prefer.) Note that there is no space in the “lab1” part for the push command.

```
git commit -am “Update for lab 1”
```

```
git push origin lab1
```

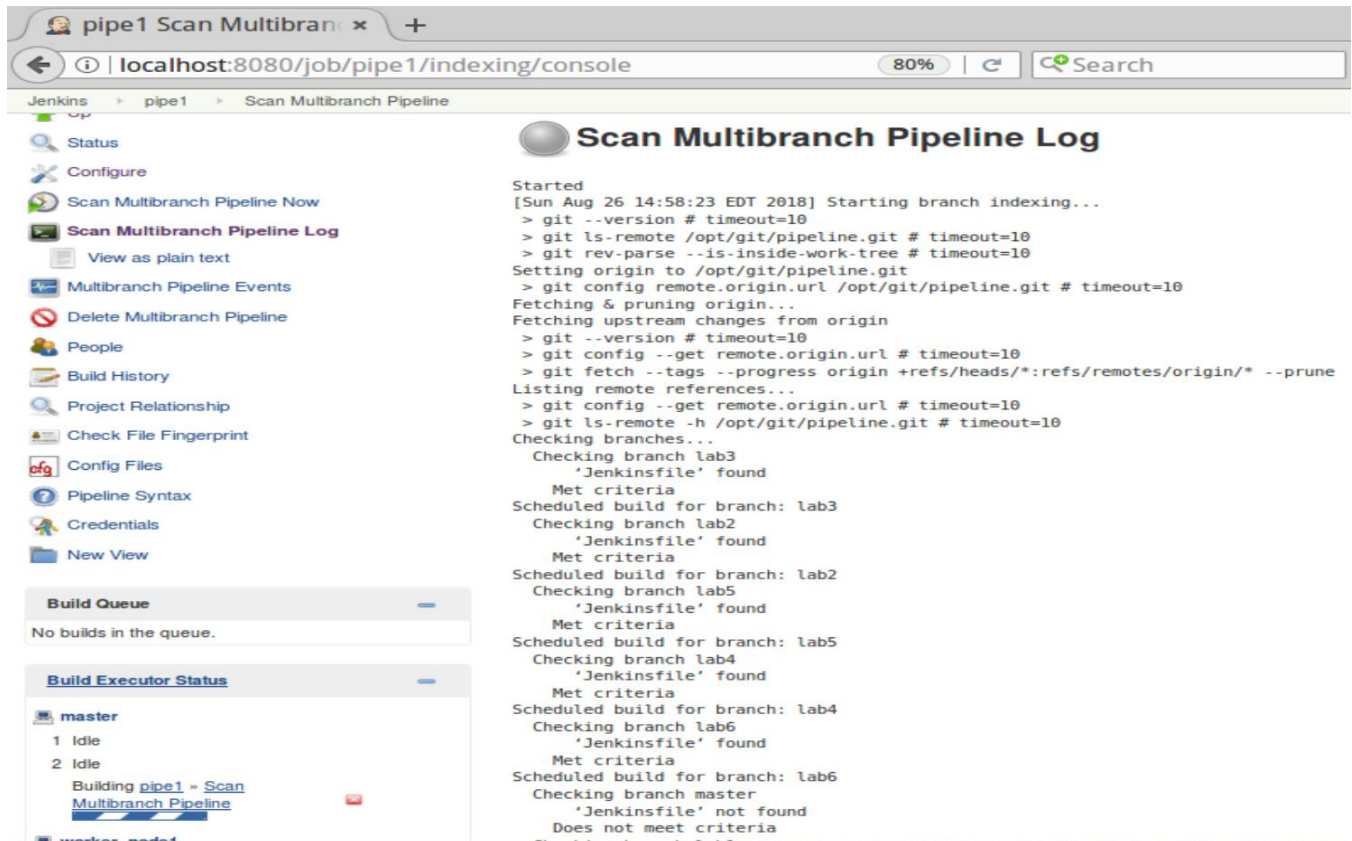
**In the Jenkins application:**

6. Start Jenkins by clicking on the “**Jenkins 2**” shortcut on the VM desktop OR opening the Firefox browser and navigating to “**http://localhost:8080**”.
7. Log in to Jenkins with the username and password provided in the class.
8. Now we’ll create a project to use our Jenkinsfile and build the pipeline. On the Jenkins dashboard, select the “**New Item**” menu item on the left-hand side.
9. Enter a name for your project such as “**pipe1**” (this can be whatever you want, just refer to it the same way throughout the labs).
10. For the type of the item, select **Multibranch Pipeline** and then click on the **OK** button. (You may need to scroll down to find the project type.) This will put you into the **configure** page for this project – eventually.
11. Scroll down to the **Branch Sources** area. Select **Add source** and then **Git**. Fill in the field with the location of the remote git repository on the system: **/opt/git/pipeline.git**. You can leave the rest of the fields as they are.



