# Jenkins 2 – Beyond the Basics

**Revision 1.1 – 5/5/18**

**Brent Laster**

**IMPORTANT NOTES:**

1. **This course picks up where Jenkins 2 – Up and Running left off. You may either use your saved VM from that course or download the new J2BB image as described in the setup doc.**

2. **You must have internet access through your VM for some of the labs.**

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

**4. When the system tells you to save a file you've been working on in the editor (such as gedit), you will also need to exit the editor unless you started it running in the background.**

**5. For some labs, email functionality is used, so you will need your email information and have Jenkins configured as described in the setup doc.**

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

**Lab 1 - Multibranch projects**

**Purpose: In this lab, we'll see how to setup a multibranch pipeline project**

1. Start Jenkins by clicking on the "**Jenkins 2**" shortcut on the desktop OR opening the Firefox browser and navigating to "**http://localhost:8080**".

2. You should be on the login screen. Log in to Jenkins with 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. For this lab, we want to add some tests to our pipeline. The tests are in the "**test**" branch of our repository, so we will need to access them there. As well, we'd like Jenkins to automatically setup a project for our test branch.

4. To get Jenkins to recognize the branch in our project, we need to have a **Jenkinsfile** - similar to a build script - as a marker in the branch. Our cloned copy of our project - at **~/gradle-demo** already has one.

5. To see it, go to/open a terminal, cd to the **~/gradle-demo** project and **checkout** the **test** branch. Then cat the file.

**cd ~/gradle-demo**

**git checkout test**

**cat Jenkinsfile**

Jenkinsfile ×

```
#!groovy
@Library('Utilities@1.5')_
node ('worker_node1') {
try {
        stage('Source') {
            // always run with a new workspace
            cleanWs()
            checkout scm
        }
        stage('Build') {
            // Run the gradle build
            gbuild2 'clean build -x test'
        }
        stage ('Test') {
            // execute required unit tests in parallel

        }
    }
    catch (err) {
        echo "Caught: ${err}"
    }
    stage ('Notify') {
        // mailUser('<your email address>', "Finished")
    }
}
```

6. This file has a couple of changes that we need to talk through.

i. Notice the **#!groovy** declaration at the top. This is a best practice for Jenkinsfiles since they don't have the .groovy extension to denote them as being groovy code.

ii. Then we have our **@Library** annotation to load our shared library. It is followed by the "**_**" character instead of an **import** statement. The reason for this is that we have switched from using the **gbuild** routine defined in a package in **org.conf.Utilities** to a global variable (function) named gbuild2. Like the mailUser function we used in the last lab, this is defined in the vars subdirectory - specifically in in **~/shared-libraries/vars/gbuild2.groovy**. So we don't need the import any longer. We have the "_" character there because it is required to have something after an annotation **(@Library**).

iii. We've also changed our build step to use the call to **gbuild2** instead of **gbuild**.

The code for the **gbuild2** function is:

```
def call(args) {
  sh "${tool 'gradle32'}/bin/gradle ${args}"
}
```

This code is simpler than the **gbuild** code. Since global variables/functions already have the script context, we don't have to pass in the "script" object as we did for **gbuild.** Also, this is more portable between scripted and declarative pipelines. (More on that to come.)

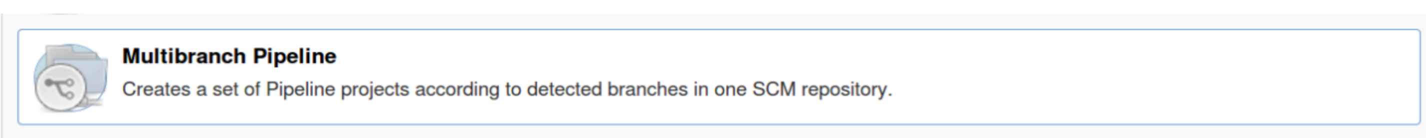
iv. Notice that this file has a different way of specifying how to get the source. It simply says **checkout scm**. This is a shortcut, available because, since this is a multibranch project, Jenkins already knows the project and branch by virtue of the Jenkinsfile being in source control.

v. Finally, notice the line "**cleanWs()**". This is telling Jenkins to wipe out (cleanup) the workspace before beginning the operation. This is specific to the node. The function is available due to the "Workspace Cleanup" plugin being installed. (Note: There is also a deleteDir() function to delete a directory.)

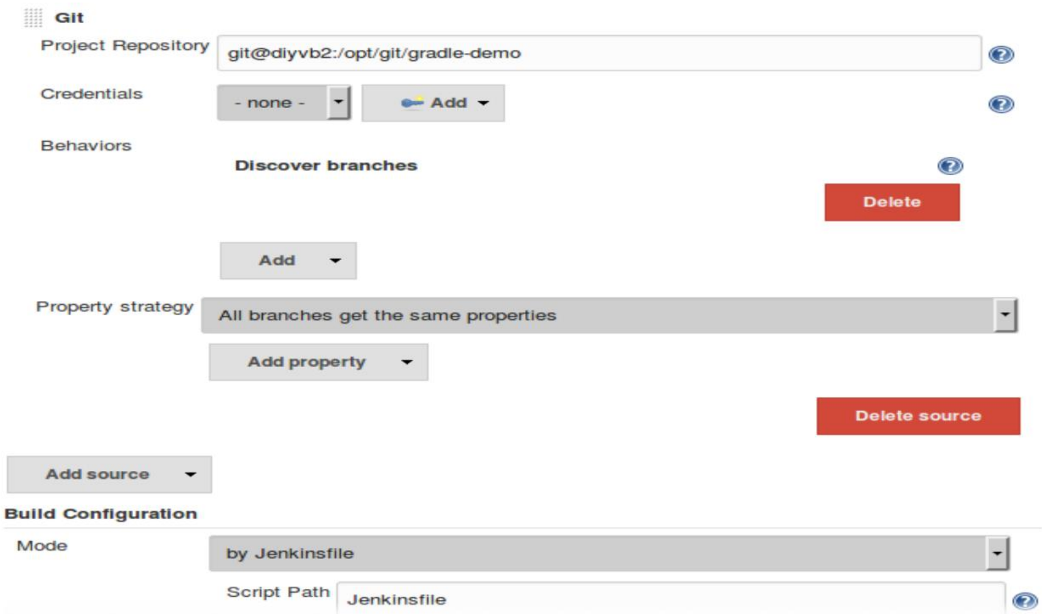
7. OPTIONAL. Currently the call to **mailUser** is commented out. If you want, you can edit the file (**gedit Jenkinsfile**), uncomment the line (remove the leading **//**), and replace **<your email address>** with a valid email address (and change the "**Finished**" message if you want). If you do this, then **save** the file and commit and push the updated file back into the repository.

```
$ git add Jenkinsfile

$ git commit -m "updated email address"

$ git push origin test
```

8. Now, we want Jenkins to automatically setup a project for this branch. From the dashboard, click on **New Item**. Give it a name of "**demo-all**" and select "**Multibranch Pipeline**" for the type. Then select **OK**.

