

Jenkins 2.0: Pipeline-as-code, Declarative Pipelines, and Blue Ocean

Revision 1.3 - 12/01/16

Brent Laster

Lab 1 - Creating a Simple Pipeline Script

Purpose: In this first lab, we'll start with a sample pipeline script and change it to work for our setup to illustrate some basic pipeline concepts.

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. Log in to Jenkins by clicking on the **"log in"** link in the upper right corner. 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 lower section of the screen, and find and click on **"Manage Nodes"**.



Jenkins CLI

Access/manage Jenkins from your shell, or from your script.



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.

4. On the next screen, notice that you already have nodes (formerly known as slaves) named **"worker_node1"** and **"worker_node2"**. We'll use one to run our processes on. Note that you wouldn't normally have a node defined on the same system as the master, but we are using this setup to simplify things for the workshop.

S	Name	Architecture	Clock Difference	Free Disk Space	Free Swap Space	Free Temp Space	Response Time
	master	Linux (amd64)	In sync	1.12 GB	3.99 GB	1.12 GB	0ms
	worker_node1	Linux (amd64)	In sync	1.12 GB	3.99 GB	1.12 GB	1801ms
	worker_node2	Linux (amd64)	In sync	1.12 GB	3.99 GB	1.12 GB	1829ms
Data obtained		0.49 sec	0.46 sec	0.48 sec	0.47 sec	0.47 sec	0.44 sec

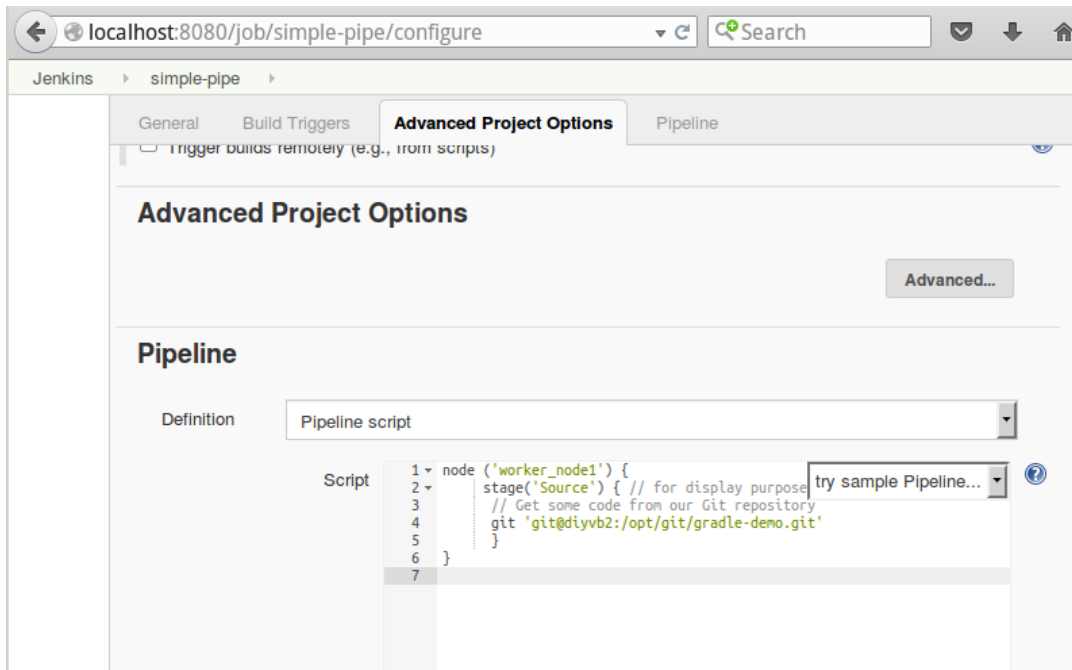
5. Click on “**Back to Dashboard**” (upper left) to get back to the Jenkins Dashboard. Now we’ll create our first pipeline project. In the left column, click on “**New Item**”. Notice that there are quite a few different types of items that we can select here. Type a name into the “**Enter an item name**” field. As a suggestion, you can use “**simple-pipe**”. Then choose the “**Pipeline**” project type and click on the “**OK**” button to create the new project.

6. Once Jenkins finishes loading the page for our new item, scroll down and find the **Pipeline** section. Leave the **Definition** selection as “**Pipeline script**”. Now, we’ll create a simple script to pull down a Git repository already installed on the image. We’ll create a **node** block to have it run on “**worker_node1**”, and then a **stage** block within that to do the actual source management command. The Pipeline DSL here provides a “**git**” command that we can leverage for that.

Inside the input area for the **Script** section, type (or paste) the code below. (You can leave out the comments if you prefer.)

