Jenkins 2 – Up and Running

Revision 1.1 – 5/5/18

Brent Laster

IMPORTANT NOTES:

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

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

3. 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.

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

Lab 1 starts on the next page!

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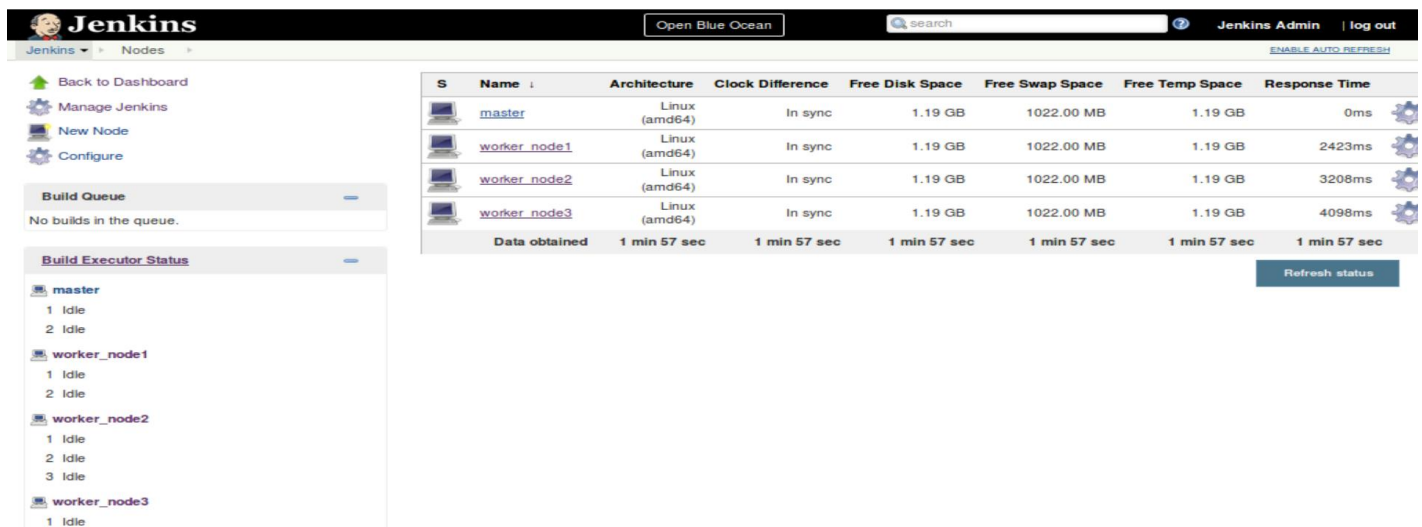Access/manage Jenkins from your shell, or from your script.

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", "workder_node2", and "worker_node3". We'll use these 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.

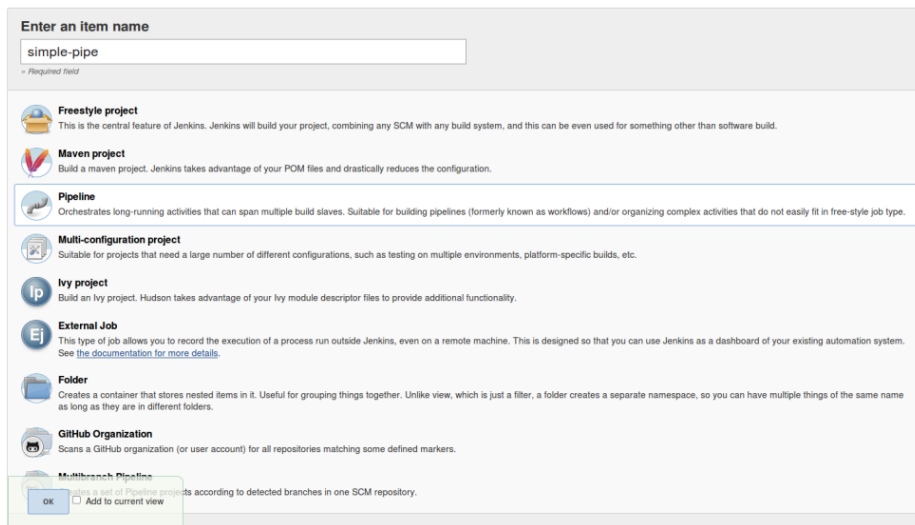5. Click on "**Back to Dashboard**" (upper left) to get back to the Jenkins Dashboard. Now we'll create our first pipeline project. In the left column, click on "**New Item**". Notice that there are quite a few different types of items that we can

