Building a Deployment Pipeline with Jenkins 2

Class Labs

Version 1.4 – 7/4/19

Brent Laster

Lab 1 – Node setup

Purpose: In this first lab, we'll create a new worker node/agent to use to execute our pipeline. Then, we'll add the basic node block to use it. (Note that you wouldn't normally create a node/agent on the same machine as the master, but we'll do this here for simplicity.)

1. Start Jenkins by clicking on the "**Jenkins**" shortcut on the desktop OR opening the Firefox browser and navigating to "**http://localhost:8080**".

2. Log in to Jenkins.   User = **jenkins2** and Password = **jenkins2**

(**Note: If at some point during the workshop you try to do something in Jenkins and find that you can't, check to see if you've been logged out. Log back in if needed**.)

3. Click on the "**Manage Jenkins**" link in the menu on the left-hand side. Next, look in the list of selections in the middle of the screen, and find and click on "**Manage Nodes**". (You may need to scroll down.)

Script Console
Executes arbitrary script for administration/trouble-shooting/diagnostics.

Manage Nodes
Add, remove, control and monitor the various nodes that Jenkins runs jobs on.

About Jenkins
See the version and license information.

4. On the next screen, notice that we already have several nodes listed here, including **master** and two **worker nodes**. Now, we're going to create a new worker node to use in our pipeline. Click on "**New Node**" on the menu on the left-hand side.

Back to Dashboard
Manage Jenkins
New Node
Configure

5. For "**Node name"**, type in

　　　　worker_node1

Click on the "**Permanent Agent**" radio button. Then click on "**OK**".

6. You should now be on the configuration screen for the new node.

For **Description**, you can enter "**Main pipeline worker node**" (or whatever you wish).

Set the "**# of executors**" to **2**.

In the **Remote root directory** field, enter "**/home/jenkins2/worker_node1**". This is the working area for the node.

For **Labels** enter "**worker_node1 docker**". These are identifiers that can be used in the pipeline to refer to the node.

For **Launch method**, click the down arrow at the far right, and select "**Launch agent agents via SSH**". This will use SSH keys to launch the nodes.

Under **Launch method,** for **Host**, enter "**diyvb2**".

Under **Launch method**, for **Credentials**, click the drop-down arrow, and select "**jenkins 2 (Jenkins 2 SSH)**". These are SSH credentials that have already been setup on the system.

Under **Launch method**, for **Host Key Verification Strategy**, click the drop-down arrow, and select "**Non verifying Verification Strategy**". We're using this less secure strategy here to keep things simple.
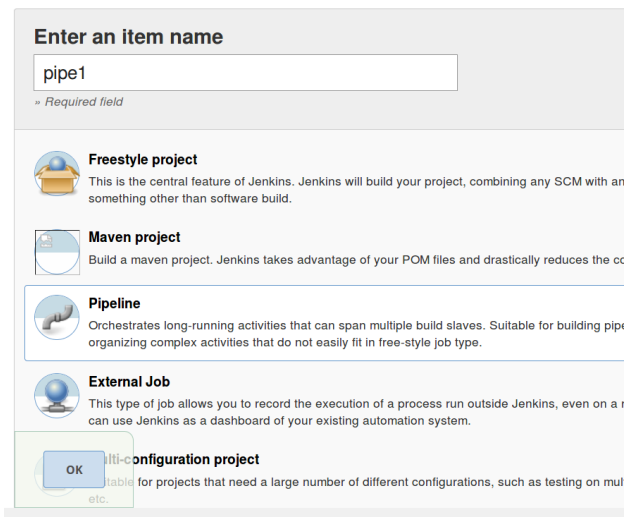

7. Leave everything else the same, and click **Save** (at the bottom of the page). You'll now be back on the **Nodes** screen. Click on the **ENABLE AUTO REFRESH** link in the upper right corner. After a moment, you should see your new node showing as running.

8. Now, let's create a new pipeline to use with our node. In the upper left corner, click on the **Jenkins** link or "**Back to Dashboard**". On the home page, click on **New Item** and give it a name of "**pipe1**". Select **Pipeline** for the type and then click **OK**.

9. You'll now be on the **Configure** page for your new pipeline. Switch to the **Pipeline** tab at the top (or scroll down the page to the **Pipeline** section. In that section is a place to type in the code for our pipeline. We'll add the main block for our pipeline here to run on our new node.

10. Enter (or copy and paste) the code below into the text box.

```
node('worker_node1') {



}
```
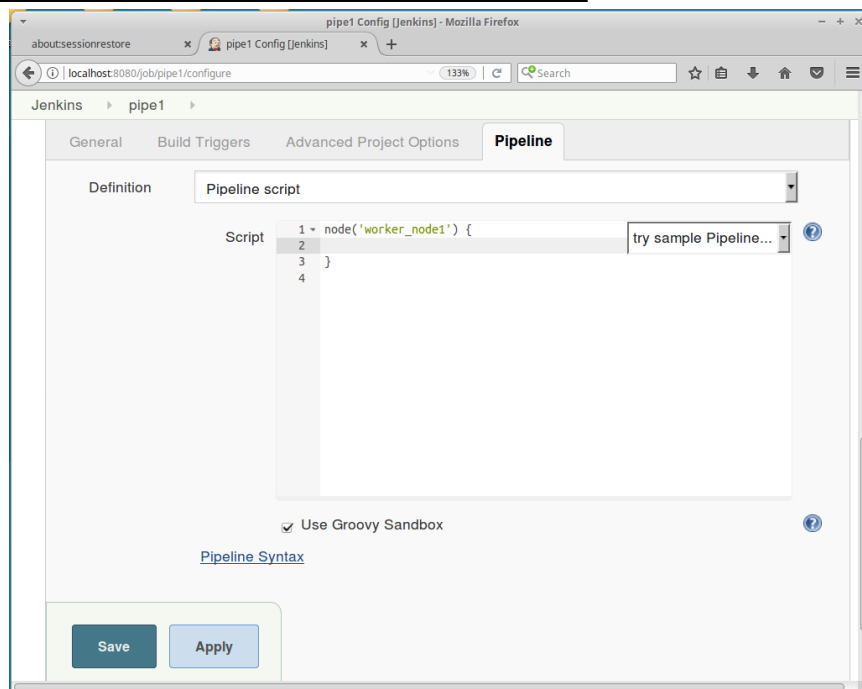

Click the "Apply" button to save your work, but leave this page open for the next lab.

© 2019 Brent Laster

Lab 2 – Adding stages, configuring tools, and using shared libraries

Purpose: In this lab, we'll start to fill out our pipeline by adding stages to retrieve the source code and do the builds. We'll see how to add shared libraries to our pipeline to encapsulate functionality.