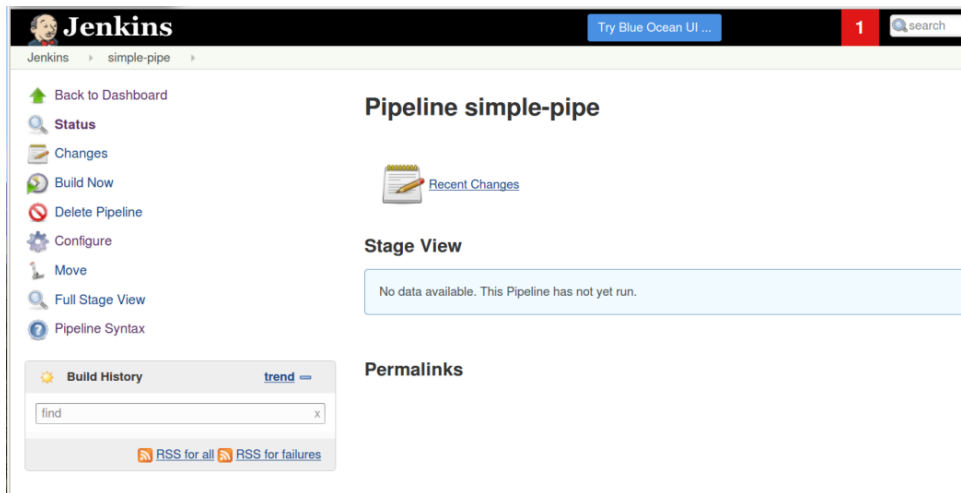
```
node('worker_node1') {
    stage('Source') { // for display purposes
        // Get some code from our Git repository
        git 'git@diylvb2:/opt/git/gradle-demo.git'
    }
}
```

This is what it will look like afterwards.

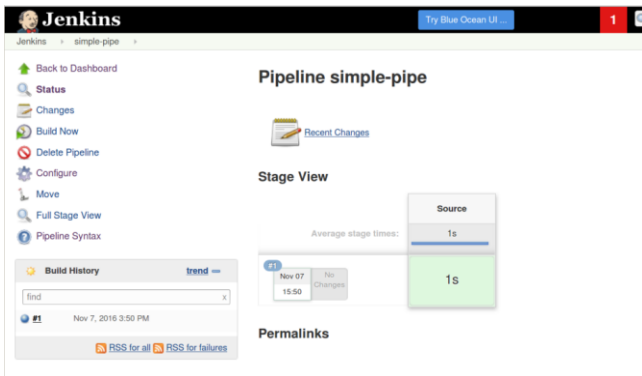


Now, click the **Save** button to save your changes.

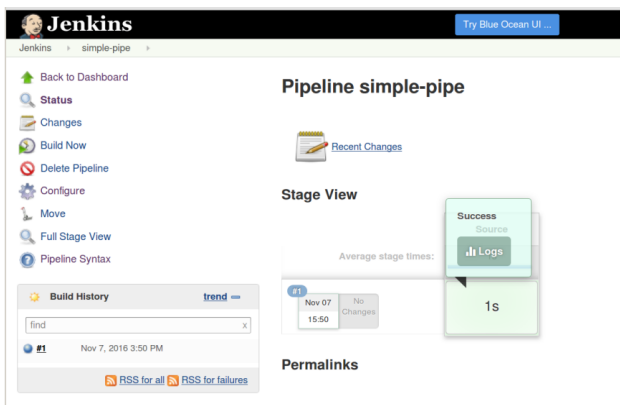
7. You'll now be on the job view for your new job. Notice the reference to the **"Stage View"**. Since we haven't run the job yet, there's nothing to show here, as the message indicates.



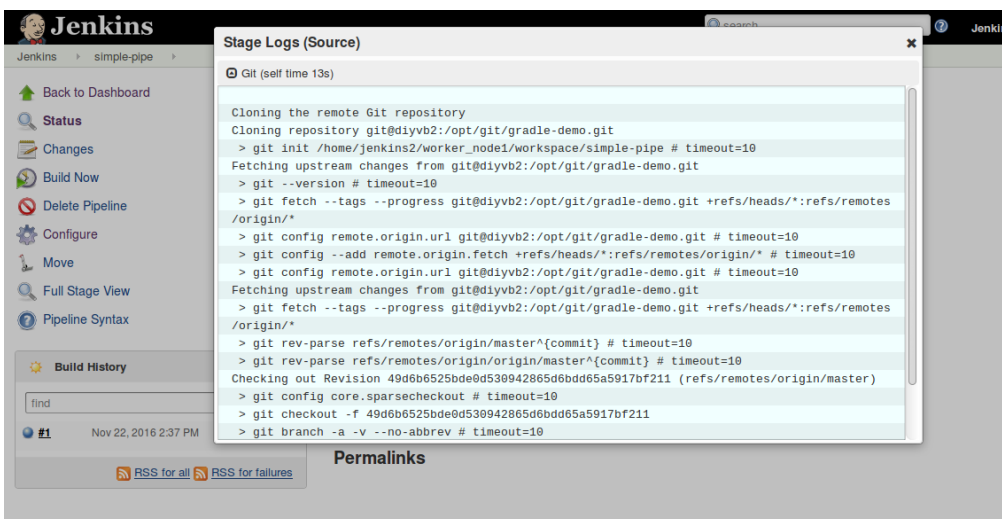
8. Click on the **"Build Now"** link to build your job. (It may take a moment to startup and run.) After it runs, notice that you now see a **Stage View** that is more informative. The **"Source"** name is what we had in our stage definition. We also have the time the stage took and the green block indicating success.



9. Let's look at the logs for this stage via this page. Hover over the green box for the **Source** stage until you see another box pop up with a smaller box named **"Logs"**.



10. Now, click on that smaller **"Logs"** box inside the **"Success"** box to see the log for the run of the stage pop up.
(Of course, you could also use the traditional Console Output route to see the logs.)