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@diyvb2:/opt/git/gradle-demo.git'

    }

}
```

This is what it will look like afterwards.
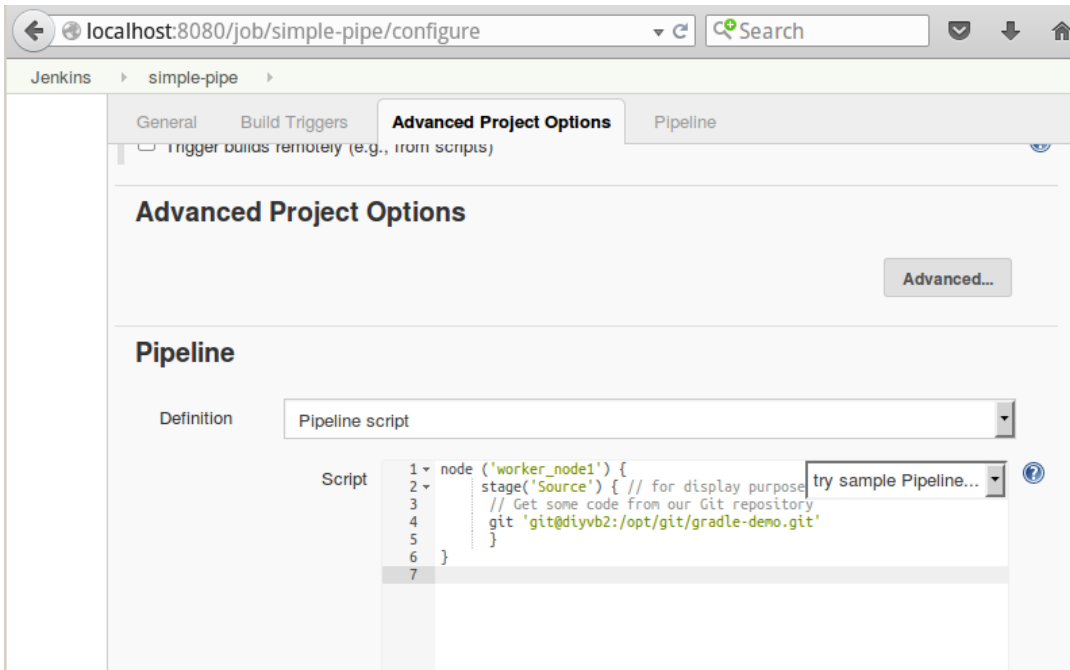
Now, click the **Save** button to save your changes.

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.
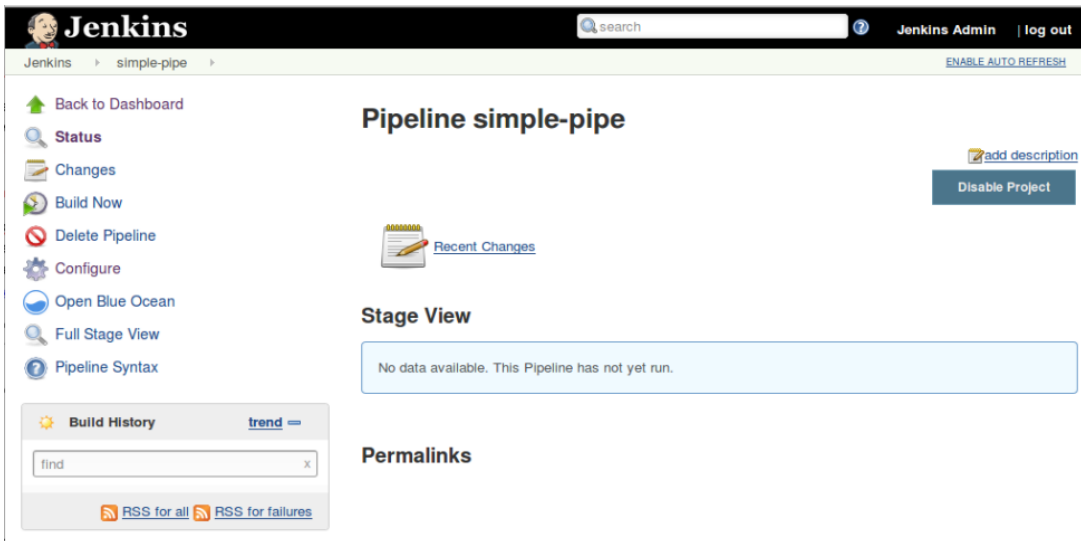
8. Click on the "**Build Now**" link to build your job. (It may take a moment to startup and run.) After it runs, notice that you now see a **Stage View** that is more informative. The "**Source**" name is what we had in our stage definition. We also have the time the stage took and the green block indicating success.
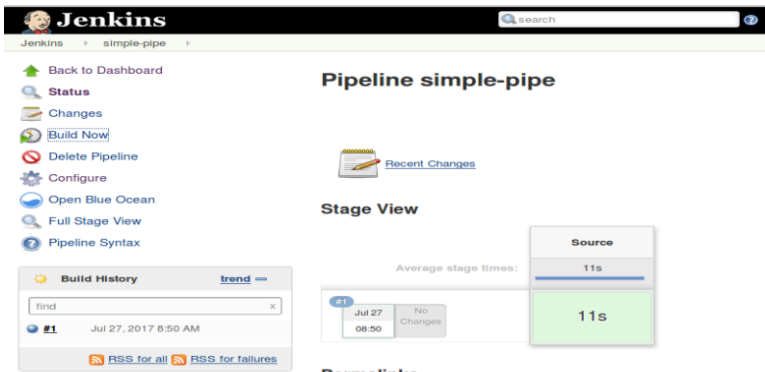
9. Let's look at the logs for this stage via this page. Hover over the green box for the **Source** stage until you see another box pop up with a smaller box named "**Logs**".