1. For each of the major parts of our pipeline, we are going to create a stage for the code related to that part. First, we'll create a stage to retrieve the source. In your Jenkins session, you should still be on the page from the first lab where we added the code for the **node** definition. Within the node definition, we'll add our first stage. Add the following code (in bold) inside the node definition. (You can leave out the comments (lines starting with //) if you want.)

```
node('worker_node1') {
   stage('Source') {
      // Get code from our git repository
      git 'git@diyvb2:/home/git/repositories/workshop.git '
   }
}
```

The source code to pull down is located on our local system in **/home/git/repositories/workshop.git** in the default branch **master** branch.

2. Now, with our first stage complete, let's move on to the next one – the **Compile** stage to build our code. Go ahead and add the framework for the stage under the **Source** stage.   We'll fill in the command next. Add the lines in bold into your code as shown.

```
node('worker_node1') {
   stage('Source') {
      // Get code from our git repository
      git 'git@diyvb2:/home/git/repositories/workshop.git '
   }
   stage('Compile') {
      // Run gradle to execute compile

   }
}
```

Save your changes when done.

3. Before we can use Gradle in our pipeline, we need to have it installed and tell Jenkins where it is. Gradle is installed on our VM in **/usr/share/gradle**. To tell Jenkins about this, go to the Jenkins homepage/dashboard (http://localhost:8080).

Click on **Manage Jenkins** in the left menu.

Then click on **Global Tool Configuration.**

© 2019 Brent Laster
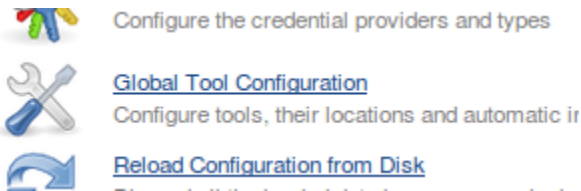
4. Scroll down and find the **Gradle** section.

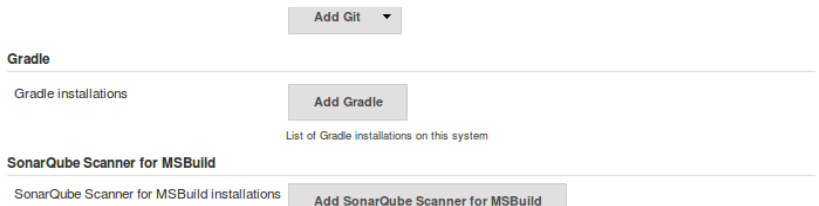**5.** Click the **Add Gradle** button. A new section pops up on the screen. <u>Uncheck</u> the **Install automatically** box since we don't want Jenkins to install Gradle for us. (We already have it installed.) Then, for **name**, enter **gradle4** – so we can refer to it by this label. For the **GRADLE_HOME** value, fill in where Gradle is available - **/usr/share/gradle** .

Your screen should look like the one below.   Click the **Save** button to save your global configuration changes.
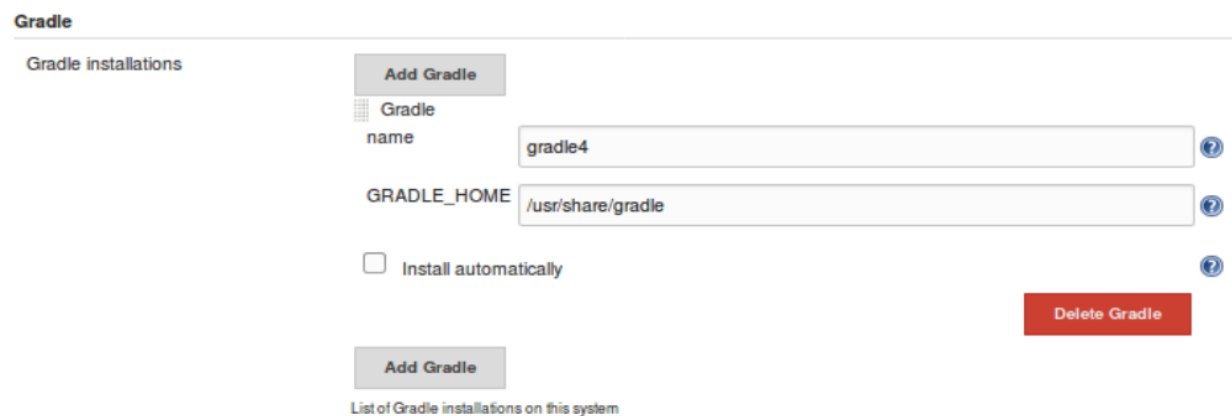
6. Now that we have Gradle installed and configured for use in Jenkins, we're ready to call it in our pipeline. We're going to do this using a shared library, so we can see how those work. We already have a simple shared library routine that does this. To see the routine, first open a terminal window by going to the VM's desktop and clicking the **Terminal Emulator** icon.
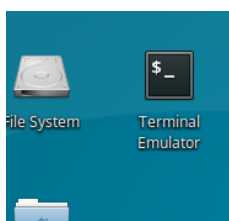
Then, switch to the local area for our **shared_libraries** project and print out the file on screen.   Type the commands below in the terminal window to do this.

© 2019 Brent Laster

Brent Laster

cd shared_libraries/vars

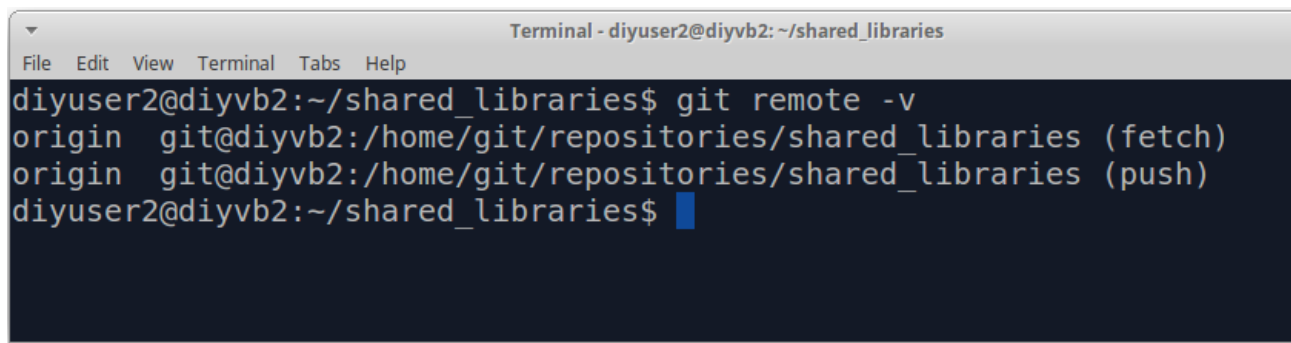**cat gbuild4.groovy**

```
diyuser2@diyvb2:~$ cd shared_libraries/vars
diyuser2@diyvb2:~/shared_libraries/vars$ cat gbuild4.groovy
def call (args) {
    sh "${tool 'gradle4'}/bin/gradle ${args}"
}
```