12. **Save** your changes.

After a moment, Jenkins should detect the various branches and Jenkinsfiles and automatically create jobs (for the one in this case).



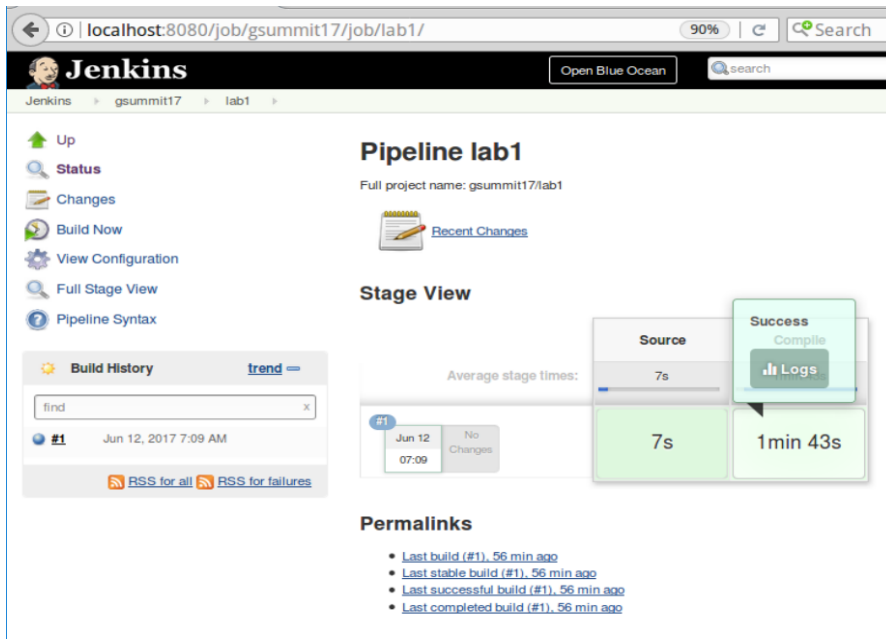
13. You can just let this run while we continue.

## Lab 2 – Incorporating a shared library

In this lab, we will introduce a shared library into our pipeline that encompasses the build functionality.

In Jenkins:

1. Take a quick look at the **Staging View** output for the lab 1 job. If you are on the “**Scan Multibranch Pipeline Log**” page, click on the “**Up**” link in the top left part of the page. You’ll be back on the dashboard (<http://localhost:8080>).
2. Now click on the multibranch pipeline project name (such as pipe1) in the list of projects. Then click on **lab1** in the list of branches.
3. Now you’ll be on the **Stage View** page for lab1’s build. You can hover over the green boxes and click on the links in the pop-up window to see the output and logs from this view.



In a terminal window/editor:

- Now we'll replace our build command with a shared library call. The shared library code is already on the VM in the `diyuser2` area. To see it, switch to a terminal window and look in the **shared-libraries** subdirectory. Notice the structure. The routine we will be calling is in the **vars/gbuild4.groovy** file.

```
cat ~/shared-libraries/vars/gbuild4.groovy
```

To update in the Jenkinsfile:

- If not in the `~/pipeline` directory, `cd` back to it. Switch to the **lab2** branch, rename and edit the Jenkinsfile.

```
git mv Jenkinsfile.start Jenkinsfile
```

```
git checkout lab2
```

```
gedit Jenkinsfile
```

- Follow step 1 in the file and add a line (in **bold** below) near the top of the script (before the node definition) to bring in our shared library: (Note that the `_` is immediately after the right parenthesis and is important.)

```
@Library('Utilities2')_
node ('worker_node1') {
```

- Replace** the line in the **Compile** stage (the one with the shell (**sh**) call) with the following line to call our shared library routine.

```
gbuild4 ' -g /home/jenkins2/.gradle clean compileJava -x test'
```

8. **Save** your changes, **Quit** the editor and update the Jenkinsfile in your Git project.