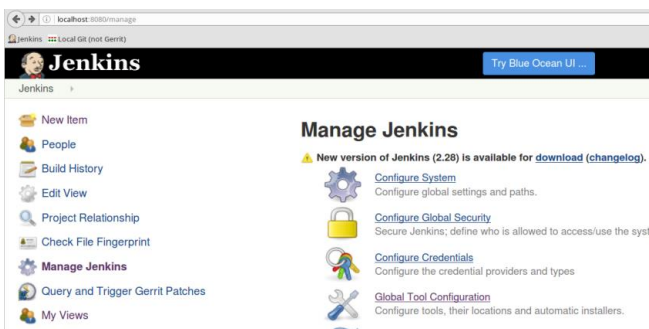
11. Close the popup window for the logs and click on **Back to Dashboard** to be ready for the next lab.

END OF LAB

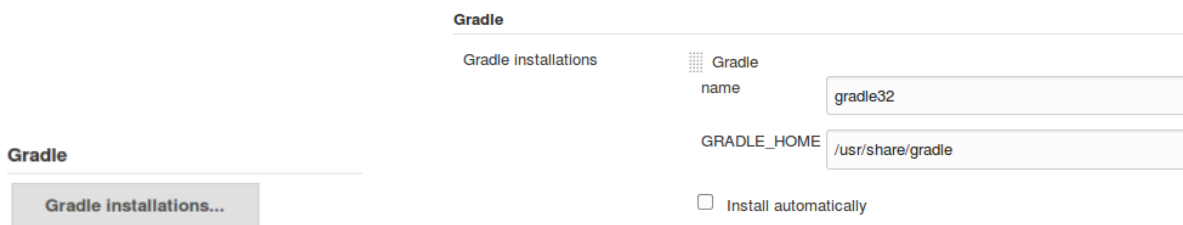
Lab 2 - Adding a Build Stage

Purpose: In this lab, we'll add a new stage to our pipeline project to build the code we download. We'll also see how to use the Retry functionality and include tools defined globally in Jenkins.

1. For this lab, we'll add a stage to use Gradle to build our **gradle-demo** project. We already have Gradle installed and configured on this Jenkins instance. So we just need to understand how to reference it in our pipeline script.
2. You should be back on the **Jenkins Dashboard** (home page) after the last lab. To see the Gradle configuration, click on **Manage Jenkins** and then select **Global Tool Configuration** in the list.



3. On the **Global Tool Configurations** page, scroll down and find the **Gradle** section. Click on the “**Gradle Installations**” button. Notice that we have our local Gradle instance named as “gradle32” here. We'll need to use that name in our pipeline script to refer to it.



4. Switch back to the configuration for your simple-pipe job - either by going to the Dashboard, then selecting the job, and then selecting **Configure** or just entering the URL: **http://localhost:8080/view/All/job/simple-pipe/configure**.
5. Scroll down to the pipeline script input area again and add the lines in bold below to your job. This sets us up to use the Gradle tool that we have installed. (Notice that the closing brace for the node has to come after the Build stage definition.)

```
node('worker_node1') {  
    © 2016 Brent Laster
```

```

def gradleHome

stage('Source') { // for display purposes
    // Get some code from our Git repository
    git 'git@diyvb2:/opt/git/gradle-demo.git'
}

stage('Build') {
    // Run the gradle build
    gradleHome = tool 'gradle32'
}
}

```

6. Now that we have the stage and have told Jenkins how to find the gradle installation that we want to use, we just need to add the command to actually run gradle and do the build. That command is effectively just **gradle** followed by a list of tasks to run.

We'll use a shell command to run this. The **gradleHome** value we have is what's configured for that value - the home path for Gradle. To get to the executable, we'll need to add on "**bin/gradle**". Then we'll add in the gradle tasks that we want to run. To make this all work, add in the line in bold below in the **Build** stage. Note the use of the single quotes and the double quotes.

```

stage('Build') {
    // Run the gradle build
    gradleHome = tool 'gradle32'
    sh "'${gradleHome}/bin/gradle' clean buildAll"
}

```

7. Now, **Save** your changes and click on the "**Build Now**" link in the left column. Wait for the run to complete. (This will take a while for the gradle part.) Notice the **Stage View** now, and the color of the boxes. The "pink" indicates a successful stage in an overall failure. The pink and red stripe indicates a failures (as does the "failed" note in the bottom right corner).

Stage View

	Source	Build
#2 Nov 07 16:46 No Changes	2s	51s failed
#1 Nov 07 15:50 No Changes	1s	

Permalinks

- Last build (#2), 8 min 55 sec ago
- Last stable build (#1), 1 hr 5 min ago
- Last successful build (#1), 1 hr 5 min ago
- Last failed build (#2), 8 min 55 sec ago
- Last unsuccessful build (#2), 8 min 55 sec ago
- Last completed build (#2), 8 min 55 sec ago

9. Hover over each box for the stages in the most recent run (#2) to see the status information for that stage.

Stage View

Stage View

Failed with the following error(s)

Shell Script script returned exit code 1

See stage logs for more detail.

10. While hovering over the **Build** stage box, notice the message telling you which step failed (“**Shell Script**”). Click on the **Logs** link to bring those up. In the pop-up box, you’ll see a description of the steps involved in this stage. Click on the small down arrow in the square next to “**Shell Script**” to expand the log information. What is the error message?