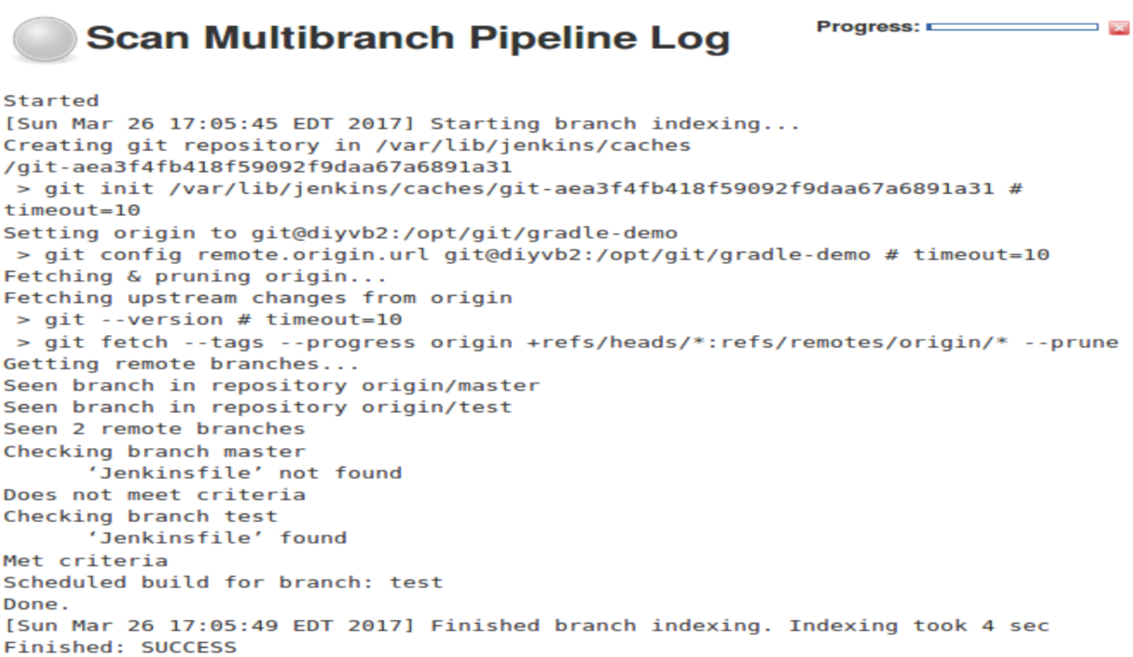



9. On the **Configuration** page, in the **Branch Sources** section, click on the "**Add source**" button. Select **Git** from the dropdown and enter our repository location (**git@diyvb2:/opt/git/gradle-demo**) for the **Project Repository** section. You can leave the rest of the Branch Sources section as is. Also, make sure that the **Build Configuration Mode** is set to "**By Jenkinsfile**".




10. Scroll down to the bottom of the page. Notice there is a **Pipeline Libraries** section there. Since we have already defined the **Global Shared Libraries**, we don't need to add anything here.

11. Now **Save** your changes. Jenkins now runs a **branch indexing** function looking for a **Jenkinsfile** in branches of the projects. Where it finds one, it can setup a build for it. Notice the log output identifying the master and test branches and the Jenkinsfile in the test branch.




```
Started
[Sun Mar 26 17:05:45 EDT 2017] Starting branch indexing...
Creating git repository in /var/lib/jenkins/caches
/git-aea3f4fb418f59092f9daa67a6891a31
 > git init /var/lib/jenkins/caches/git-aea3f4fb418f59092f9daa67a6891a31 #
timeout=10
Setting origin to git@diyvb2:/opt/git/gradle-demo
 > git config remote.origin.url git@diyvb2:/opt/git/gradle-demo # timeout=10
Fetching & pruning origin...
Fetching upstream changes from origin
 > git --version # timeout=10
 > git fetch --tags --progress origin +refs/heads/*:refs/remotes/origin/* --prune
Getting remote branches...
Seen branch in repository origin/master
Seen branch in repository origin/test
Seen 2 remote branches
Checking branch master
      'Jenkinsfile' not found
Does not meet criteria
Checking branch test
      'Jenkinsfile' found
Met criteria
Scheduled build for branch: test
Done.
[Sun Mar 26 17:05:49 EDT 2017] Finished branch indexing. Indexing took 4 sec
Finished: SUCCESS
```

12. Click on the **Up** link in the top left, then go back into **demo-all** and notice that Jenkins has created a new job for the **test** branch and run it.

================================================================================

**END OF LAB**

================================================================================

**Lab 2 - Running in parallel and across multiple nodes**

**Purpose: In this lab, we'll see how to add code to a script to do parallel processing and run across multiple nodes.**

1. Now we want to add the missing code for the **Test stage**. We have two tests in our Gradle project: **TestExample1.test** and **TestExample2.test**. To run either of these independently, we can use the Gradle invocation **-D test.single=<Test Name> test**.

2. For a simple demonstration of parallel execution, we'll run each of these tests on a separate node. Since we earlier defined the **gbuild2** function as running "**gradle**", we can reuse it here to call each test. We'll wrap each call to **gradle** (via the **gbuild2** function) in a different node. So our initial pass at a parallel block might look like this.

```
parallel (
        worker2: { node ('worker_node2'){
               // always run with a new workspace
                cleanWs()
                gbuild2 '-D test.single=TestExample1 test'
        }},
        worker3: { node ('worker_node3'){
               // always run with a new workspace
                cleanWs()
                gbuild2 '-D test.single=TestExample2 test'
        }},
       )
```

3. Go ahead and copy and paste this code in for the test stage in your **Jenkinsfile** in the **test** branch of the **gradle-demo** project. In a terminal session:

**gedit ~/gradle-demo/Jenkinsfile** (paste the code after the "// execute required unit tests…" line)

After adding the code in the editor, your file should look like the one below ( plus your email address if you changed that in the optional step in the previous lab ).

```
#!groovy
@Library('Utilities@1.5')_
node ('worker_node1') {
try {
      stage('Source') {
         // always run with a new workspace
         cleanWs()
         checkout scm
      }
      stage('Build') {
         // Run the gradle build
         gbuild2 'clean build -x test'
      }
      stage ('Test') {
         // execute required unit tests in parallel
         parallel (
            worker2: { node ('worker_node2'){
               // always run with a new workspace
               cleanWs()
               gbuild2 '-D test.single=TestExample1 test'
            }},
            worker3: { node ('worker_node3'){
               // always run with a new workspace
               cleanWs()
               gbuild2 '-D test.single=TestExample2 test'
            }},
         )
      }
   }
   catch (err) {
      echo "Caught: ${err}"
   }
   stage ('Notify') {
      // mailUser('<your email address>', "Finished")
   }
}
```

4. **Save** your changes, **stage**, **commit** and **push** the file.

**<Save changes to Jenkinsfile and quit the editor>**

**git add Jenkinsfile**

**git commit -m "updated for testing"**

**git push origin test**

5. Go back and re-run the **branch indexing** for **demo-all** again.

Click back to "**demo-all**".

Select the "**Scan Multibranch Pipeline Now**" left menu item.