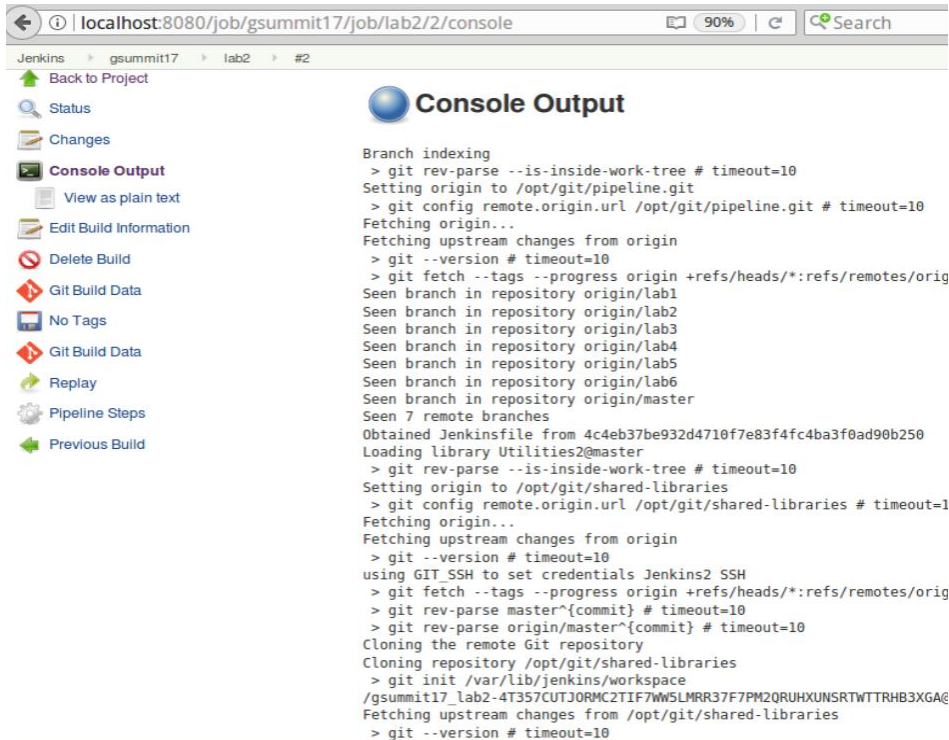
```
git commit -am "update for lab 2"  
git push origin lab2
```

#### In Jenkins:

9. Switch back to Jenkins and back/up to the dashboard. From the dashboard select **Manage Jenkins** and then **Configure System**. Scroll down to find the **"Global Pipeline Libraries"** section. Nothing needs changing here, but look at how it is configured.
10. Go back into your **pipe1** project (via the dashboard or at localhost:8080/job/pipe1). Click on **ENABLE AUTO REFRESH** in the top right corner if not already enabled.
11. Click on the **"Scan Multibranch Pipeline Now"** link to tell it to rescan for changes. The project should detect the changes for the lab2 branch and build it now - loading the library and using it instead of the direct Gradle call.



12. To see this you can go to the console log for the project. (Direct link is <http://localhost:8080/job/pipe1/job/lab2/1/console> or go into your project, select lab2 and then select the console log. (In the Build History window, click on the blue dot next to #2.)) Look for the line **"Loading library Utilities2@master"** to mark the start of where the shared library is being brought in.



```
Branch indexing
> git rev-parse --is-inside-work-tree # timeout=10
Setting origin to /opt/git/pipeline.git
> git config remote.origin.url /opt/git/pipeline.git # timeout=10
Fetching origin...
Fetching upstream changes from origin
> git --version # timeout=10
> git fetch --tags --progress origin +refs/heads/*:refs/remotes/orig
Seen branch in repository origin/lab1
Seen branch in repository origin/lab2
Seen branch in repository origin/lab3
Seen branch in repository origin/lab4
Seen branch in repository origin/lab5
Seen branch in repository origin/lab6
Seen branch in repository origin/master
Seen 7 remote branches
Obtained Jenkinsfile from 4c4eb37be932d4710f7e83f4fc4ba3f0ad90b250
Loading library Utilities2@master
> git rev-parse --is-inside-work-tree # timeout=10
Setting origin to /opt/git/shared-libraries
> git config remote.origin.url /opt/git/shared-libraries # timeout=1
Fetching origin...
Fetching upstream changes from origin
> git --version # timeout=10
using GIT_SSH to set credentials Jenkins2 SSH
> git fetch --tags --progress origin +refs/heads/*:refs/remotes/orig
> git rev-parse master^{commit} # timeout=10
> git rev-parse origin/master^{commit} # timeout=10
Cloning the remote Git repository
Cloning repository /opt/git/shared-libraries
> git init /var/lib/jenkins/workspace
/gsummit17_lab2-4T357CUTJ0RMC2TIF7WW5LMRR37F7PM2QRUHXUNSRTWTRHB3XGA@
Fetching upstream changes from /opt/git/shared-libraries
> git --version # timeout=10
```

### Lab 3 – Leveraging parallel functionality across multiple nodes

In this lab we will see how to use the parallel function to run Gradle unit tests in parallel across multiple nodes.