Stage Logs (Build)

Stage View

Stage Logs (Build)

FAILURE: Build failed with an exception.

* What went wrong:

Task 'buildAll' not found in root project 'simple-pipe'. Some candidates are: 'build'.

* Try:

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.

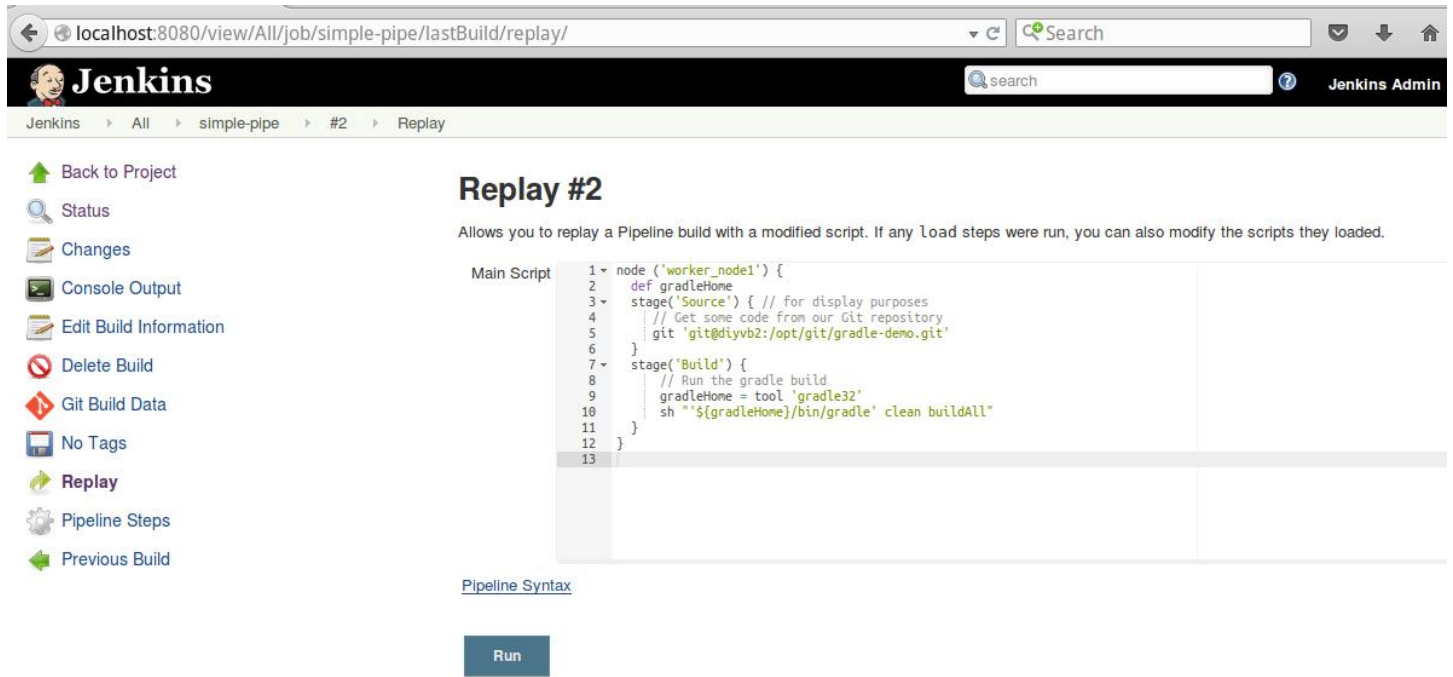
BUILD FAILED

Total time: 31.497 secs

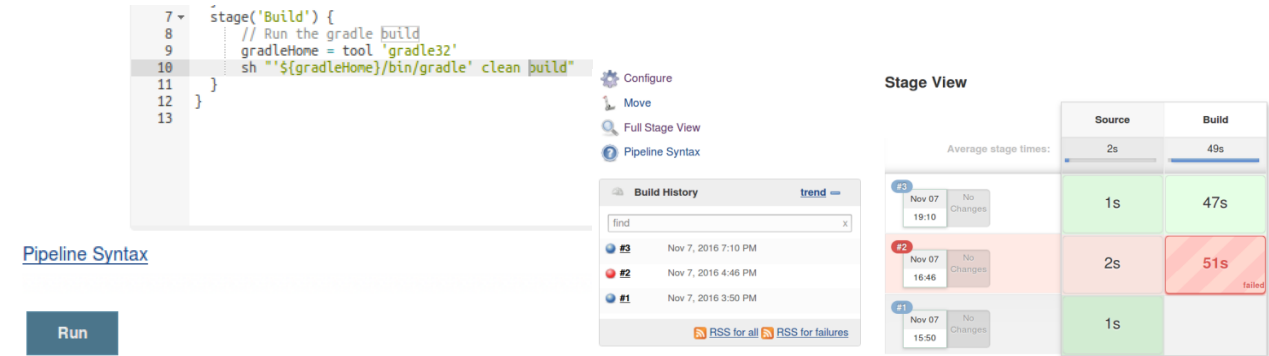
11. The log shows the problem - there is not a “**buildAll**” task. The reason is that, for Gradle, the task should be “**build**”. Let’s fix that. Use the escape key or X in the corner to close the **Stage Logs (Build)** window.