7. This code has already been pushed to a remote Git repository on our VM. This is where Jenkins will download it from as a shared pipeline library. To see the location of the code, we can ask Git where the remote repository is. Type the following in the same directory in the same terminal window.

   git remote -v

   The output from this command shows the remote repository location where the code is pushed to and where it can be fetched from.

```
                    Terminal - diyuser2@diyvb2: ~/shared_libraries
File   Edit   View   Terminal   Tabs   Help
diyuser2@diyvb2:~/shared_libraries$ git remote -v
origin  git@diyvb2:/home/git/repositories/shared_libraries (fetch)
origin  git@diyvb2:/home/git/repositories/shared_libraries (push)
diyuser2@diyvb2:~/shared_libraries$
```

8. Now that we have our code to invoke Gradle in the shared library repository, let's look at how Jenkins makes that available. Go back to the Jenkins homepage/dashboard (http://localhost:8080**).**

   Click on **Manage Jenkins.**

   Click on **Configure System.**

   Scroll down on this page until you find the **Global Pipeline Libraries** section. This is where we load in the library from Git that we were just looking at. There is nothing to change here, but look at the different fields – particularly the **Project Repository** field. Notice that the location in here is the same location on the Git remote where we have our shared pipeline library.

Sharable libraries available to any Pipeline jobs running on this system. These libraries will be trusted, meaning they run without "sandbox" restrictions and may use @Grab.

Library
Name

Utilities2

Default version

master

Currently maps to revision:
59070287ef9a483c3f6f544a481369570591faf5

Load Implicitly ☐

Allow default version to be overridden ☑

Include @Library changes in job recent changes ☐

Retrieval method

◉ Modern SCM

Source Code Management

◉ Git

Project Repository

/home/git/repositories/shared_libraries

Credentials

jenkins2 (Jenkins2 SSH)

Add ▾

Behaviors

Within Repository

**Discover branches**

Delete

Additional

Add ▾

9. Now, we need to go back to our pipeline, bring in the shared library with the gradle build function (gbuild4) and call it to build our code. Go to this URL: **http://localhost:8080/job/pipe1/configure** (or go to the **dashboard**, then to the **pipe1** project, and click **configure**).

   Switch to the **Pipeline** tab (or scroll down) to the text entry box with our code. Add the lines in bold below. (We will discuss what these lines are doing in the lecture. Note the underscore character that is required at the end of the @Library line. Also, those are parentheses around 'Utilities2', not brackets.)

```
    @Library('Utilities2') _
   node('worker_node1') {
     stage('Source') {
         // Get code from our git repository
        git 'git@diyvb2:/home/git/repositories/workshop.git '
     }
     stage('Compile') {  // Compile and do unit testing
        // Run gradle to execute compile and unit testing
        gbuild4 "clean compileJava -x test"
     }
   }
```

10. Before we leave this page, let's setup our job in Jenkins to poll the SCM to watch for changes in source control and then trigger a build. Switch to the **Build Triggers** tab (or scroll up to the **Build Triggers** section of the page).

© 2019 Brent Laster

Click the box for **Poll SCM.** Type a string of five asterisks separated by spaces (**" * * * * *"**) in the text box next to **Poll SCM.**   You'll see a warning message and can ignore it.
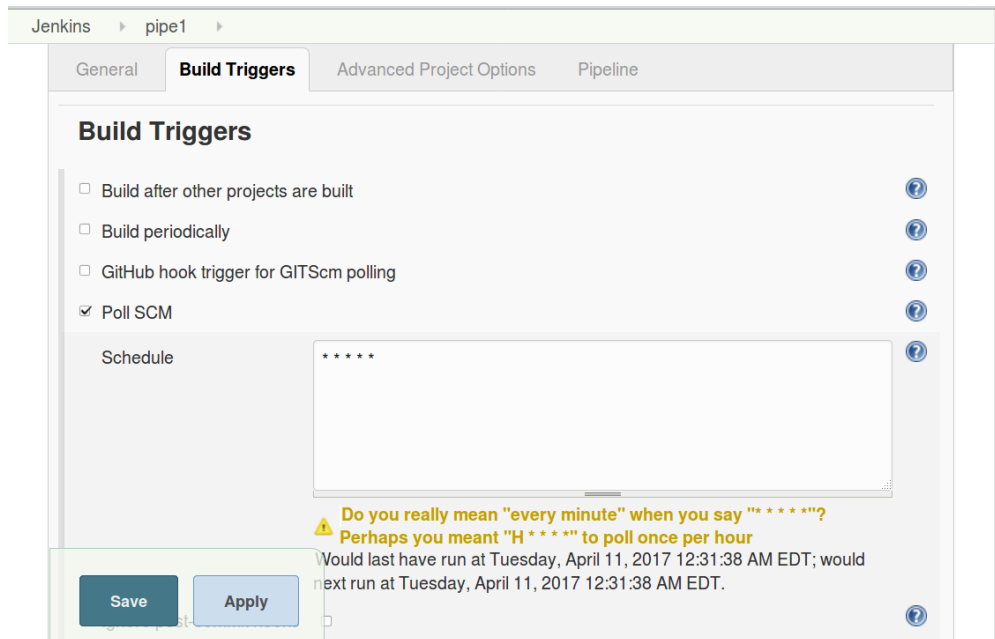
11. Click on **Save**. After a minute or so, Jenkins should automatically detect that there is "new" code it hasn't built and start a build running. After the build completes, you will be able to see the results in the **Staging View**. Notice the matrix here with rows for each build (we only have 1 so far) and columns corresponding to the stages that we have defined so far (**Source** and **Build**). Hover over each box and look at the logs from each by clicking on the **Logs** link in the pop up window. (Note: You may need to **Disable Auto Refresh** in the upper right corner to keep the pop ups on the screen.)
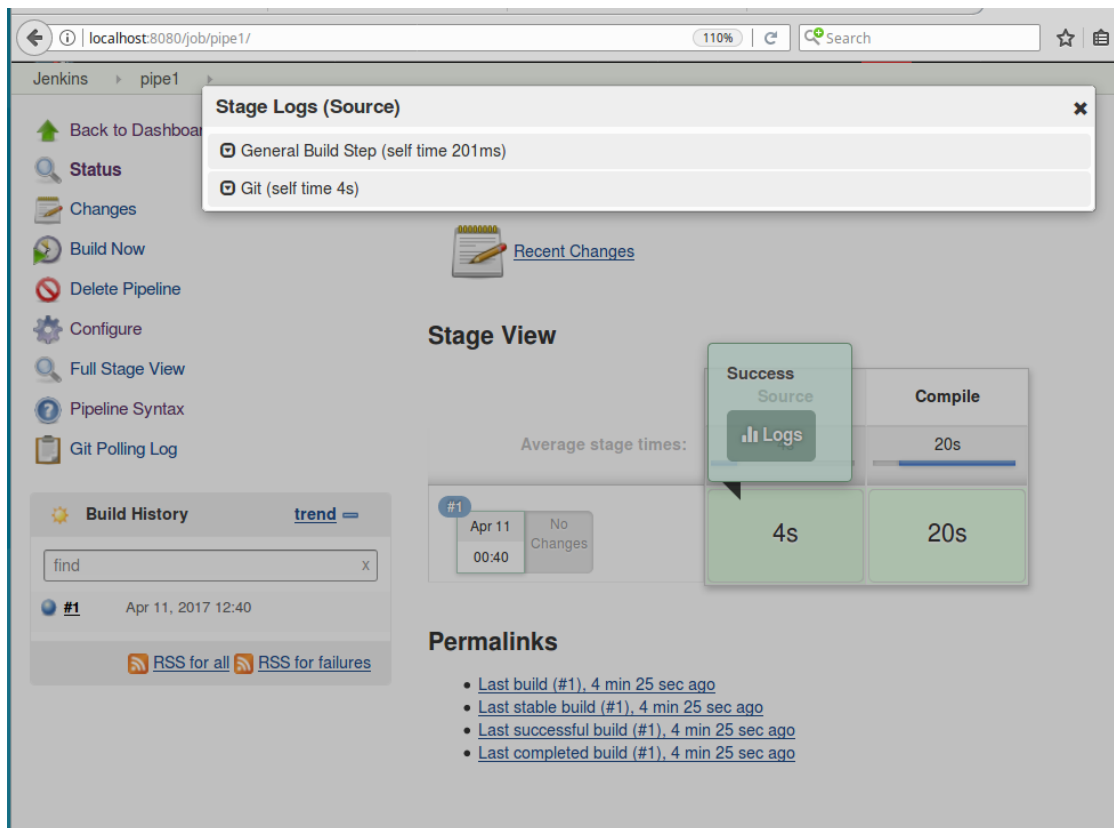
Lab 3 – Adding in testing