In the Jenkins application:

1. We are going to create a step to make a stash of files that we can share across the parallel nodes. First, go to the **pipeline syntax snippet generator**. Do this by first going back to your lab2 or pipe1 project page. Then, click on the “**Pipeline Syntax**” link at the bottom of the left-hand menu in Jenkins.
2. From the **Steps** drop-down, select “**stash**”.

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

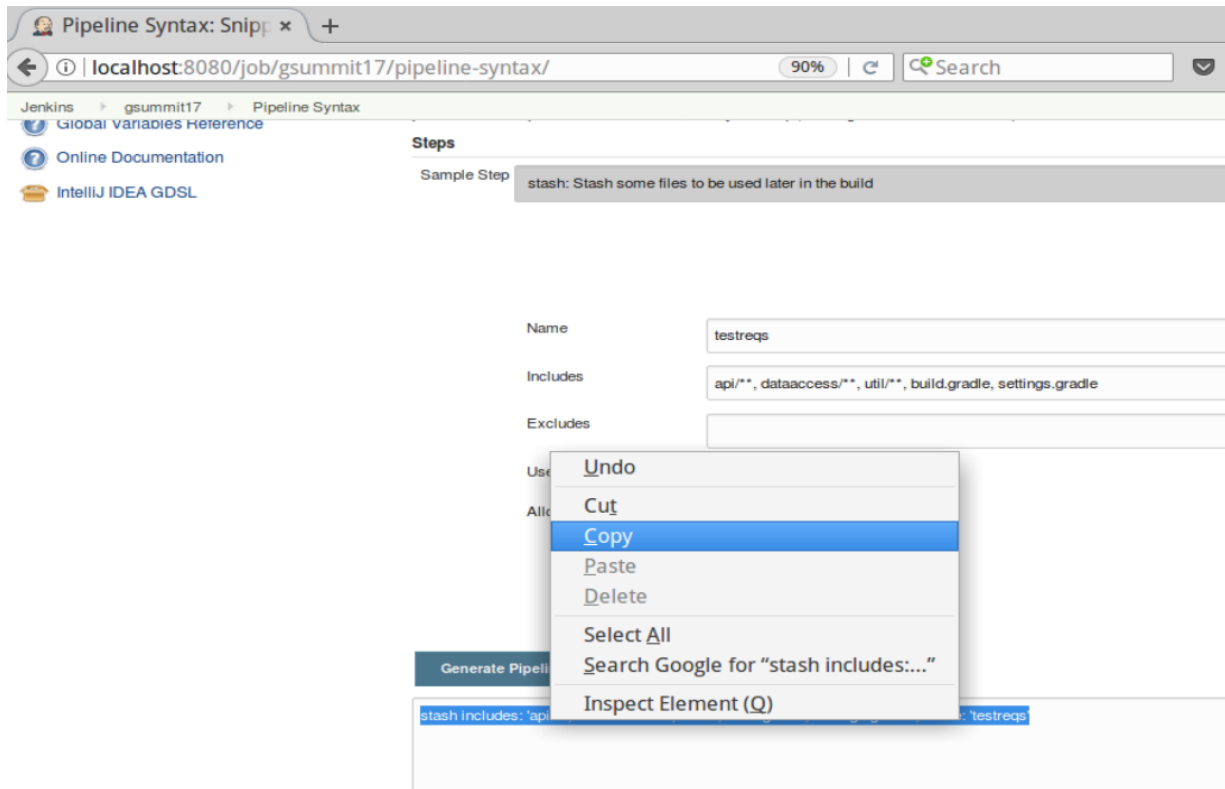
**Name:** testreqs

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

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

You can leave the rest of the fields as they are.

3. Click on the “**Generate Pipeline Script**” button and then **highlight and Copy (to the clipboard)** the generated command from the window.



In the terminal window/editor:

4. In the **Terminal Emulator** session where you have been working, switch to the branch for **lab 3**, rename the Jenkinsfile, and then edit it.