6. Now switch back to the "**test**" project under "**demo-all**". It should have run again (if not, select "**Build Now**"). Within the **Stage View**, you should see that the **Test** stage failed. Let's figure out why.

Open up the **Console Log** output for the latest run. Scroll down to find the section where the output for the "**Test**" stage starts. Further down find the lines for "**not found in root project**" when **worker2** and **worker3** were trying to run the tests. Why is this?

…

**[Pipeline] // stage**
**[Pipeline] stage**

```
[Pipeline] { (Test)
[Pipeline] parallel
[Pipeline] [worker2] { (Branch: worker2)
[Pipeline] [worker3] { (Branch: worker3)
[Pipeline] [worker2] node
[worker2] Running on worker_node2 in /home/jenkins2/worker_node2/workspace/demo-all_test-ZQC7KCTZRKI2O5V23U5SXILJOWC6TUBFMWH2SZ3ND747GW7SHEKQ
[Pipeline] [worker3] node
[worker3] Running on worker_node3 in /home/jenkins2/worker_node3/workspace/demo-all_test-ZQC7KCTZRKI2O5V23U5SXILJOWC6TUBFMWH2SZ3ND747GW7SHEKQ
…
[worker2]
[worker2] FAILURE: Build failed with an exception.
[worker2]
[worker2] * What went wrong:
[worker2] Task 'test' not found in root project 'demo-all_test-ZQC7KCTZRKI2O5V23U5SXILJOWC6TUBFMWH2SZ3ND747GW7SHEKQ'.
[worker2]
[worker2] * Try:
[worker2] Run gradle tasks to get a list of available tasks. Run with --stacktrace option to get the stack trace. Run with --info or --debug option to get more log output.
[worker2]
[worker2] BUILD FAILED
[worker2]
[worker2] Total time: 3.862 secs
[worker3]
[worker3] FAILURE: Build failed with an exception.
[worker3]
[worker3] * What went wrong:
[worker3] Task 'test' not found in root project 'demo-all_test-ZQC7KCTZRKI2O5V23U5SXILJOWC6TUBFMWH2SZ3ND747GW7SHEKQ'.
[worker3]
[worker3] * Try:
[worker3] Run gradle tasks to get a list of available tasks. Run with --stacktrace option to get the stack trace. Run with --info or --debug option to get more log output.
[worker3]
[worker3] BUILD FAILED
[worker3] …
```

7. The reason we can't run the **Test** stage on **worker2** (**worker-node2**) is because the cloned content is only on **worker-node1** (where the original node was running and where the source management clone was done). Fortunately, Jenkins provides a way to save and share content between nodes with the **stash** command. To begin using this, we first need to **stash** the **src** directory (where the test content is) and the **build.gradle** file from the worker-node1 node.

8. Edit the **Jenkinsfile** again. Add a line after the **checkout scm** command like this:

```
stash name: 'test-sources', includes: 'build.gradle,src/test/'
```

9. For the nodes where we are running the parallel tests, we use the corresponding unstash command to recover the files needed from the original node. Add lines to do the "un-stashing" **after** the step call to clean up the workspace in each **node** specification in the **parallel** block. (See listing below.)

**unstash 'test-sources'**

When you have made the changes, your file should look like this (with the **new lines in bold**).

```
#!groovy
@Library('Utilities@1.5')_
node ('worker_node1') {
try {
    stage('Source') {
      // always run with a new workspace
      cleanupWs()
      checkout scm
      stash name: 'test-sources', includes: 'build.gradle,src/test/'
    }
    stage('Build') {
      // Run the gradle build
      gbuild2 'clean build -x test'
    }
    stage ('Test') {
      // execute required unit tests in parallel
           parallel (
        worker2: { node ('worker_node2'){
          // always run with a new workspace
          cleanupWs()
           unstash 'test-sources'
           gbuild2 '-D test.single=TestExample1 test'
        }},
        worker3: { node ('worker_node3'){
          // always run with a new workspace
          cleanupWs()
          unstash 'test-sources'
          gbuild2 '-D test.single=TestExample2 test'
        }},
      )
    }
  }
  catch (err) {
    echo "Caught: ${err}"
  }
  stage ('Notify') {
    // mailUser('<your email address>', "Finished")
  }
}
```

9. **Save** your changes to the file, **stage, commit,** and **push** it.

**<Save changes to Jenkinsfile>**

**git add Jenkinsfile**

**git commit -m “updated for stashing”**

```
git push
```

10. Now, go back and **Build Now** the **test** project. In the **Console Output**, you should be able to see the two parallel tests being executed. Eventually one will succeed and and one will fail (with a legitimate test failure).

```
…
[Pipeline] }
[Pipeline] // stage
[Pipeline] stage
[Pipeline] { (Test)
[Pipeline] parallel
[Pipeline] [worker2] { (Branch: worker2)
[Pipeline] [worker3] { (Branch: worker3)
[Pipeline] [worker2] node
[worker2] Running on worker_node2 in /home/jenkins2/worker_node2/workspace/demo-all_test-ZQC7KCTZRKI2O5V23U5SXILJOWC6TUBFMWH2SZ3ND747GW7SHEKQ
[Pipeline] [worker3] node
[Pipeline] [worker2] {
[worker3] Running on worker_node3 in /home/jenkins2/worker_node3/workspace/demo-all_test-ZQC7KCTZRKI2O5V23U5SXILJOWC6TUBFMWH2SZ3ND747GW7SHEKQ
…
[worker2]
[worker2]
[worker2] BUILD SUCCESSFUL
[worker2]
[worker2] Total time: 26.408 secs
[Pipeline] [worker2] }
[Pipeline] [worker2] // node
[Pipeline] [worker2] }
 [worker3] :compileJava UP-TO-DATE
[worker3] :processResources UP-TO-DATE
[worker3] :classes UP-TO-DATE
[worker3] :compileTestJava
[worker3] :processTestResources UP-TO-DATE
[worker3] :testClasses
[worker3] :test
[worker3]
[worker3]
[worker3] TestExample2 > example2 FAILED
[worker3]     org.junit.ComparisonFailure at TestExample2.java:10
[worker3]
[worker3] 1 test completed, 1 failed
[worker3] :test FAILED
…
[worker3]
[worker3] FAILURE: Build failed with an exception.
[worker3]
[worker3] * What went wrong:
[worker3] Execution failed for task ':test'.
[worker3] > There were failing tests.
```



=====================================================================================

**END OF LAB**

==========================================================================================

**Lab 3 - Creating a Declarative Pipeline**

**Purpose: In this lab, we'll see how to convert a scripted pipeline to a declarative pipeline.**

1. To start to get a handle on how declarative pipelines are structured, we're going to convert the scripted pipeline in the **Jenkinsfile** from Lab 7 into a declarative pipeline. To begin the process, we'll create a new branch to work in. (This will also be incorporated into our multibranch pipeline later.)

Switch back to the terminal session. You should be in the **/home/diyuser2/gradle-demo** directory. Now we want to create a new branch in our project so we can separate the declarative Jenkinsfile from the scripted one in branch **test.**

Run the following command to create the new branch and switch to it.