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

    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.

```
stage('Compile') {  // Compile and do unit testing
    // Run gradle to execute compile and unit testing
    gbuild4 "clean compileJava -x test"
}
stage('Unit Test') {

}
stage('Integration Test')
{

}
```

© 2019 Brent Laster

2. For our unit tests, we are going to execute them in parallel. So, within the **Unit Test** stage, add a **Parallel** step (block). Do this by adding the lines below in bold. (Note that **parallel** is lower-cased and those are **parentheses** after parallel, **NOT brackets**.)

```
stage('Unit Test') {
    parallel (


    )
}
stage('Integration Test')
{


}
```

3. 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.

© 2019 Brent Laster

Brent Laster
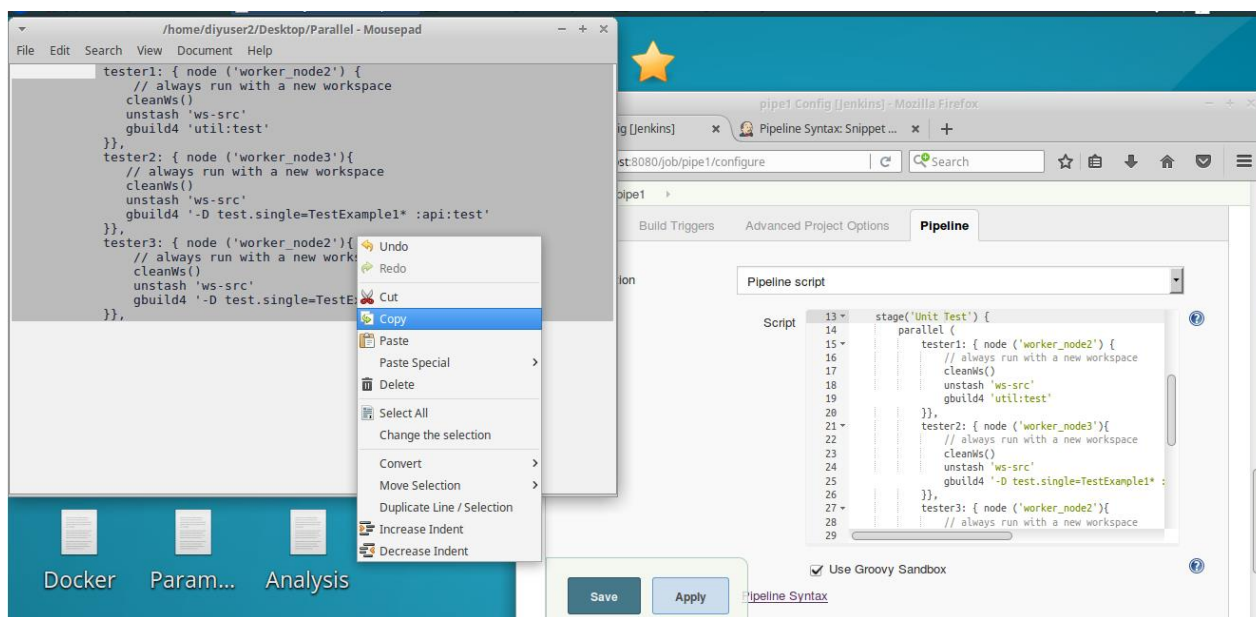
```
Pipeline script                                                          ▼

Script   13 ▾     stage('Unit Test') {
         14          parallel (
         15 ▾          tester1: { node ('worker_node2') {
         16                // always run with a new workspace
         17                cleanWs()
         18                unstash 'ws-src'
         19                gbuild4 'util:test'
         20            }},
         21 ▾          tester2: { node ('worker_node3'){
         22                // always run with a new workspace
         23                cleanWs()
         24                unstash 'ws-src'
         25                gbuild4 '-D test.single=TestExample1* :api:test'
         26            }},
         27 ▾          tester3: { node ('worker_node2'){
         28                // always run with a new workspace
         29

         ☑ Use Groovy Sandbox

Pipeline Syntax
```

4. 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.