**git checkout lab3**

**git mv Jenkinsfile.start Jenkinsfile**

**gedit Jenkinsfile**

This branch will have the previous changes and a new **Unit Test** stage with a parallel step partially filled in.

5. We will need to create a stash so we can share files across the parallel nodes. In the **'Source'** stage, after the **'checkout scm'** line, paste the stash command that you generated from the **snippet generator** previously. The line should look like this (in bold):

**// \* 1. Add the stash step here from the Snippet Generator**

**stash includes: 'api/\*\*, dataaccess/\*\*, util/\*\*, build.gradle, settings.gradle', name: 'testreqs'**

6. Scroll down and look at the **'Unit Test'** stage. We have 2 branches for the parallel execution here – one for **worker\_node2** and one for **worker\_node3**.



- Look for the lines that start with **“Add commands here”**. For the one in the **worker\_node2** branch, add the commands to unstash and to execute **only the TestExample1\*** tests in the **api** subproject. (Note that there is a space between the \* and the **:api:test** task name on the end.)

```
unstash 'testreqs'
```

```
gbuild4 '-D test.single=TestExample1* :api:test'
```

- Add similar lines in the **worker\_node3** branch to execute only the **TestExample2\*** tests. (You can copy and paste and just change **“TestExample1\*”** to **“TestExample2\*”**.)

```
unstash 'testreqs'
```

```
gbuild4 '-D test.single=TestExample2* :api:test'
```

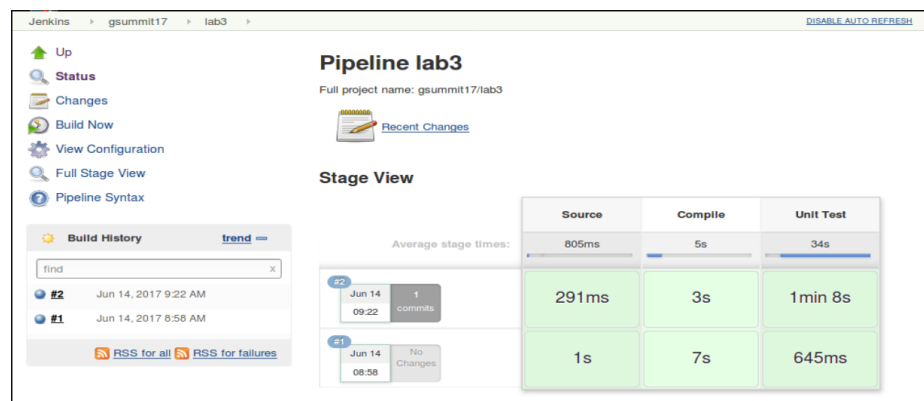
- Save** your changes and quit the editor. Update your Jenkinsfile in source control.

```
git commit -am “update for lab3”
```

```
git push origin lab3
```

#### In Jenkins:

- Switch back to Jenkins to your multibranch pipeline project (i.e. pipe1).
- Click on the **Scan Multibranch Pipeline Now** branch project to find the new project. After a moment, Jenkins should



- If you want, you can look in the Console Log for the project to see the interspersed messages from tester2 and tester3.

#### Lab 4 – Using credentials and sourcesets

In this lab, we will see how to use credentials and how to invoke the separate sourcesets.

## In Jenkins:

1. Open up your multi-branch project and go to the **Snippet Generator** (reminder - click on the **Pipeline Syntax** link). Select the “**withCredentials**” step. Then click on the “**Add**” button in the “**Bindings**” section.
2. Select “**Username and password (separated)**” from the list of options. Complete the remaining fields as follows:

Username Variable = **DBUSER**

Password Variable = **DBPASS**

Credentials = **admin/\*\*\*\*\*** (These are for a mysql-access credential already setup in Jenkins. You can look in the **Credentials** section from the Jenkins dashboard to see it if you want.)

3. Click on the **Generate Pipeline Script** button and **Copy** the generated code to the clipboard.

The screenshot shows the Jenkins Snippet Generator interface. At the top, there's a 'Steps' section with a sample step 'withCredentials: Bind credentials to variables'. Below this is the 'Bindings' section, which is currently set to 'Username and password (separated)'. It has three input fields: 'Username Variable' with the value 'DBUSER', 'Password Variable' with the value 'DBPASS', and 'Credentials' with a dropdown menu showing 'admin/\*\*\*\*\* (Username and password to access mysql)'. There are 'Add' and 'Delete' buttons next to the 'Credentials' field. At the bottom, there's a 'Generate Pipeline Script' button and a text area containing the generated pipeline script:

```
withCredentials([UsernamePassword(credentialsId: 'mysql-access', passwordVariable: 'DBPASS', usernameVariable: 'DBUSER')]) {  
    // some block  
}
```