12. Before we change our script though, let’s use the **Replay** functionality to test this change out. To use this, we’ll need to get to the screen for the specific build that failed. In the “**Build History**” window, click on the red ball next to the latest build, or click on the “**Last build (#2)**” link in the **Permalinks** section.

13. In the left column, click on the **Replay** menu item. After a moment, the replay screen will appear with a copy of the pipeline script as it is for this build.



14. On the line that has the gradle command and the “**buildAll**” task, change “**buildAll**” to just “**build**”. Then click the “**Run**” button. Notice that this kicks off another build, which should complete successfully after a few moments.



15. Click on the **Configure** menu item on the left. Scroll down to the **Pipeline** section and look at the code in the **Script** window. Notice that it still has the old code in it (“**buildAll**”). The **Replay** allowed us to try something out without changing the code. Edit the same line in this window as in the previous step to change “**buildAll**” to just “**build**”. **Save** your changes and then **Build Now** (left menu). This build (#4) should complete successfully after a few moments.

=====

END OF LAB

Lab 3 - Using Shared Libraries

Purpose: In this lab, we'll replace our use of the build command with code defined in a shared library.

1. For the workshop, we have defined some shared libraries and functions in a “shared-libraries” Git repository. We have one named “Utilities” that we'll use to replace our build step here. You can see the definition of it in the directory `~/shared-libraries/src/org/conf/Utilities.groovy`. It takes a reference to our script and build arguments to pass to our gradle instance for a build.

```
package org.foo

class Utilities {

    static def gbuild(script, args) {

        script.sh "${script.tool 'gradle32'}/bin/gradle ${args}"

    }

}
```

2. To make this available to our pipeline scripts, we need to add it into our Jenkins system configuration. From the dashboard, click on **Manage Jenkins**, then **Configure System**.

3. Scroll down until you find the “**Global Pipeline Libraries**” section. Click the “**Add**” button.

Global Pipeline Libraries

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.

Add

4. Fill in the general fields in this section as follows:

For **Name**, you can just enter something like “**Utilities**”. This will be the name you refer to the library by in the script.

For **Default version**, enter a tagged version. In this case, the latest version is “**1.3**”, so enter that. Note that while a branch name can be used here, updates to that may not be picked up over time.

Check the “**Load implicitly**” box to tell Jenkins to automatically provide access to the library without having to use @Library to load it in the script.

Check the “**Allow default version to be overridden**” box to tell Jenkins to allow loading a different version of this library in your script if you specify the version in conjunction with the @Library directive. (i.e. @Library('libname@version'))

For **Retrieval method**, **Legacy SCM** should be indicated. This is basically allowing pulling the library from source control via any standard Jenkins SCM plugin. To ensure we get the desired version, we'll include the variable string `${library.LIBNAME.version}` in the SCM configuration.

Library Name: Utilities

Default version: 1.3

Cannot validate default version until after saving and reconfiguring.

Load implicitly: ☒

Allow default version to be overridden: ☒

Retrieval method

☐ Modern SCM

☒ Legacy SCM

5. We'll use **Git** as our **Source Code Management** system to include the library since it is stored in a Git repository. So we'll need to select Git in that section and configure it to point to that repository.

Fill in the "**Repository URL**" section with the url "**git@diylvb2:/opt/git/shared-libraries**". Click the **Advanced** button and for Refspec, enter the variable string for the specific version "**\${library.Utilities.version}**".

Finally, for "Branches to build", we'll use the fully qualified tag specifier to be precise "**refs/tags/1.3**".

Source Code Management

☒ Git

Repositories

Repository URL: git@diylvb2:/opt/git/shared-libraries

Credentials: - none -

Name:

Refspec: \${library.Utilities.version}

Add Repository

Branches to build

Branch Specifier (blank for 'any'): refs/tags/1.3

6. **Save** your changes to the system configuration.

7. Now, go back to your **simple-pipe** job and select the **Configure** item. Modify the pipeline script in your job to import the method from our library and use the new call to build the code. (See listing below.) Afterwards, your pipeline script should look like below - note the added/changed lines in bold. (Lines referencing gradleHome are no longer needed.)

```
import static org.conf.Utilities.*

node('worker_node1') {
    def gradleHome
    stage('Source') { // for display purposes
        // Get some code from our Git repository
        git 'git@diylvb2:/opt/git/gradle-demo.git'
    }
    stage('Build') {
```

```

        // Run the gradle build
        gbuild this, 'clean build'
    }
}

```

Pipeline script

Script

```

1  import static org.conf.Utilities.*
2  node ('worker_node1') {
3      stage('Source') { // for display purposes
4          // Get some code from our Git repository
5          git 'git@diyvb2:/opt/git/gradle-demo.git'
6      }
7      stage('Build') {
8          // Run the gradle build
9          gbuild this, 'clean build'
10     }
11 }
12

```

8. **Save** your changes and **Build Now**.

=====

END OF LAB

=====

Lab 4 - Loading Groovy Code Directly, User Inputs, Timeouts, and the Snippet Generator

Purpose: In this lab, we'll see how to load Groovy code directly into our pipeline scripts, how to work with user inputs and timeouts and how to use the built-in Snippet Generator to help construct Groovy code for our script.