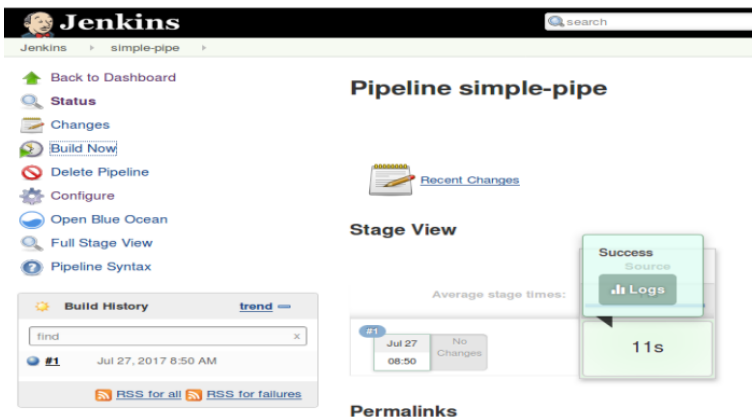
10. Now, click on that smaller "**Logs**" box inside the "**Success**" box to see the log for the run of the stage pop up.

(Of course, you could also use the traditional Console Output route to see the logs.)
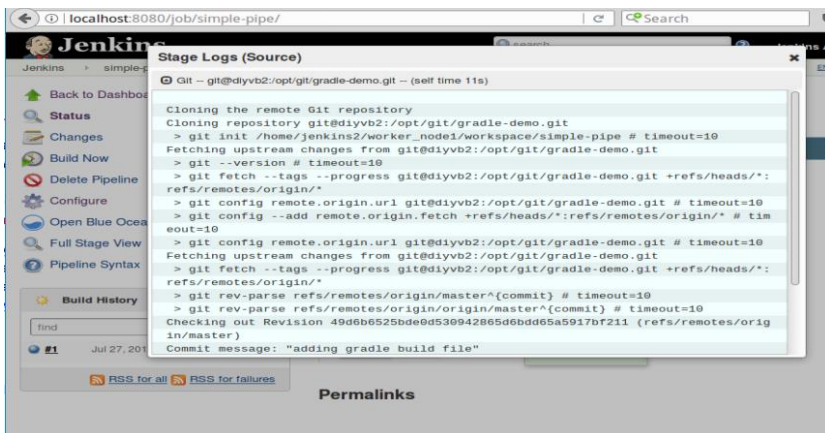
11. Close the popup window for the logs and click on **Back to Dashboard** to be ready for the next lab.

==

END OF LAB

==

© 2018 Brent Laster

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 Replay 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 (continued on next page) 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') {

    def gradleHome

    stage('Source') { // for display purposes

        // Get some code from our Git repository
```