## In the terminal window/editor:

4. Switch to terminal window, checkout the lab4 branch, rename Jenkinsfile.start and edit the Jenkinsfile.

**git checkout lab4**

**git mv Jenkinsfile.start Jenkinsfile**

**gedit Jenkinsfile**

5. In the “**Integration Test**” stage where you see the line for  
// \* 1 Insert “withCredentials” step here ...

**Paste** the command you generated from the snippet generator.

6. Move the ending bracket of the “**withCredentials**” block to be after the “**sh**” step that invokes mysql. (The mysql command should be **inside the “withCredentials” block**. If you copied the command verbatim, the sh “mysql ... “ command goes where the “// some block” text is.)
7. Now, add another call afterwards (where the // \* 2. Insert command here to run ... line is) to invoke the integration tests. You can use the library routine here again.

**gbuild4 'integrationTest'**

8. Save your changes, quit the editor, and update the Jenkinsfile in the repository.

**git commit -am “update for lab 4”**

**git push**

**In Jenkins:**

9. Go into your multibranch pipeline project and either **Scan Multibranch Pipeline Now**. The job for lab4 should run with the integration testing done successfully.

## **Lab 5 - Code Analysis**

**Purpose:** In this lab, we’ll learn how to do code analysis with SonarQube via a Jenkinsfile.

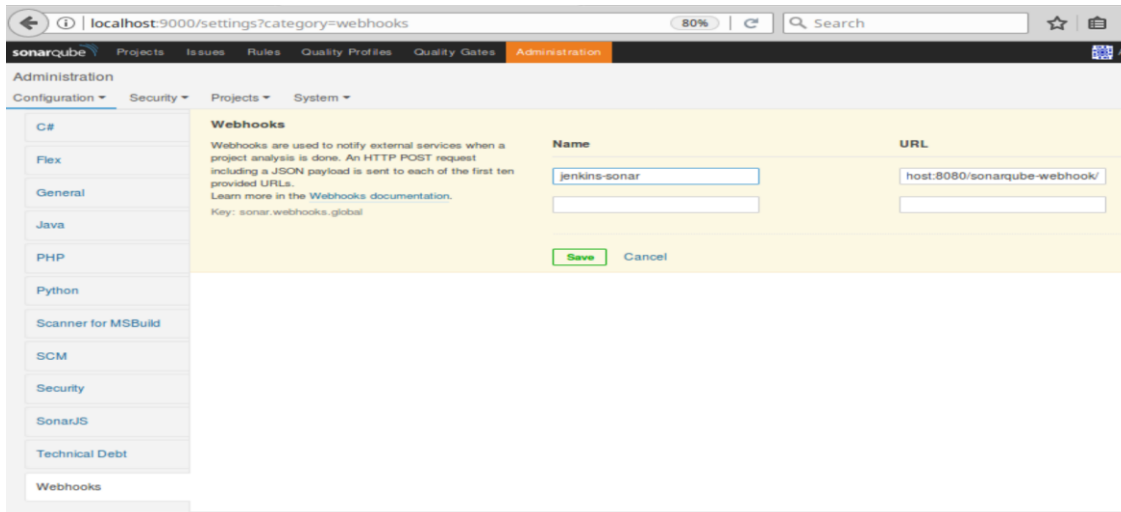
**In the browser:**

1. Open the SonarQube instance in a new tab of Firefox by going to <http://localhost:9000> and logging in with user **admin** and password **admin**.
2. Click on **Administration**, then **Configuration**, and then on **General Settings**. Then scroll down until you see the **Webhooks** section directly under that column. Click on that and fill in the fields as follows:

Name: **jenkins\_sonar**

URL: <http://localhost:8080/sonarqube-webhook/> (Note: The trailing slash on the URL is important! )

3. Once you have filled in the fields, click the **Save** button. Then the webhook is in place.



In the terminal window/editor:

4. Switch back to the terminal session. Checkout the lab 5 branch, rename, and edit the Jenkinsfile.

```
git checkout lab5
```

```
git mv Jenkinsfile.start Jenkinsfile
```

```
gedit Jenkinsfile
```

5. Scroll down in the Jenkinsfile and find the line that looks like this:  
// \* 1. Wrap the step below in a block that will run it in our local SonarQube environment