1. We can also load groovy code directly and use it in our scripts. There is a file in the `~/shared-libraries/src` area named **verify.groovy** that has these contents:

```

def call(String message) {
    input "${message}"
}

```

It allows us to wait on user input to proceed. Let's add it to our script.

2. We'll wrap this in a new **stage** section. Add the following code **after** the 'build' stage, but **before** the closing bracket of the node definition.

```

stage ('Verify') {
    // Now load 'verify.groovy'.
    def verifyCall = load("/home/diyuser2/shared-libraries/src/verify.groovy")
}

```

```

        verifyCall("Please Verify the build")
    }

```

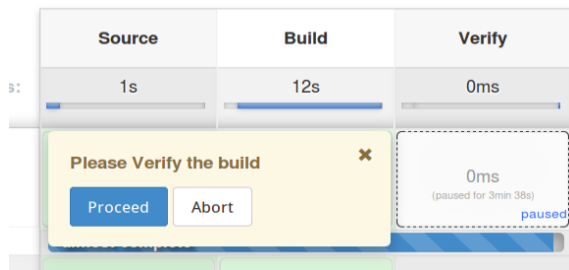
```

1  import static org.conf.Utilities.*
2  node ('worker_node1') {
3      stage('Source') { // for display purposes
4          // Get some code from our Git repository
5          git 'git@diyvb2:/opt/git/gradle-demo.git'
6      }
7      stage('Build') {
8          // Run the gradle build
9          gbuild this, 'clean build'
10     }
11     stage ('Verify') {
12         // Now load 'verify.groovy'.
13         def verifyCall = load("/home/diyuser2/shared-libraries/src/verify.groovy")
14         verifyCall("Please Verify the build")
15     }
16 }

```

3. **Save** your changes and **Build Now**. You will now have a new stage **Verify** that shows up in the stage output view.

When the build reaches this stage, if you hover over the box in the **Verify** section, you'll see the pop-up for you to tell it to **Proceed** or **Abort**.



4. As well, if you look at the console output, you'll see the process waiting for you to tell it to **proceed** or **abort** (via clicking on one of the links).

```

:test UP-TO-DATE
:check UP-TO-DATE
:build

```

BUILD SUCCESSFUL

```

Total time: 6.835 secs
[Pipeline] }
[Pipeline] // stage
[Pipeline] stage
[Pipeline] { (Verify)
[Pipeline] load
[Pipeline] { (/home/diyuser2/shared-libraries/src/verify.groovy)
[Pipeline] }
[Pipeline] // load
[Pipeline] input
Please Verify the build
Proceed or Abort

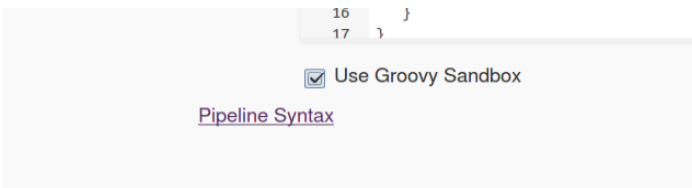
```



5. Go ahead and click the button or link to **Abort** the build. Note the failed status in the **Stage View**.

Source	Build	Verify
1s	13s	267ms
1s	23s	217ms (paused for 26s) failed

6. Let's add a timeout to make sure the build doesn't go on forever. To figure out the exact syntax, we'll use the **Snippet Generator** tool. Click on the **Configure** button for the job, then scroll to the bottom and click on the **Pipeline Syntax** link.



7. You'll now be in the Snippet Generator page. Here you can select what you want to do and then have Jenkins generate the Groovy pipeline code for you. To get our timeout code, select the following values:

In the **Sample Step dropdown**, select **"timeout: Enforce time limit"**

For the **Timeout** value, enter **5**,

For the **Unit value**, select **SECONDS**

8. Click on the **Generate Pipeline Script** button to see the resulting code.

Overview

This **Snippet Generator** will help you learn the Pipeline Script code which can be used to define various steps. Pick a step you are interested in from the list, configure it, click **Generate Pipeline Script**, and you will see a Pipeline Script statement that would call the step with that configuration. You may copy and paste the whole statement into your script, or pick up just the options you care about. (Most parameters are optional and can be omitted in your script, leaving them at default values.)

Steps

Sample Step: **timeout: Enforce time limit**

Timeout:

Unit: **SECONDS**

Generate Pipeline Script

Global Variables

There are many features of the Pipeline that are not steps. These are often exposed via global variables, which are not supported by the snippet generator. See the [Global Variables Reference](#) for details.

9. Copy and paste the **timeout** block lines around the `verifyCall` invocation in your pipeline script. You can use **Back->Configure** to get back to the script page.

```
        timeout(time: 5, unit: 'SECONDS') {  
            verifyCall("Please Verify the build")  
        }
```

```
11 ▾ stage ('Verify') {  
12     // Now load 'verify.groovy'.  
13     def verifyCall = load("/home/diyuser2/shared-libraries/src/verify.groovy")  
14 ▾     timeout(time: 5, unit: 'SECONDS') {  
15         verifyCall("Please Verify the build")  
16     }  
17 }  
18 }
```

10. **Save** your changes and **Build Now**. Let the job run to see the timeout pass and the job fail.