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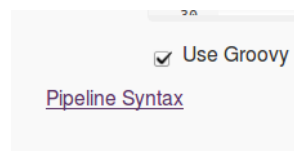
6.   You'll now be in the **Snippet Generator**.   In the drop-down list of **Sample Steps**, find and select the **stash** command. A set of fields for the different named parameters associated with the stash command will pop up. You can click on the blue (with a ?) **help button** for any of the parameters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

7. 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.
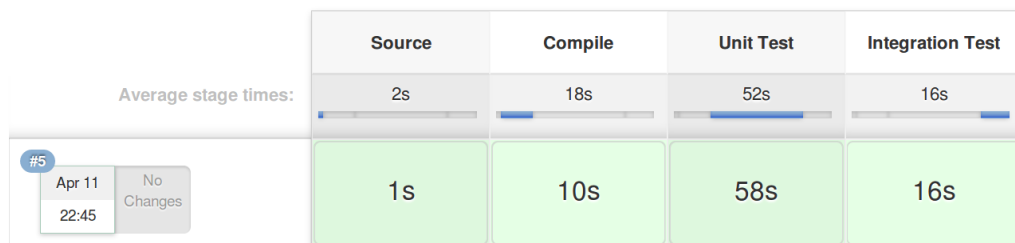

8. Finally, we'll add the commands to run the integration tests. We'll add a line to setup the test database we use for the integration testing and then invoke Gradle to run the **integrationTest** task that we have defined in our

© 2019 Brent Laster

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"
    gbuild4 'integrationTest'
}
```

9. **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

10. 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.

```
[Pipeline] [tester1] node
[Pipeline] [tester2] node
[Pipeline] [tester3] node
[tester1] Running on worker_node2 in /home/jenkins2/worker_node2/workspace
[tester3] Running on worker_node2 in /home/jenkins2/worker_node2/workspace
[Pipeline] [tester1] {
[Pipeline] [tester3] {
[tester2] Running on worker_node3 in /home/jenkins2/worker_node3/workspace
[Pipeline] [tester1] cleanWs
[tester1] [WS-CLEANUP] Deleting project workspace...[WS-CLEANUP] done
[Pipeline] [tester3] cleanWs
[tester3] [WS-CLEANUP] Deleting project workspace...[WS-CLEANUP] done
[Pipeline] [tester2] {
[Pipeline] [tester1] unstash
[Pipeline] [tester3] unstash
[Pipeline] [tester2] cleanWs
[tester2] [WS-CLEANUP] Deleting project workspace...[WS-CLEANUP] done
[Pipeline] [tester2] unstash
[Pipeline] [tester3] tool
[Pipeline] [tester3] sh
[tester3] [pipe1@2] Running shell script
[Pipeline] [tester1] tool
[Pipeline] [tester1] sh
[tester3] + /usr/lib/gradle/4.4.1/bin/gradle -D test.single=TestExample2*
[tester1] [pipe1] Running shell script
```

© 2019 Brent Laster

Lab 4 – Analysis

Purpose: In this lab, we'll see how to configure SonarQube to work with our pipeline. We'll set up a web hook to have SonarQube notify us when it's done. We'll also add in the Jacoco integration for code coverage and see how that works when we run our pipeline.

1. Start out by adding the empty stage definition for an **Analysis** 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.

```
   stage('Integration Test')
   {
      // set up integration database
      sh "mysql -uadmin -padmin registry_test < registry_test.sql"
      gbuild3 'integrationTest'
   }
   stage('Analysis')
   {

   }
```

2. We will do our analysis part using the tool **SonarQube**. **SonarQube** is already installed and configured globally on the system. If you want to look at the configuration for this in Jenkins, click "**Apply**" to save your changes and go to http://localhost:8080/manage - the url for the Manage Jenkins functionality. From here, click on the top entry to **Configure System**. Scroll down the page until you find the **SonarQube Servers** section. (This integration is provided by the SonarQube plugin.)

3. Switch back to the **configure** page for **pipe1** if not already there. Let's look at one of the DSL commands available for us to work with SonarQube.   Click on the **Pipeline Syntax** link at the bottom of the page. This puts us back into the **Snippet Generator**.

In the **Sample Steps** dropdown, find the **withSonarQubeEnv** method**.**   There are several of these types of "with environment" methods available in Jenkins DSL. They allow us to wrap other steps with specific environment variables or settings needed to run certain steps**.** Click on the blue Help button (with the ?) if you want to see more information.

Next, find the **waitForQualityGate** entry. A Quality Gate here is a set of standards defined in SonarQube that the analysis must meet or exceed for the code being analyzed to pass. Select this item and then click on the blue Help button (with the ?) and read about how this works.

Notice that it basically waits for a webhook that has been setup in SonarQube to signal that the analysis is complete before moving on. We're going to setup such a webhook now.

4. Open the SonarQube instance on your VM by going to http://localhost:9000 and logging in (upper right corner) with user **admin** and password **admin**).

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. Fill in the fields as follows:

   Name: **Jenkins-sonar**
   URL: http://localhost:8080/sonarqube-webhook/ **(**Note: The trailing slash on the URL is important!)

Once you have filled in the fields, click the **Save** button. Then the webhook is in place. Switch back to your Jenkins **pipe1 configure** page.

© 2019 Brent Laster

5. Now we are ready to start filling in our **Analysis** stage. We have seen the SonarQube server configuration. To actually run SonarQube against our project, we use an executable called a **sonar runner** that is installed on the system at **/opt/sonar/bin/sonar-runner**.

Add the lines below in bold at the start of the **Analysis** stage.

```
stage('Analysis')
{
    withSonarQubeEnv('Local SonarQube') {
        sh "/opt/sonar-runner/bin/sonar-runner -X -e"
    }
}
```

Then, we'll add in the step that we saw in the Snippet Generator to wait on the web hook that we created. Add the lines in bold at the end of the **Analysis** stage. (Note that we are wrapping this in a Jenkins timeout step just to ensure that we don't get stuck waiting on a hung server or other issue.)