Add a **withSonarQubeEnv** closure (step with beginning and ending curly braces) to wrap the sonar-runner line.

```
withSonarQubeEnv {  
    sh "${tool 'sq-scanner'}/bin/sonar-runner' -X -e"  
}
```

6. Next, for step 2, we will add a step above "if (qg.status..." that defines the qg variable and tells Jenkins to wait for sonarqube to complete and tell us whether or not it passed the quality gate. The line to add is:

```
def qg = waitForQualityGate();
```

7. Save your changes, exit the editor, and update the Jenkinsfile.

```
git commit -am "update for lab5"
```

```
git push origin lab5
```

#### In Jenkins:

8. Switch to Jenkins in the browser. Go into your multibranch pipeline project and **Scan Multibranch Pipeline Now**. The job for lab 5 should run with the additional analysis and artifactory processing stages.

You can let this run while the class continues.

#### Lab 6 – Using Docker in the pipeline

**Purpose:** In this lab, we'll look first at the results from the Sonar and Artifactory stages in lab 5. Then we'll see how to work with Docker in a Jenkins 2 pipeline.

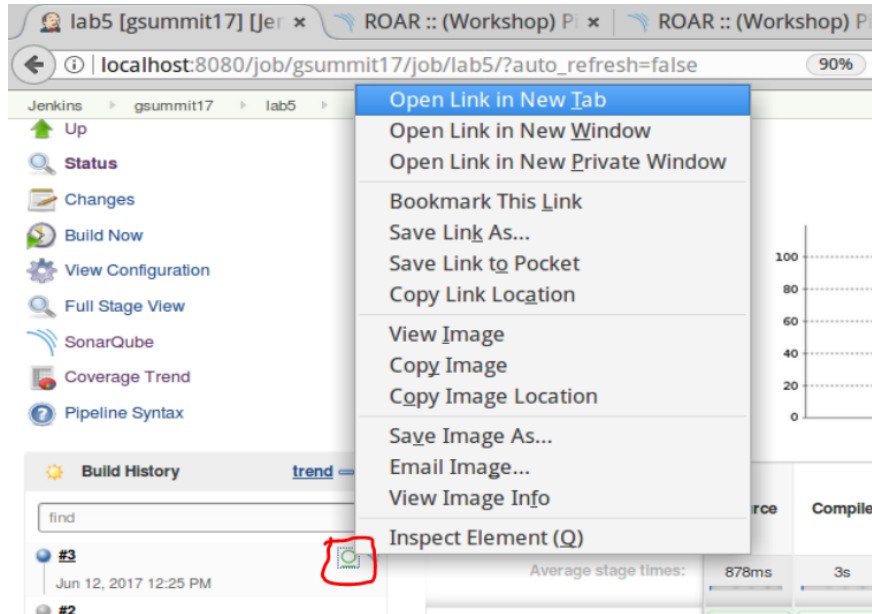
#### In Jenkins/Sonar/Artifactory:

1. On the **lab5 job** page, click on either the Sonar symbol next to the build number in the Build History window or the **OK** button under the "**SonarQube Quality Gate**" to open the SonarQube analysis of the project. (You may want to right-click and open it in another tab.) You can see the Sonar dashboard and the kind of information available here.