**$ git checkout -b decl test**

2. Now we'll edit the **Jenkinsfile** in this branch and convert it from a scripted pipeline to a declarative pipeline. Open the file in an editor.

**$ gedit Jenkinsfile**

3. We need to first make some changes in the beginning section of the pipeline script. We will keep the **#!groovy** as an indicator that this is a Jenkins script. But we need to wrap everything in our pipeline in a **pipeline** closure. Then instead of the **node** definition here, we need an **agent** specification. We can also remove the **Library** block as we'll do this in a different way in the declarative model.

Change these lines:

```
@Library('Utilities@1.5') _

node ('worker_node1') {
```

to

```
pipeline {

  agent{ label 'worker_node1'}
```

4. Next, let's add the load of our shared-library via the libraries directive. Put this directive under the agent declaration from step 3 (under the **agent {** line and before the **try {** line).

```
libraries {

    lib('Utilities@1.5')

  }
```

5. In a declarative pipeline, our collection of stages need to be wrapped in a **stages** closure. In our scripted pipeline, because we are using the Groovy **try-catch** mechanism to wrap all the stages, we can just replace that with the **stages** closure.

Change the

**try {**

statement to be

**stages {**

The top part of your Jenkinsfile should now look like this (changed lines in bold):

```
#!groovy
pipeline {
   agent{ label 'worker_node1'}
   libraries {
      lib('Utilities@1.5')
   }
   stages {
      stage('Source') {
```

6. In the declarative structure, each set of individual steps in a stage needs to be enclosed in a **steps** closure. Within each individual **stage** section (except for the Notify one), add a **steps** closure around the statements. Since we're not adding any other directives in the stages, you can just add "**steps {**" after each "**stage (…) {**" line and a closing bracket "**}**" at the end of each stage.

For example, after the changes, our first stage for **Source** would look like this (indention updated for clarity).

```
stage('Source') {
     steps {
        cleanWs()
        checkout scm
        stash name: 'test-sources', includes: 'build.gradle,src/test/'
     }
 }
```

Make sure to add the **steps {}** closure in the **Build** and **Test** stages as well!

7. Next, let's change the failing test in the 'Test' stage to one that will pass so we can see a successful run of the declarative pipeline. Change the line for

gbuild2 '-D test.single=TestExample**2** test' to

gbuild2 '-D test.single=TestExample**3** test'

8. Lastly, in declarative syntax, we have a **post** section that we can use to emulate the **post-build actions** of freestyle jobs. We're going to replace the scripted **catch** section with a declarative **post** section.

In the Jenkinsfile, erase/delete the **catch** and **notify** blocks from the pipeline. (Note there should still be a closing bracket after this from the original node closure we had. Leave that ending bracket.) The lines below are the ones to remove.

```
catch (err) {
  echo "Caught: ${err}"
}
stage ('Notify') {
 // mailUser('<your email address>', "Finished")
}
```

9. Now, add the section below at the place in the file where you just deleted the lines in the last step. Substitute in an email address where you can receive email for the "<your email address>" parts. Add these lines before the final closing bracket of the **pipeline** block.

```
post {
    always {
      echo "Build stage complete"
    }
    failure {
      echo "Build failed"
      mail body: 'build failed', subject: 'Build failed!', to: '<your email address>'
    }
    success {
      echo "Build succeeded"
      mail body: 'build succeeded', subject: 'Build Succeeded', to: '<your email address>'
    }
  }
```

**IMPORTANT**: Don't forget to put in a valid email address in the lines starting with "**mail**"!

9. After all of these changes, your pipeline in the Jenkinsfile should look like the one below, except that you should have your email address inserted for “<your email address>”. Verify and then **save** the file as **Jenkinsfile** in the directory.

```
#!groovy
pipeline {
    agent{ label 'worker_node1'}
    libraries {
      lib('Utilities@1.5')
   }
   stages {
     stage('Source') {
       steps {
         cleanWs()
         checkout scm
         stash name: 'test-sources', includes: 'build.gradle,src/test/'
      }
    }
    stage('Build') {
       // Run the gradle build
       steps {
          gbuild2 'clean build -x test'
       }
     }
     stage ('Test') {
        // execute required unit tests in parallel
        steps {
            parallel (
               worker2: { node ('worker_node2'){
                   // always run with a new workspace
                  cleanWs()
                  unstash 'test-sources'
                  gbuild2 '-D test.single=TestExample1 test'
               }},
              worker3: { node ('worker_node3'){
                 // always run with a new workspace
                cleanWs()
                unstash 'test-sources'
                gbuild2 '-D test.single=TestExample3 test'
               }},
           )
       }
     }
  } // end stages
  post {
      always {
        echo "Build stage complete"
      }
```

```
      failure{
        echo "Build failed"
        mail body: 'build failed', subject: 'Build failed!', to: '<your email address>'
      }
      success {
         echo "Build succeeded"
         mail body: 'build succeeded', subject: 'Build Succeeded', to: '<your email address>'
     }
   }
} // end pipeline
```

10. After saving the file, add, commit, and push the updated Jenkinsfile (in branch **decl**).

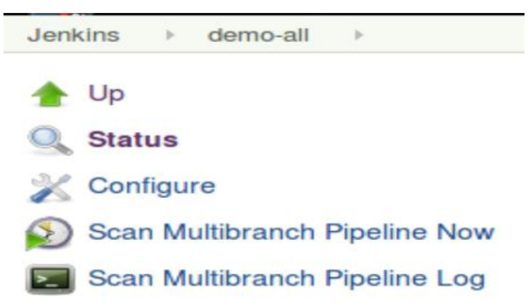
**$ git add Jenkinsfile**

**$ git commit -m "Add declarative Jenkinsfile"**

**$ git push origin decl**

11. We've now created a new branch in our **multibranch** setup. Now we want Jenkins to create a new project for it. Switch back to **Jenkins** and go into the **demo-all** project (if not already there).

Click on the "**Scan Multibranch Pipeline Now**" to generate a new job for the **decl** branch and run the script. It should succeed.





==========================================================================================

**END OF LAB**

==========================================================================================

**Lab 4 - Using Blue Ocean**

**Purpose: In this lab, we'll see how to use Jenkin's new Blue Ocean interface**

1. In Jenkins, go back to the dashboard (**localhost:8080**). Open the Blue Ocean interface either by going to **http://localhost:8080/blue** or by clicking the "**Open Blue Ocean**" menu item on the lefthand side of the page.

2. Once the main Blue Ocean page finishes loading, you should be on the **Pipelines** page. You can think of this page as being like the **dashboard** in the traditional interface.

It allows you to see a list of defined projects and their health (the weather icon) and success/failure status.