```
stage('Analysis')
{
    withSonarQubeEnv('Local SonarQube') {
        sh "/opt/sonar-runner/bin/sonar-runner -X -e"
    }

    timeout(time: 1, unit: 'HOURS') {
      def qg = waitForQualityGate()
      if (qg.status != 'OK') {
          error "Pipeline aborted due to quality gate failure: ${qg.status}"
      }
    }
}
```

6. There is one other item we want to add in to our **Analysis** stage - code coverage using a tool called **Jacoco**. For expediency, we'll just add in the code here since we've discussed it in the lecture. You can find this code in the desktop **Analysis** file. So you can copy and paste it between the sonar-runner invocation and then wait for the quality gate. Add the lines in bold below into your code.

© 2019 Brent Laster

```
withSonarQubeEnv('Local SonarQube') {
    sh "/opt/sonar-runner/bin/sonar-runner -X -e"
  }

  step([$class: 'JacocoPublisher',
    execPattern:'**/**.exec',
    classPattern: '**/classes/main/com/demo/util,**/classes/main/com/demo/dao',
    sourcePattern: '**/src/main/java/com/demo/util,**/src/main/java/com/demo/dao',
    exclusionPattern: '**/*Test*.class'])


  timeout(time: 1, unit: 'HOURS') {
    def qg = waitForQualityGate()
    if (qg.status != 'OK') {
        error "Pipeline aborted due to quality gate failure: ${qg.status}"
    }
  }
}
```

7. **Save** your changes and **Build Now**. After this, you'll have another stage in the pipeline in your stage view.

	Source	Compile	Unit Test	Integration Test	Analysis
Average stage times:	2s	13s	54s	14s	21s
#7 Apr 12 00:53 No Changes	1s	5s	51s	9s	17s (paused for 2s)
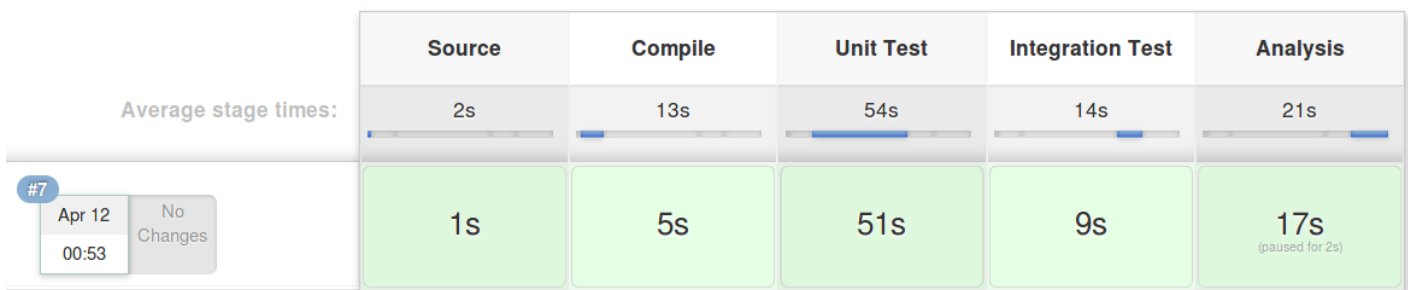
8. You should also see some new symbols and links available to you that relate to SonarQube. In the left-hand menu will be a link to **SonarQube**. Note the symbol next to it. This same symbol should be next to your last build.   Click on either the symbol or the **SonarQube** link in the menu to go to the SonarQube analysis for this build.
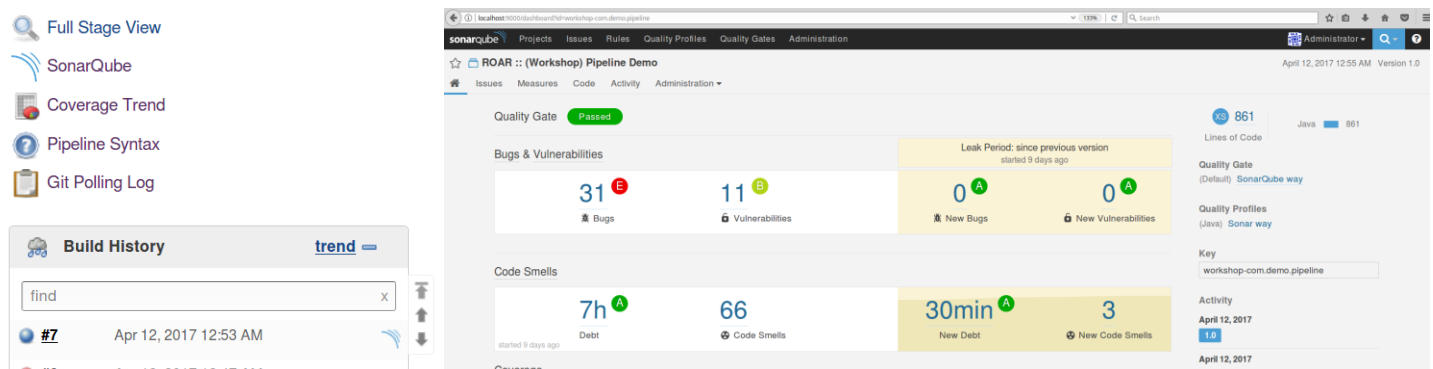
9. When you are done clicking around in the SonarQube report, go back to the **Stage View** page for **pipe1** (http://localhost:8080/job/pipe1/). You may have noticed that in the SonarQube report we did not have line

coverage information. This is because we chose to add it separately for demo purpose. However, you should now see a graph above the stage view, with the title **Code Coverage Trend**, showing some info about line coverage.

Click on that graph (or click the **Coverage Trend** link in the left-hand menu and click again on the larger graph that pops up). You can now see more of the code coverage information for the project. You can also click around and drill into the details.
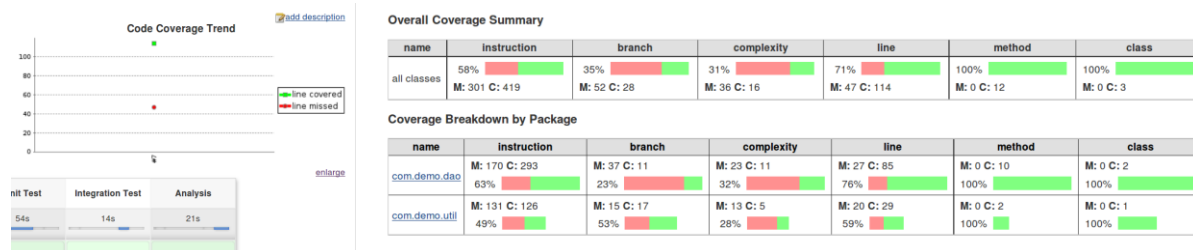
Lab 5 – Assembly

Purpose: In this lab, we'll 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 **Analysis** stage but before the closing bracket. Lines to add are shown in bold below.

```
            if (qg.status != 'OK') {
                error "Pipeline aborted due to quality gate failure: ${qg.status}"
            }
        }
      }
   stage('Assemble')
   {

   }
}
```

2. Now, 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**

© 2019 Brent Laster

Name: PATCH_VERSION
Default Value: $BUILD_NUMBER

Name: BUILD_STAGE
Default Value: SNAPSHOT

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 by updating the **gradle.properties** file in our project with the 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.

© 2019 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}")
}
```

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}")

gbuild4 '-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 have to 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.