=====

END OF LAB

=====

Lab 5 - Using global functions and exception handling

Purpose: In this lab, we'll see how to reference global functions and one way to handle exception processing

1. For shared libraries, we can also define global functions under the “**vars**” subdirectory of our shared library repository. For use in this workshop, we have a function named “**mailUser**” defined in `~/shared-libraries/vars/mailUser.groovy`. The code in this function is:

```
def call(user, result) {  
    // Add mail message from snippet generator here  
    mail bcc: "", body: "The current build\'s result is ${result}.", cc: "", from: "", replyTo: "", subject: 'Build Status'  
}
```

2. As suggested by the comment, the basic text for the body of the function came from the **Snippet Generator**. As an optional exercise, you can go to the Snippet Generator (**see Lab 4**) and plug in the parts to create the basic “**mail**” line here. (We have changed some quoting and variables.)

3. In our pipeline script, we'll add a “**Notify**” stage to send a basic message when the build is done. (Jenkins is already configured globally to be able to send email.) Add the following stage at the end of your pipeline script **after** the **Verify** stage, but **before** the ending bracket for the node definition. (Substitute in whatever email address is appropriate.)

```
    } // end verify stage  
    stage ('Notify') {  
        mailUser(<email address in single quotes>,"Finished")  
    }
```

© 2016 Brent Laster

```

    }
} // end node

```

```

} // end Verify
stage ('Notify') {
    mailUser('<email address here>', "Finished")
}
}

```

4. **Save** and **Build Now**. Try it with selecting **Proceed** and also with selecting **Abort** or letting the timeout happen. (Note: It may be easier to set the timeout value to 10 seconds instead of 5 to give you time to find and click Proceed.)
5. Notice that unless we select **Proceed**, the **Notify** stage is never reached. We want to be able to send the notification no matter what. How do we do this? We can handle this by adding a **try/catch** block around the other stages to catch the error and still allow the **Notify** stage to proceed.
6. Add the **“try {“** line after the node definition and the **“catch”** block after the **“Verify”** stage (and before the **“Notify”** stage) as shown below.

```

node ('worker_node1') {
    try {
        stage('Source') { // for display purposes
            ...
            verifyCall("Please Verify the build")
        } // end timeout
    } // end verify
} // end try
catch (err) {
    echo "Caught: ${err}"
}
stage ('Notify')

```

```

1 import static org.conf.Utilities.*
2 node ('worker_node1') {
3     try {
4         stage('Source') { // for display purposes

```

```

16         verifyCall("Please Verify the build")
17     }
18 } // end try
19 catch (err) {
20     echo "Caught: ${err}"
21 }
22 stage ('Notify') {
23     mailUser('<email address here>', "Finished")
24 }
25 }
26

```

7. **Save** and **Build Now**. Let the process timeout and note that the mail is now sent regardless.

=====

END OF LAB

=====

Lab 6 - Multibranch projects

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

1. 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.
2. 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.
3. To see it, in 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
```

```
#!/groovy
import static org.foo.Utilities.*
node ('worker_node1') {
    // always run with a new workspace
    step([$class: 'WsCleanup'])
    try {
        stage('Source') {
            checkout scm
        }
        stage('Build') {
            // Run the gradle build
            gbuild this, 'clean build -x test'
        }
        stage ('Test') {
            // execute required unit tests in parallel
            |
        }
    }
    catch (err) {
        echo "Caught: ${err}"
    }
    stage ('Notify') {
        // mailUser('<your email address>', "Finished")
    }
}
```

4. 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 there.

Also notice the line “**step([\$class: ‘WsCleanup’])**”. 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.

5. 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**.




Multibranch Pipeline

Creates a set of Pipeline projects according to detected branches in one SCM repository.

6. 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**”.

Branch Sources

 **Git**

Project Repository

Credentials - none - Add

Ignore on push notifications ☐

Repository browser (Auto) ?

Additional Behaviours Add

Advanced...

Property strategy All branches get the same properties

Add property

Delete source

Add source

Build Configuration

Mode by Jenkinsfile

7. 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 will get those for free and so don't need to add anything here.
8. 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.

Getting remote branches...

Seen branch in repository origin/master

Seen branch in repository origin/test

Seen 2 remote branches

Checking branch master

Does not meet criteria

Checking branch test

Met criteria

Scheduled build for branch: test

Done.

Finished: SUCCESS

9. 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

=====

© 2016 Brent Laster

Lab 7 - 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 **gbuild** function as running "**gradle**", we can reuse it here to call each test. We'll wrap each call to **gradle** (via the **gbuild** function in a different node. So our initial pass at a parallel block might look like this.

```
parallel (  
    master: { node ('master'){  
        // always run with a new workspace  
        step([$class: 'WsCleanup'])  
        gbuild this, '-D test.single=TestExample1 test'  
    }},  
    worker2: { node ('worker_node2'){  
        // always run with a new workspace  
        step([$class: 'WsCleanup'])  
        gbuild this, '-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.

```
gedit ~/gradle-demo/Jenkinsfile
```

Your file should look like the one below.

```

#!/groovy
import static org.conf.Utilities.*
node ('worker_node1') {
    // always run with a new workspace
    step([$class: 'WsCleanup'])
    try {
        stage('Source') {
            checkout scm
            stash name: 'test-sources', includes: 'build.gradle,src/test/'
        }
        stage('Build') {
            // Run the gradle build
            gbuild this, 'clean build -x test'
        }
        stage ('Test') {
            // execute required unit tests in parallel
            parallel (
                master: { node ('master'){
                    // always run with a new workspace
                    step([$class: 'WsCleanup'])
                    gbuild this, '-D test.single=TestExample1 test'
                }},
                worker2: { node ('worker_node2'){
                    // always run with a new workspace
                    step([$class: 'WsCleanup'])
                    gbuild this, '-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>

git add Jenkinsfile

git commit -m "updated for testing"

git push

5. Go back and re-run the **branch indexing** or build of the project for **test** again.

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

Select the **"Branch Indexing"** left menu item.

Click **"Run Now"**.)

6. Now switch back to the **"test"** project under **"demo-all"**. It should have run again (if not, select **"Build Now"**). 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 **master** were trying to run the tests. Why is this?