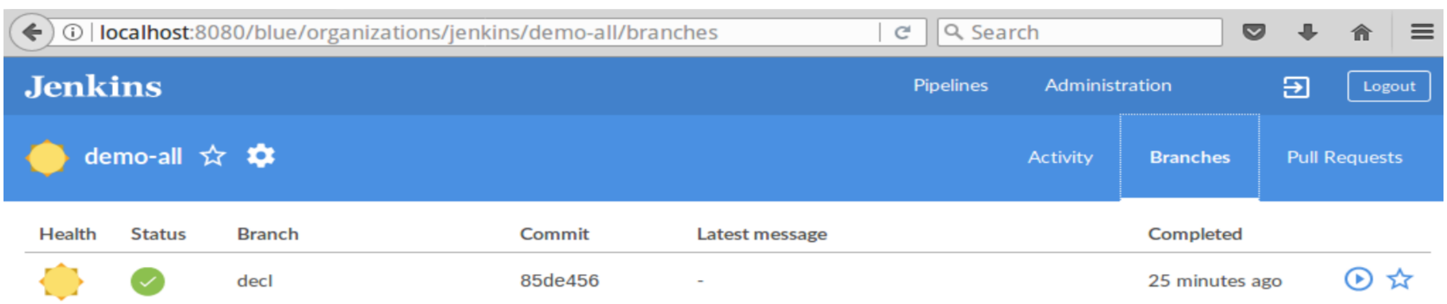
The **PR** column is for Pull Requests if we have any related to a project.

The star on the end is a way to add a favorite for quick reference at the top of the page. (Note: This only works if Jenkins can determine what the "default" item in the project should be. For project types with subitems, it may not be able to "favorite" them.) The star is a toggle. Click on some of the stars to see how the favorite action works. You can click on them again to remove something from the favorites area.

3. Since Blue Ocean is optimized for declarative pipelines, we'll take a closer look at our pipeline for the **decl** branch through it.

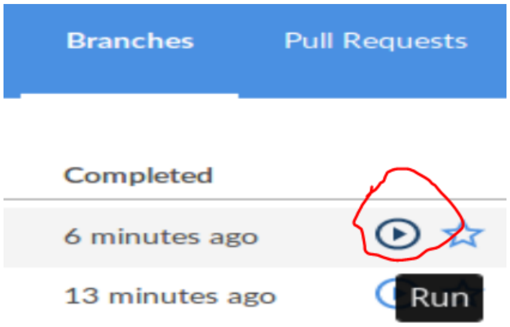
Click on the **demo-all** project. This takes you to the **Activity** page for **demo-all**. Here, you can see the set of runs related to this project. You can think of this like the page in the traditional interface that shows the runs for a project.

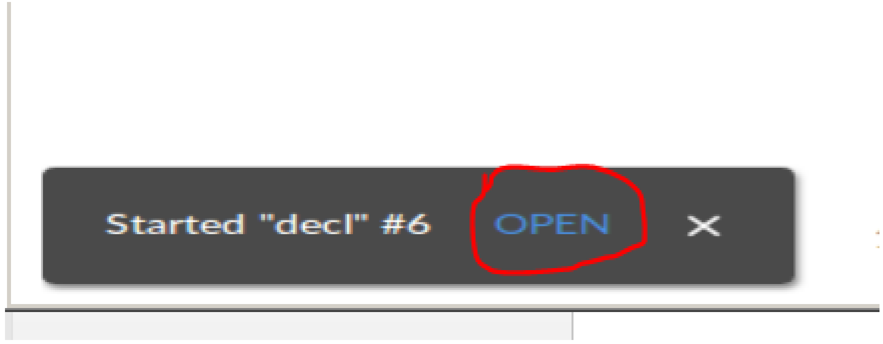
4. Now, in the row above the list, click on the **Branches** item. As expected, this shows the list of branches associated with this multi-branch pipeline. (This option (clicking on Branches ) only works for multibranch projects. )



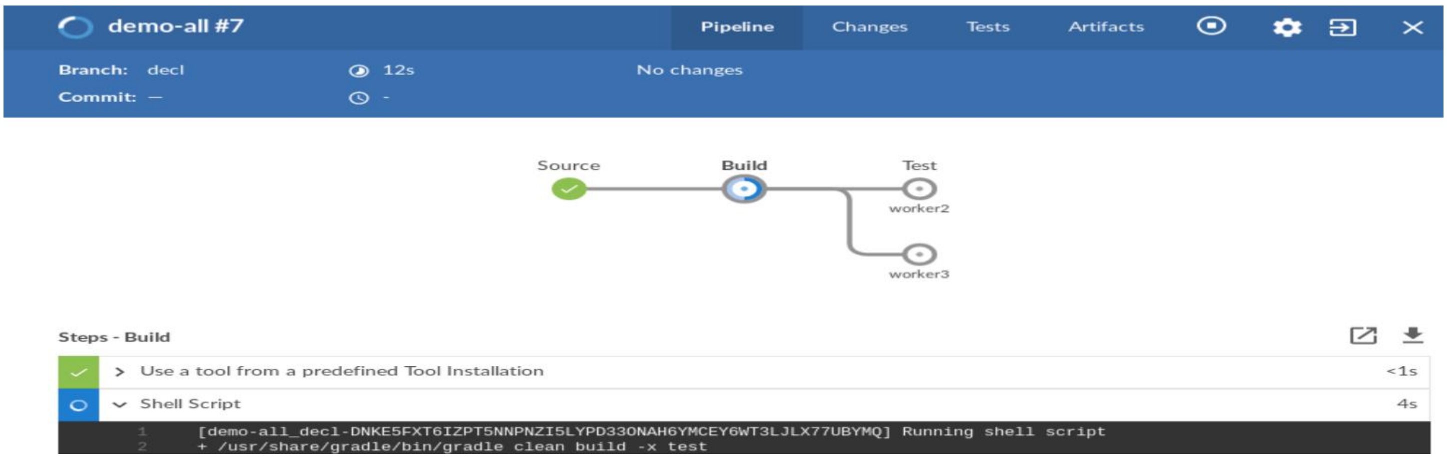

5. Find the row for our decl branch. Near the end of the row is a *circle with an arrow* in it. If you hover over that, you'll see that it says "**Run**". Click on that symbol.

A small box should pop up in the lower left corner with an "**OPEN**" link. Click on that (or click anywhere in the row for **decl**).