© 2018 Brent Laster

```
                    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. (There is a double quote and then a single quote before ${gradleHome …)

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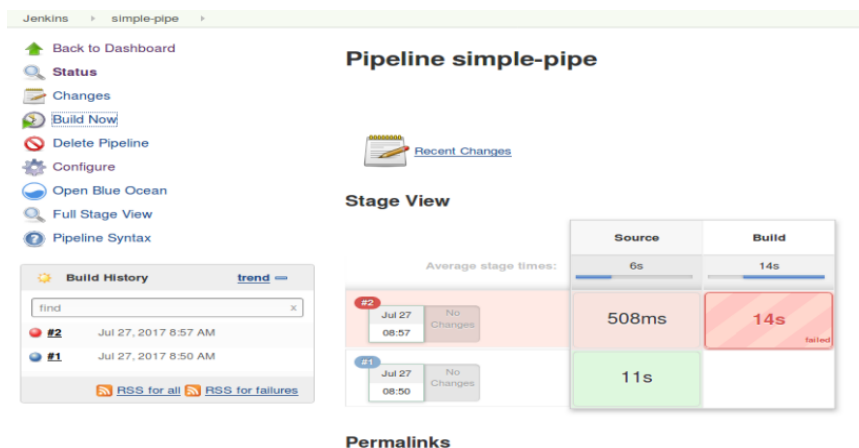
8. Hover over each box for the stages in the most recent run (#2) to see the status information for that stage.

Brent Laster
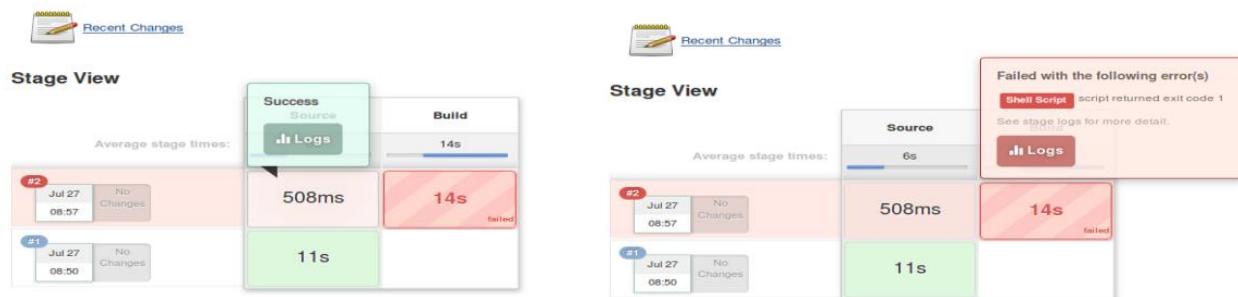
9. While hovering over the **Build** stage box, notice the message telling you which step failed ("**Shell Script**"). Click on the **Logs** link to bring those up. In the pop-up box, you'll see a description of the steps involved in this stage. Click on the small down arrow in the square next to "**Shell Script**" to expand the log information. What is the error message?