```
...
[Pipeline] }
[Pipeline] // stage
[Pipeline] stage
[Pipeline] { (Test)
[Pipeline] parallel
[Pipeline] [master] { (Branch: master)
[Pipeline] [worker2] { (Branch: worker2)
[Pipeline] [master] node
[master] Running on master in /var/lib/jenkins/workspace/demo-all_test-
2NSKNREYDSQJJDIIJNWQBDGJOQGIJ2XBTUWE2GRP3JLQPD7DCS5Q
[Pipeline] [worker2] node
[worker2] Running on worker_node2 in /home/jenkins2/worker_node2/workspace/demo-all_test-
2NSKNREYDSQJJDIIJNWQBDGJOQGIJ2XBTUWE2GRP3JLQPD7DCS5Q...
...
[worker2]
[worker2] FAILURE: Build failed with an exception.
[worker2]
[worker2] * What went wrong:
[worker2] Task 'test' not found in root project 'demo-all_test-
2NSKNREYDSQJJDIIJNWQBDGJOQGIJ2XBTUWE2GRP3JLQPD7DCS5Q'.
[worker2]
...
[master]
[master] FAILURE: Build failed with an exception.
[master]
[master] * What went wrong:
[master] Task 'test' not found in root project 'demo-all_test-
2NSKNREYDSQJJDIIJNWQBDGJOQGIJ2XBTUWE2GRP3JLQPD7DCS5Q'.
[master]
...
```

6. The reason we can’t run the **Test** stage on **master** (or on **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.

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

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

8. 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 cleanup 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
import static org.foo.Utilities.*
node ('worker_node1') {
    // always run with a new workspace
    step([$class: 'WsCleanup'])
    try {
        stage('Source') {
            checkout scm
            stash name: 'test-sources', includes: 'build.gradle,src/test/'
        }
        stage('Build') {
            // Run the gradle build
            gbuild this, 'clean build -x test'
        }
        stage ('Test') {
            // execute required unit tests in parallel

            parallel (
                master: { node ('master'){
                    // always run with a new workspace
                    step([$class: 'WsCleanup'])
                    unstash 'test-sources'
                    gbuild this, '-D test.single=TestExample1 test'
                }},
                worker2: { node ('worker_node2'){
                    // always run with a new workspace
                    step([$class: 'WsCleanup'])
                    unstash 'test-sources'
                    gbuild this, '-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 one will fail (with a legitimate test failure).

```
...
[Pipeline] stage
[Pipeline] { (Test)
[Pipeline] parallel
[Pipeline] [master] { (Branch: master)
[Pipeline] [worker2] { (Branch: worker2)
[Pipeline] [master] node
[master] Running on master in /var/lib/jenkins/workspace/demo -all_test-
2NSKNREYDSQJJDIIJNWQBDGJOQGIJ2XBTUWE2GRP3JLQPD7DCS5Q
[Pipeline] [worker2] node
[worker2] Running on worker_node2 in /home/jenkins/worker_node2/workspace/demo-all_test-
2NSKNREYDSQJJDIIJNWQBDGJOQGIJ2XBTUWE2GRP3JLQPD7DCS5Q
...
[Pipeline] [master] sh
[master] [demo-all_test-2NSKNREYDSQJJDIIJNWQBDGJOQGIJ2XBTUWE2GRP3JLQPD7DCS5Q] Running shell
script
[master] + /usr/share/gradle/bin/gradle -D test.single=TestExample1 test
[Pipeline] [worker2] tool
[Pipeline] [worker2] sh
[worker2] [demo-all_test-2NSKNREYDSQJJDIIJNWQBDGJOQGIJ2XBTUWE2GRP3JLQPD7DCS5Q] Running shell
script
[worker2] + /usr/share/gradle/bin/gradle -D test.single=TestExample2 test
...
[master] :compileTestJava
[master] :processTestResources UP-TO-DATE
[master] :testClasses
[worker2] :test
[worker2]
[worker2] TestExample2 > example2 FAILED
[worker2]   org.junit.ComparisonFailure at TestExample2.java:10
[worker2]
[worker2] 1 test completed, 1 failed
[worker2] :test FAILED
[worker2]
[worker2] FAILURE: Build failed with an exception.
[worker2]
[worker2] * What went wrong:
[worker2] Execution failed for task ':test'.
[worker2] > There were failing tests. See the report at: file:///home/jenkins/worker_node2/workspace/demo-
all_test-2NSKNREYDSQJJDIIJNWQBDGJOQGIJ2XBTUWE2GRP3JLQPD7DCS5Q/build/reports/tests/index.html
[worker2]
```

=====

END OF LAB

=====