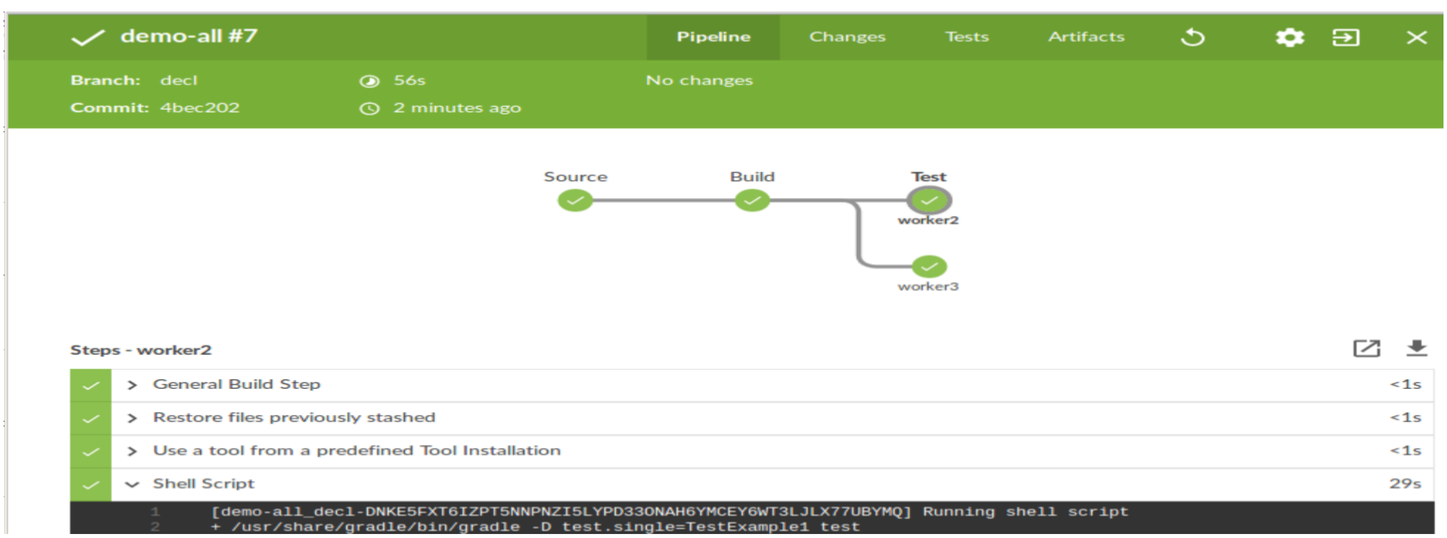




That will take you to the screen showing the pipeline executing. It will look something like the picture below.

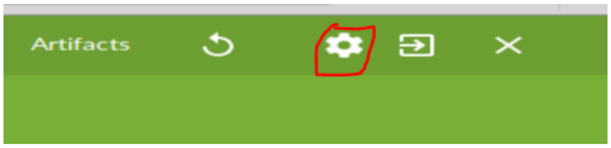



6. Notice the circle shapes representing each stage as it progresses, along with the running log at the bottom. When the pipeline completes the run, look at the logs for one of the stages. To do this, click on one of the circles representing a stage in the figure, and then click on the one of the steps listed in the bottom half of the page.

For example, click on the circle at the end labeled **worker2**. This is one of the parallel parts of our **Test** stage. Now, click on the **Shell Script** item in the bottom section (under **Steps - worker2**). Notice that Jenkins displays the part of the log for this part of the pipeline Clicking on Shell Script again collapses the log information.



7.  Find the icon that looks like a gear in the upper left section of the page.  That opens up a browser tab that puts you into the traditional **Configure** page for the project.  Click on the Gear icon now to try this out.

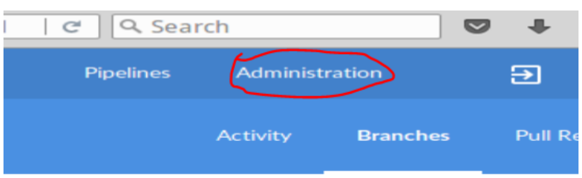



Note that this opens the configure page in a new tab.

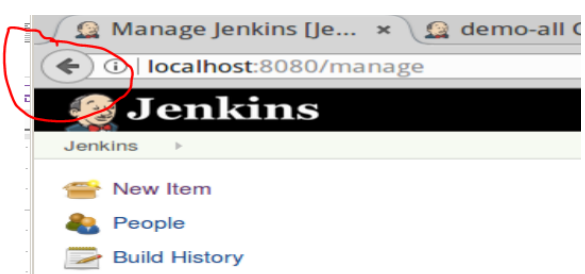
8. Switch back to the Blue Ocean tab for the decl job.  Exit the log of this run by clicking on the X in the upper right corner.   Then click Pipelines in the top row to get back to the Blue Ocean dashboard.


9.  Let's look at a couple of other features in Blue Ocean.   The **Administration** link is like the **Manage Jenkins** link in the classic interface.  Click on it to go to the **Manage Jenkins** page.

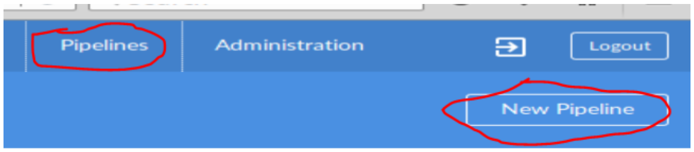
Click on the **Adminstration** link.




Then click on the back arrow in the browser to get back to the Blue Ocean interface.




9.  You should be back on the **Pipelines** page (http://localhost:8080/blue/pipelines).   (If not, click on the **Pipelines** link at the top of the page.)  Notice that there is a button here to create a **New Pipeline**.  Let's see how that works.  Click on the button.

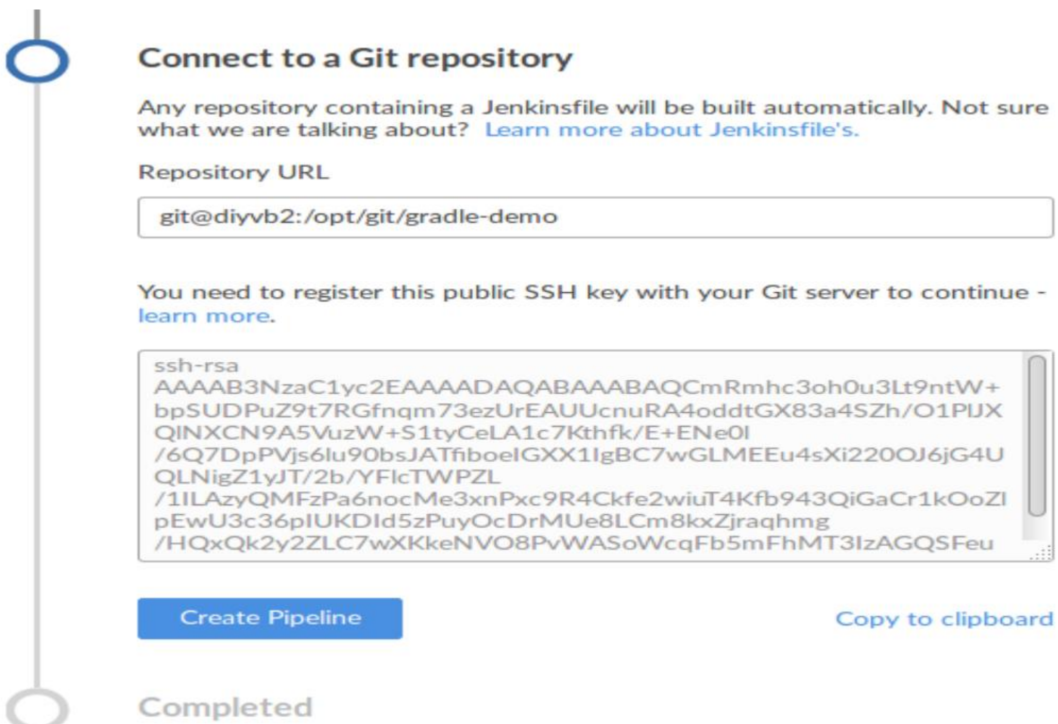



10. This page is essentially the "**New Item**" interface for a project stored in Git. The idea is you would create a **Jenkinsfile** in a project stored in Git and that becomes the basis for your new pipeline or you can create one through the interface's Pipeline Editor. You are first asked if your project is stored in Git, Github, BitBucket (or their enterprise versions). We will just reference one on our local system to see how this is done. Select the **Git** option.