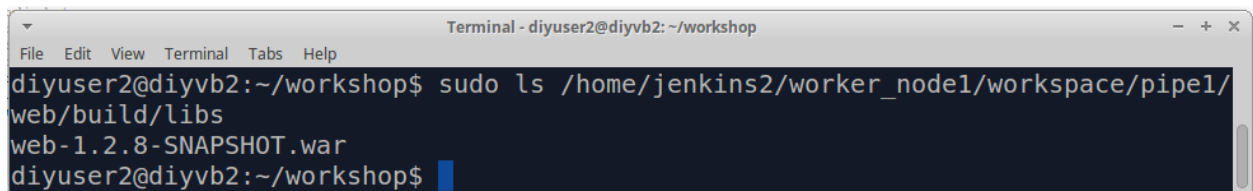7. You should have a successful build with another stage in your pipeline.

You can open the console log for the most recent build and find the place where the code from github was pulled in.

```
[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
```

8. 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 6 – Artifactory

Purpose: In this lab, we'll see how to integrate Jenkins and Gradle with Artifactory to deploy our war file artifact into an Artifactory repository and a way to retrieve the latest version of our artifact from Artifactory.

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

```
            gbuild4 '-x test build assemble'
        }
        stage ('Publish Artifacts')
        {

        }
```

2. 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 clicking the "**Apply**" button to save your changes, then 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.

© 2019 Brent Laster

3. 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**. Steps a-h below will guide you through adding the commands you need to setup and execute the publish to Artifactory using Gradle.

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

Add the commands in each step below (a-h) to your new stage. **Note – all of these lines are in a file on your desktop named Publish that you can copy and paste from.**

a. First, we need to configure a couple of things.

To point the Artifactory object to our installed instance, add the following line in your stage block. (Here, **LocalArtifactory** is the **Server ID** we gave it in the global configuration.)

```
def  server = Artifactory.server "LocalArtifactory"
```

b. Now, we'll 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. Add the lines below.

```
def artifactoryGradle = Artifactory.newGradleBuild()

artifactoryGradle.tool = "gradle4"
```

c. Next, we'll 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. Artifactory can also publish info about each build. We'll have it do that as well. Add the lines below.

```
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). Add these lines to accomplish those objectives.

```
artifactoryGradle.deployer.deployMavenDescriptors = true

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

f. We want to tell Jenkins that we aren't already loading the related Gradle plugin (com.jfrog.artifactory) in our Gradle script. So it needs to do it. We do that with the line below. Add it.

```
artifactoryGradle.usesPlugin = false
```

g. Finally, we are ready to run the Artifactory Gradle build and invoke Gradle's built-in **artifactoryPublish** task. Add this line to do that.

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

h. Just one more – add the line below to publish the build info to Artifactory as well.

```
server.publishBuildInfo buildInfo
```

Now, you should have your completed **Publish Artifacts** stage.
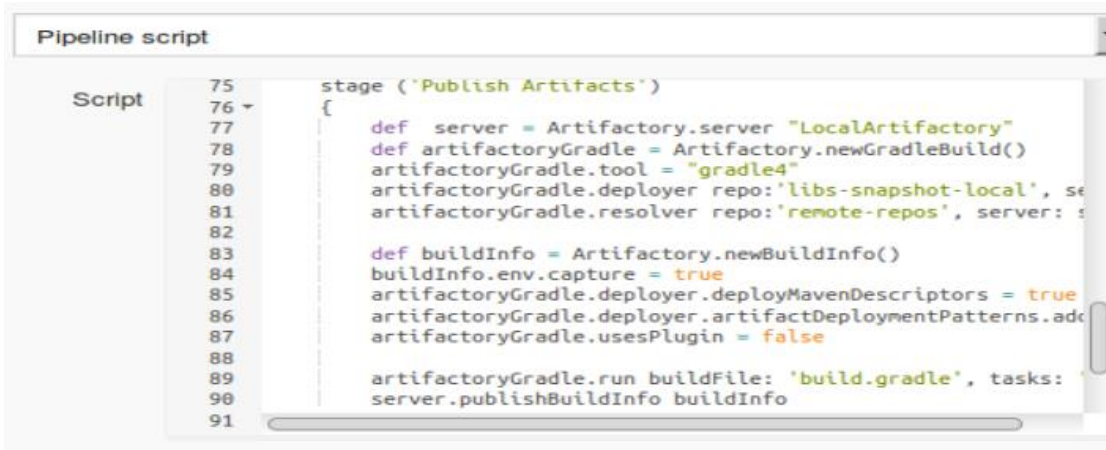
4. Our next stage is the one that will retrieve the latest version from Artifactory. We may not always want to retrieve the latest version, but we'll show how to do it here via a kludge process since the free version of Artifactory doesn't support this.

© 2019 Brent Laster

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.

We again are storing this code in an external file – in this case in the resources directory of our shared library.

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

```
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>\([^<]*\)<\/lates
t>.*/\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 $url
diyuser2@diyvb2:~$
```