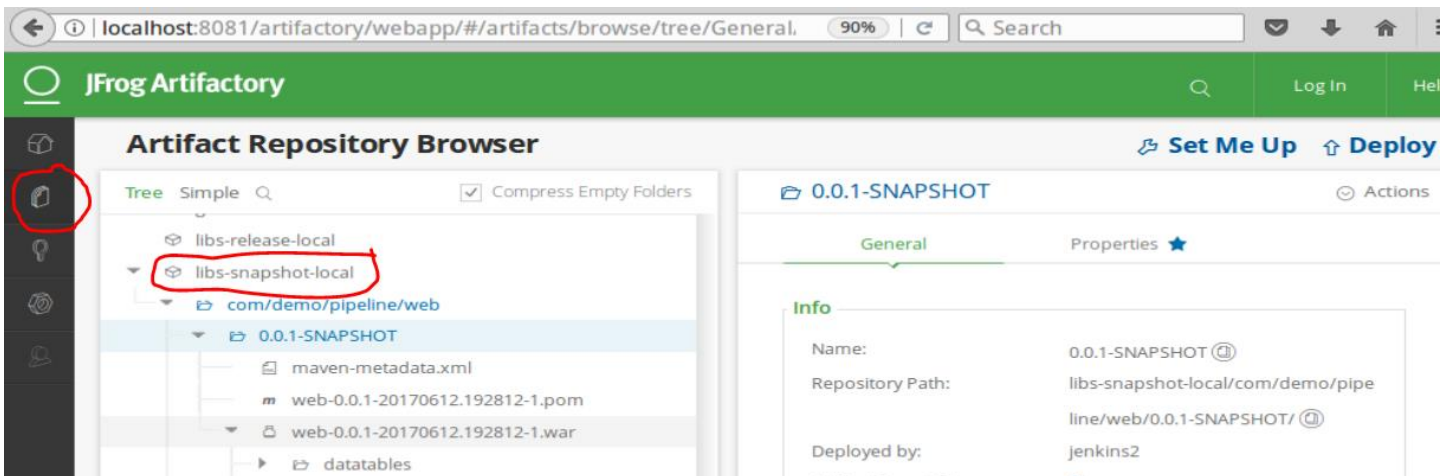
The screenshot shows the Jenkins job page for 'lab5'. On the left, the 'Build History' table lists three builds: #3 (Jun 12, 2017 12:25 PM), #2 (Jun 12, 2017 12:21 PM), and #1 (Jun 12, 2017 7:09 AM). A red circle highlights the Sonar icon next to build #3. In the center, the 'Average stage times' table shows 'Source' at 878ms and 'Compile' at 5s. Below this, the 'SonarQube Quality Gate' section shows 'ROAR :: (Workshop) Pipeline Demo' with a 'server-side processing: Success' status. A red circle highlights the 'OK' button. A right-click context menu is open over the 'OK' button, showing options like 'Open Link in New Tab', 'Open Link in New Window', 'Open Link in New Private Window', 'Bookmark This Link', 'Save Link As...', 'Save Link to Pocket', 'Copy Link Location', 'Search Google for "OK"', and 'Inspect Element (Q)'. On the right, a table shows stage times: 'assemble' at 5s and 'Publish Artifacts' at 29s. A red circle highlights the 'OK' button in the 'Publish Artifacts' column.

2. Back on the **lab5 job** page, you can also click on the Artifactory symbol (circle with line under it next to Sonar symbol) to go to Artifactory (<http://localhost:8081>). If you go directly from the

symbol, you'll see the build info. You can also browse through there and find the artifact produced by our pipeline (the war file).



3. Once in Artifactory, you can click on the “Artifacts” browser button and then drill down into the lib-snapshot-local -> com/demo/pipeline/web -> 0.0.1-SNAPSHOT area to see the artifact.



In terminal window/editor:

4. Now we'll see how to use Docker in the pipeline with Jenkins 2. Change to the terminal session, checkout the lab 6 branch, and rename and edit that Jenkinsfile.

```
git checkout lab6
```

```
git mv Jenkinsfile.start Jenkinsfile
```

## gedit Jenkinsfile

- Find the **Compile** stage. For the first change here, we will create a new **docker image with gradle** to use for building our code. We want to add a command that uses the built-in **docker variable** to build an image based off the **Dockerfile** in our **/home/diyuser2/docker/Dockerfile** location. The command below will do this. Add this command in the '**Compile**' stage. (Look for the line that starts with **// \* 1. Add a step to build a docker image...**)

```
def myImg = docker.build('gradle:4.9','-f /home/diyuser2/docker/Dockerfile
/home/diyuser2/docker')
```

- For the second change, after we create our Docker image, we can use it to do our build instead of the library routine. Add an **"inside"** method call that will use our new image (and optionally mapping the gradle cache to reduce the download traffic).

```
myImg.inside('-v /home/jenkins2/.gradle:/home/jenkins2/.gradle') {
}
```

- Add two shell commands within the **docker.inside** closure/block that will use the version of docker in our new image. One will simply print out the version and one will do the build. Add the lines in bold below.

```
myImg.inside('-v /home/jenkins2/.gradle:/home/jenkins2/.gradle') {
    sh 'gradle -version'
    sh 'gradle -g /home/jenkins2/.gradle clean compileJava -x test'
}
```

- Save your changes, and quit the editor. Push the changed Jenkinsfile out to the repository.

```
git commit -am "update for lab 6"
git push origin lab6
```

## In the Jenkins application:

- Go into your multibranch pipeline project and **Scan Multibranch Pipeline Now**. The job for lab 6 should run with the additional creation and use of the docker image with Gradle 4.
- Look at the **Console Output for the lab6 build** and note the output from the Compile stage that indicates we are using the Gradle 4.9 version.