11. For the **Repository URL**, we'll just enter the location of the same Git repository we've been using.

Enter **git@diyvb2:/opt/git/gradle-demo** in the **Repository URL** field.

12. Because of the URL, Jenkins will prompt you to "register" an SSH key (as shown below). This key needs to be added as an authorized key in the authorized_key file under /home/git/.ssh.




To get this added to the file, do the following:

Click on the "**Copy to clipboard**" link.

In a terminal, edit the file

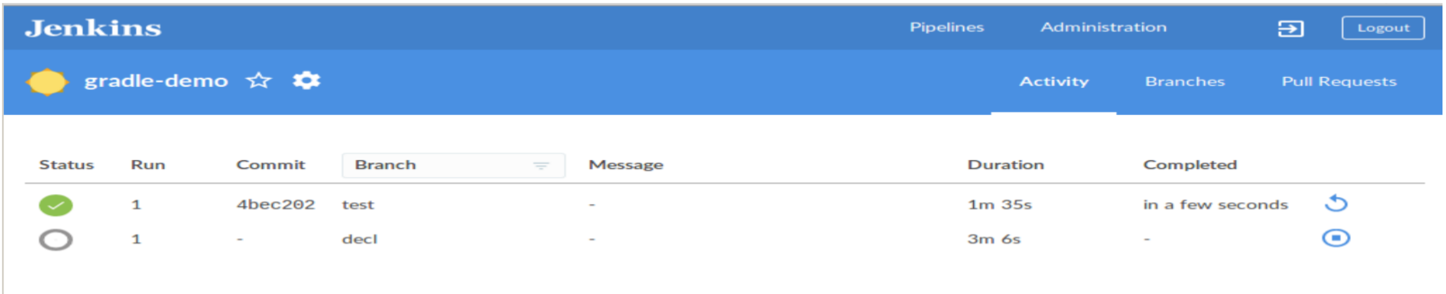
**sudo gedit /home/git/.ssh/authorized_keys**

When the file opens up in the editor, paste the copied key into the file.

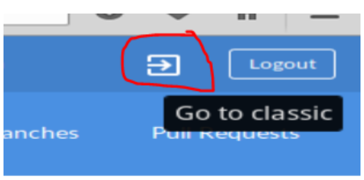
Save the file and exit the editor.

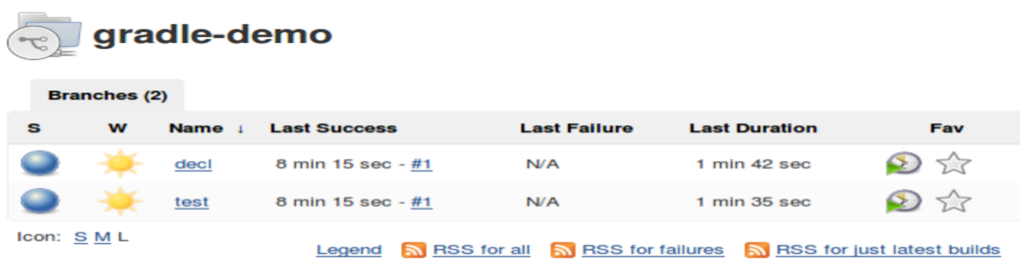
13. Click on the **Create Pipeline** button to create the new pipeline.

14. Jenkins will setup a pipeline for the projects/branches that it finds with a **Jenkinsfile** in it. Then it will start a build for all those projects/branches (eventually).




15. Click on the symbol with the arrow in the square (next to **Logout**) to go back to the "classic" interface. You'll see the classic interface of the pipeline that was created.




==========================================================================================

**END OF LAB**

==========================================================================================

**Lab 5 - Integrating Docker**

**Purpose: In this lab, we'll see how to use Docker in our pipeline. For simplicity, we'll start with a copy of our declarative Jenkinsfile from the last lab and just update it to use Docker.**

1. In this lab, we'll work with Docker images in two ways. First, we'll create a custom Docker image containing Gradle to use in our Gradle operations. We'll create this image using an **agent** specification that takes a **Dockerfile**. We have a **Dockerfile** already on disk. Take a look at it with the following command in a terminal session.

**$ cat /home/diyuser2/docker/Dockerfile**

2. For some parts of the pipeline, we'll use a ready-made version of this image already existing on your disk. To see it, in a terminal session, run this command:

**$ docker images**

The image named **gradle** with tag **3.4.1** is the one that contains Gradle for us to use.

3. To start working on this version separately, we need to create a separate branch in our **demo-all** project with its own copy of the **Jenkinsfile**. To do this create a new branch based off branch decl in our gradle-demo project. Make sure you are in the **gradle-demo** project directory in a terminal session. Then create the new branch and edit the **Jenkinsfile**.

**$ git checkout -b docker decl**

**$ gedit Jenkinsfile**

4. In the **Jenkinsfile**, we need to make a couple of changes for using Docker. First, since we are going to be using Docker images instead of library routines to run Gradle, we can remove the **libraries** directive and information. <u>Delete</u> the lines below from your **Jenkinsfile**.

```
libraries {
    lib('Utilities@1.5')
  }
```

5. Now, in the **Build** stage, we'll create a new Docker agent to use for the build. This **agent** will be created using the Dockerfile we looked at earlier. Add the lines shown in bold below to tell Jenkins to construct the Docker container built from the Dockerfile. Note that they are after the **Build** stage closure - but before the **steps** closure.

```
stage('Build') {
    agent {
        dockerfile {
            dir '/home/diyuser2/docker'
         }
     }
     steps {
          // Run the gradle build
```

6. This will do several things for us. It will create an image with Gradle in it - based on the statements in the Dockerfile. Then it will start a Docker container that runs the image. Next, it will map our workspace into the container (assuming it has access). And, then it will run any steps starting with "**sh**" (shell) inside the context of the container.

7. Since we will be running our Gradle build command <u>inside</u> the Docker agent's container and <u>not through the library</u> anymore, we need to change our build command. In the **Build** stage, in the **steps** block, change the line

**gbuild2 'clean build -x test'**

to be

**sh 'gradle clean build -x test'**

8. Now, in the **Test** stage, we will update the existing steps to just use a predefined instance of our image. As a reminder, that image is named "**gradle**" with a Docker tag of "**3.4.1**" (indicating the version). So we can refer to the image by the identifier "**gradle:3.4.1**".

Within the **parallel** step in the **Test** stage, the mappings for **worker2** and **worker3** already have **nodes** specified. We cannot create multiple agents in a step, so we'll use another approach. We'll wrap our build command in each parallel part in a "**withDockerContainer**" block. **withDockerContainer** takes the name of an image to run as an argument. We will use the predefined **gradle:3.4.1** image for this.