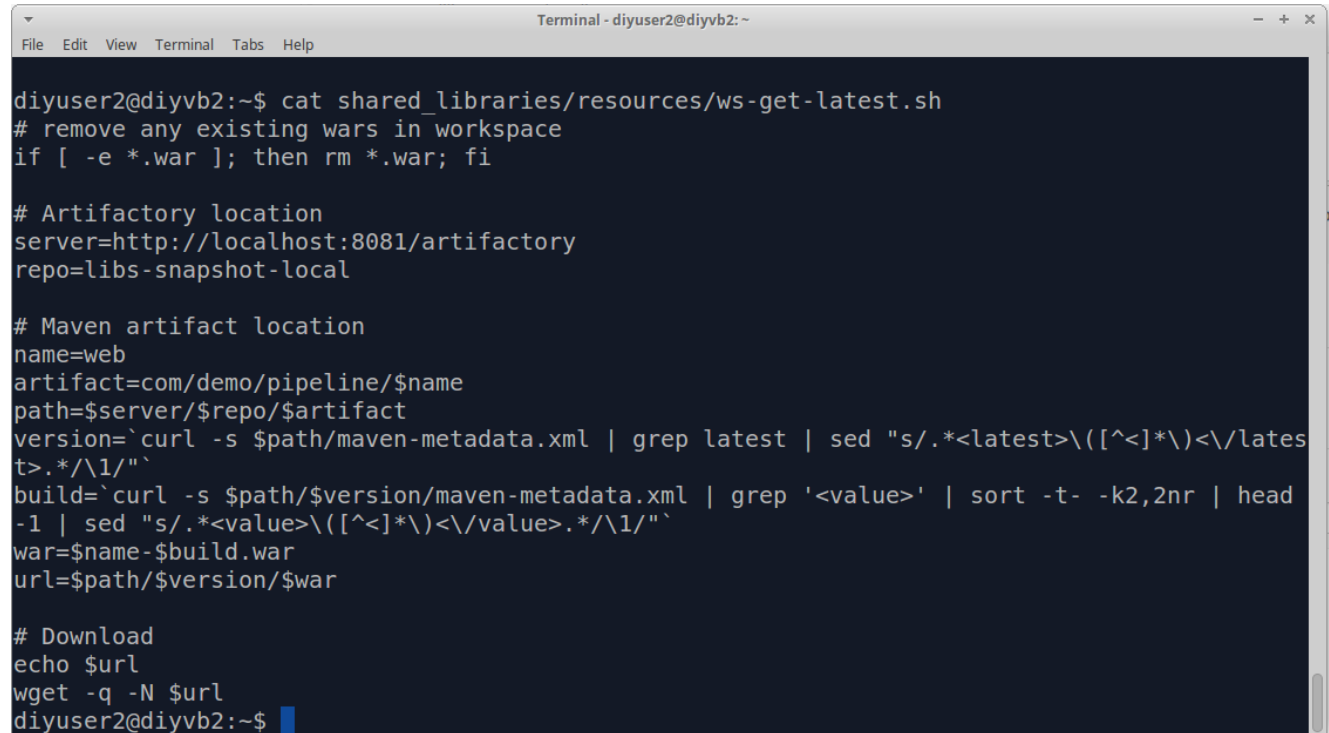5. To invoke this code in our stage, we will use the **libraryResource** step to load it. Then we can just pass the code to the **sh** step and have it executed. Add the stage below in bold into your pipeline.

```
stage('Retrieve Latest Artifact')  {
       def getLatestScript = libraryResource 'ws-get-latest.sh'
       sh getLatestScript


}
```

6. 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.

© 2019 Brent Laster

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

7. 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.

	Source	Compile	Unit Test	Integration Test	Analysis	Assemble	Publish Artifacts	Retrieve Latest Artifact
Average stage times:	1s	10s	56s	15s	21s	11s	16s	3s
#13 Apr 13 00:01 No Changes	1s	7s	58s	17s	21s (paused for 3s)	10s	19s	3s

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.)

```
[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.war
[Pipeline] stash
Stashed 1 file(s)
[Pipeline] }
```

Lab 7 – Deploy to Docker

Purpose: In this lab, we'll see how to deploy our war file into a docker container.

1. For the deployment to Docker, we're going to add a new stage as always. But we'll do a couple of things differently up front in our stage.
   a. We'll pull down the input files for Docker to use in building the images from a Git repository on the system.
   b. We'll run the processes on a different node so we don't overwrite the workspace we already have on worker_node1.
   c. We want to deploy the latest artifact (war file) into the container that we are spinning up. So we can use the **unstash** command on the stash that we created at the end of the **Retrieve Latest Artifact** stage.

   To handle this, start out by adding the following lines to your pipeline. (These go after the **Retrieve Latest Artifact** stage and before the ending bracket of the pipeline.)

```
stage('Deploy To Docker') {
    node ('worker_node3') {
        git 'git@diyvb2:/home/git/repositories/roarv2-docker.git'
        unstash 'latest-warfile'

    }
}
```

2. The remaining steps in this process use a combination of shell commands driving Docker and the built-in **Docker** variable to
   a. Clean out any old images and containers
   b. Build images from **Dockerfiles** in our git repository. There are two images – one for the **webapp** (which pulls in the war) and one for the **database** piece.
   c. Start up and link together containers for each image. This results in our app running under Docker.
   d. Find (inspect) the container to get the ip address where it is running and display that so we can look at it.

   The following lines in bold need to be added to the stage to complete it. But for simplicity and avoiding typos, it is recommended you copy and paste them from the file named **Docker** on the desktop. Either way add these lines in the Deploy to Docker stage .

```
stage('Deploy To Docker') {
    node ('worker_node3') {
        git 'git@diyvb2:/home/git/repositories/roarv2-docker.git'
        unstash 'latest-warfile'


        sh "docker stop `docker ps -a --format '{{.Names}} \n\n`' || true"
        sh "docker rm -f  `docker ps -a --format '{{.Names}} \n\n`' || true"
        sh "docker rmi -f \$(docker images | cut -d' ' -f1 | grep roar) || true"

        def dbImage = docker.build("roar-db-image", "-f Dockerfile_roar_db_image .")
        def webImage = docker.build("roar-web-image", "--build-arg warFile=web*.war -f
Dockerfile_roar_web_image .")
```

© 2019 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)"


    }
}
```

3. **Save** your changes and **Build with Parameters.**   Once completed, you will have another stage in your pipeline.

	Source	Compile	Unit Test	Integration Test	Analysis	Assemble	Publish Artifacts	Retrieve Latest Artifact	Deploy To Docker
Average stage times:	1s	10s	56s	15s	21s	11s	16s	4s	14s
#14 Apr 13 00:48 No Changes	1s	7s	1min 1s	17s	22s (paused for 2s)	11s	16s	4s	14s

4. Now, open the **Console log** for the last build of **pipe1.**   Scroll down near the bottom of the console log and you should see the output of the Docker inspect command. It will look something like the figure below.

```
[pipe1] Running shell script
+ docker run -d --link 506540fcafbabb2b0116f6e96cbbe4096479f19bcd8b8e664a9b84d85530396d:mysql -p 8089:8080 roar-web-image
[Pipeline] dockerFingerprintRun
[Pipeline] sh
[pipe1] Running shell script
+ docker ps -q -l
+ docker inspect --format {{.Name}} is available at http://{{.NetworkSettings.IPAddress }}:8080/roar ac2b2a997346
/cranky_blackwell is available at http://172.17.0.3:8080/roar
[Pipeline] }
[Pipeline] // node
```

5. Click on the link after "**is available at**". This is a link to our webapp running in the Docker container. The ip address/url is on the Docker container. Once you click on that you should see the webapp as its being run in Docker.

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

==

END OF CLASS LABS

© 2019 Brent Laster

==

NOTES ABOUT THE IMAGE:

Jenkins: Running on http://localhost:8080   login: jenkins2 password: jenkins2

Sonar: Running on http://localhost:9000/sonar login: admin password: admin

Tomcat: Running on http://localhost:8084 Manager app login: admin password: admin   For deploying artifact login: tomcat password: tomcat

Mysql server: Running on port 3036   login: mysql password: mysql

Artifactory: Running on localhost:8081/artifactory login: admin password: admin