Add a **withDockerContainer** closure around each build command in the **Test** stage. This means add the lines in bold below.

```
worker2: { node ('worker_node2') {
        // always run with a new workspace
        step([$class: 'WsCleanup'])
        unstash 'test-sources'
       withDockerContainer('gradle:3.4.1') {
             gbuild2 ' -D test.single=TestExample1 test'
       }
     }} ,
worker3: { node ('worker_node3'){
      // always run with a new workspace
      step([$class: 'WsCleanup'])
     unstash 'test-sources'
      withDockerContainer('gradle:3.4.1') {
            gbuild2 '-D test.single=TestExample3 test'
      }
    }},
```

9. Also, we'll need to change the gbuild2 commands to the same kind of sh commands we used above. This is because we removed the library reference that had the gbuild2 command, and are replacing that with Gradle running in the Docker container. Change the start of the build lines in the Test stage from:

**gbuild2 '-D test.single**

to

**sh 'gradle -D test.single**

10. After these changes, the main part of your **Test** stage should look like this (changed or added lines are in bold):

```
worker2: { node ('worker_node2') {
        cleanupWs()
        unstash 'test-sources'
       withDockerContainer('gradle:3.4.1') {
             sh 'gradle -D test.single=TestExample1 test'
       }
```

```
    }} ,
  worker3: { node ('worker_node3'){
      // always run with a new workspace
      cleanupWs()
      unstash 'test-sources'
      withDockerContainer('gradle:3.4.1') {
           sh 'gradle -D test.single=TestExample3 test'
      }
    }},
```

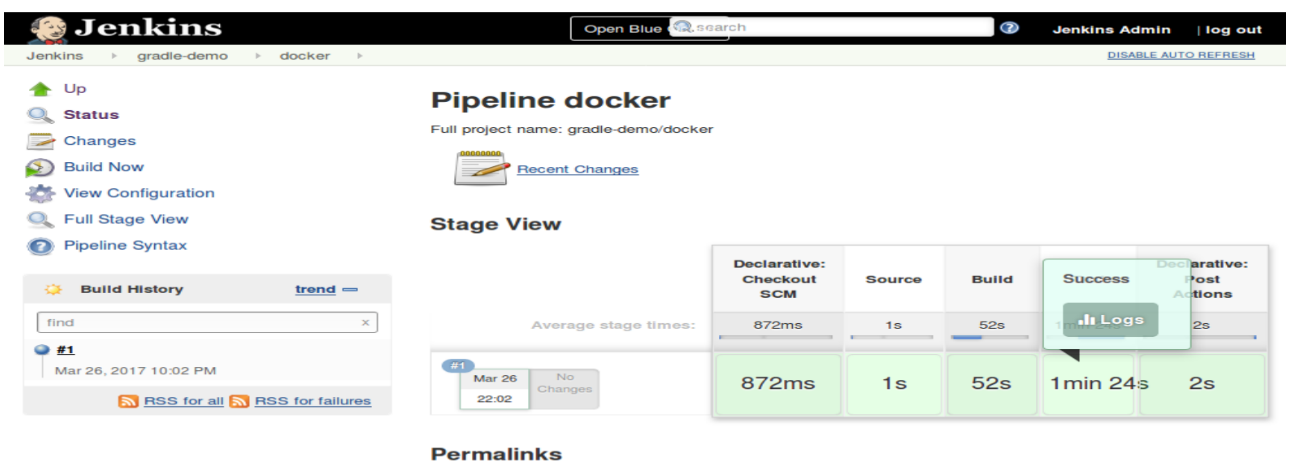
11. Save your Jenkinsfile.

12. Now add, commit, and push the updated Jenkinsfile (in branch docker).

**$ git add Jenkinsfile**
**$ git commit -am "Update to use Docker"**
**$ git push origin docker**

13. We've created a new branch in our multibranch setup. Now we want Jenkins to create a new project for it. Switch back to **Jenkins.** We can use the **gradle-demo** multibranch project that got created in our last lab through Blue Ocean. Switch to that project (if not already there).

Click on the "**Scan Multibranch Pipeline Now**" to generate a new job for the **docker** branch and run the script. It should succeed.




14. Take a look at the console output for the run to see the construction of the Docker image and use of Docker in the **Test** stage.

==========================================================================================

**END OF LAB**

=========================================================================================

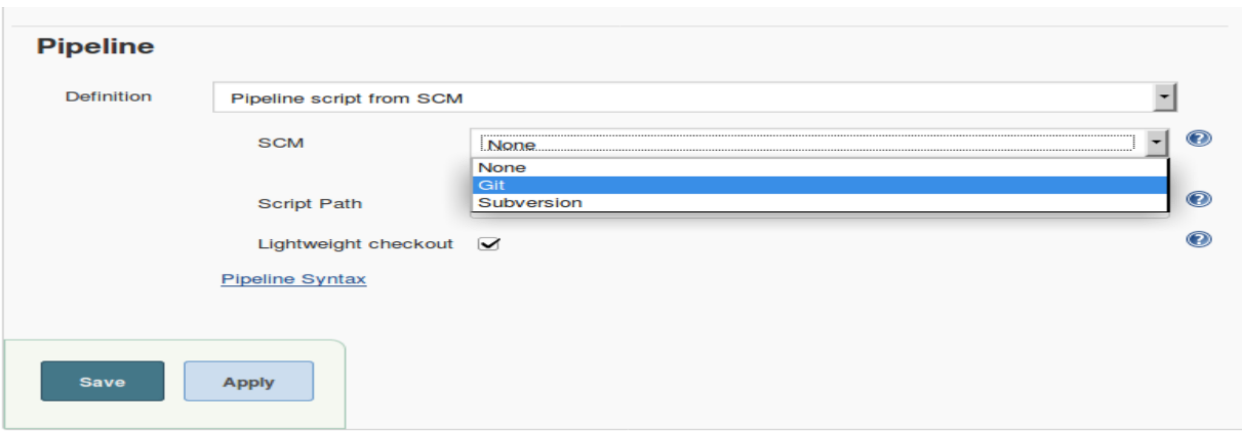
**OPTIONAL Lab 6 - Including a Jenkinsfile back into a native Jenkins project**

**Purpose: In this optional lab, we'll see how to take a Jenkinsfile and include it in a native Jenkins pipeline project. We'll use the Docker project we just created in the previous lab as our example.**

1. In Jenkins, create a new project from the dashboard. (Switch back to the dashboard at http://localhost:8080 and click on the **New Item** menu item.) Select **Pipeline** for type of project. Name it whatever you like. Once you are on the configuration page, scroll down to the text entry block for the code.

2. On the **Pipeline** section of the page, you'll see a field called **Definition**. This will have **Pipeline script** in it by default. At the end of that field is an arrow to select entries from a list. Click on that arrow and select **Pipeline script from SCM**.

3. You'll now see a new field named **SCM**. Select **Git** from the list in that field.




4. Once you select Git, additional fields will show up on the screen.

Fill in the **Repository URL** field with **git@diyvb2:/opt/git/gradle-demo.git**

Fill in the **Branches** field with **docker**

5. You can leave everything else as-is. Now **Save** the changes and select **Build Now** to build the pipeline.

=========================================================================================

**END OF LAB**

=========================================================================================