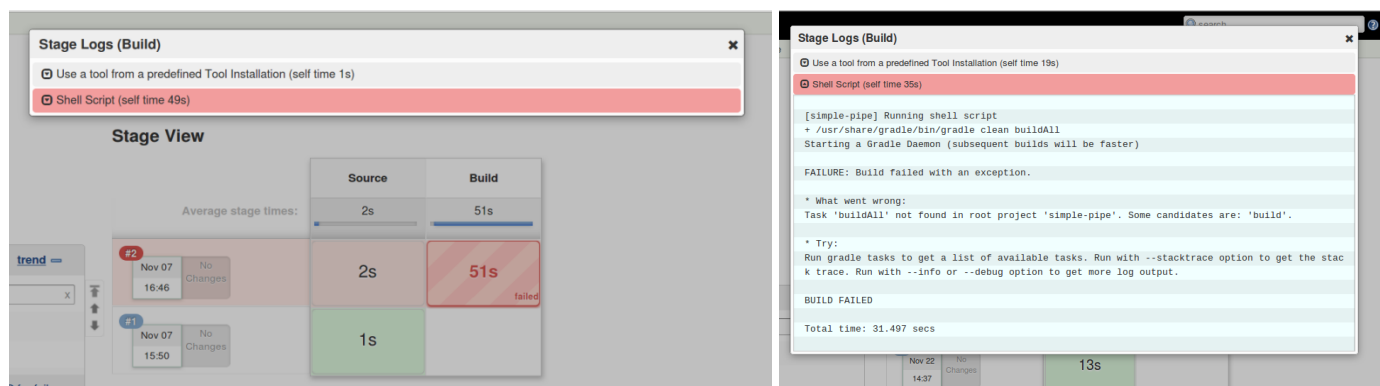
10. The log shows the problem - there is not a "**buildAll**" task. The reason is that, for Gradle, the task should be "**build**". Let's fix that. Use the escape key or X in the corner to close the **Stage Logs (Build)** window.
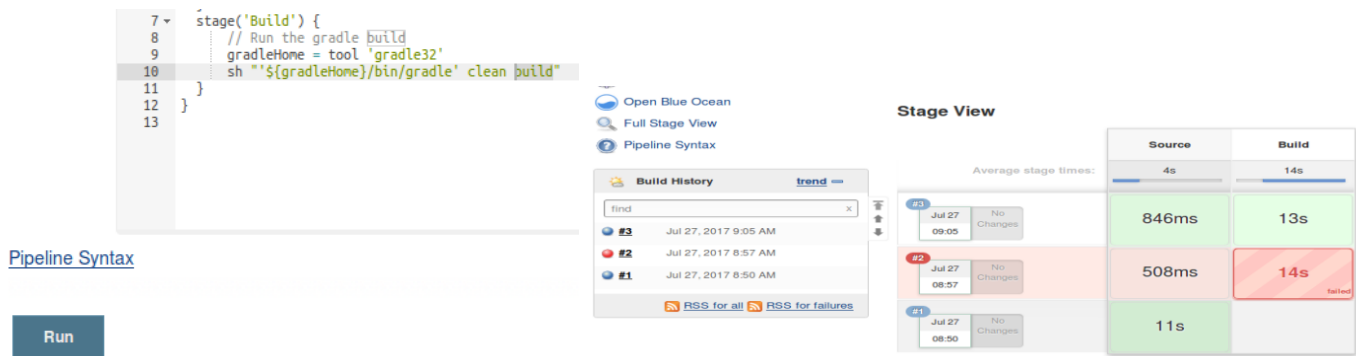
11. Before we change our script though, let's use the **Replay** functionality to test this change out. To use this, we'll need to get to the screen for the specific build that failed. In the "**Build History**" window, click on the red ball next to the latest build, or click on the "**Last build (#2)**" link in the **Permalinks** section.

12. 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.

© 2018 Brent Laster
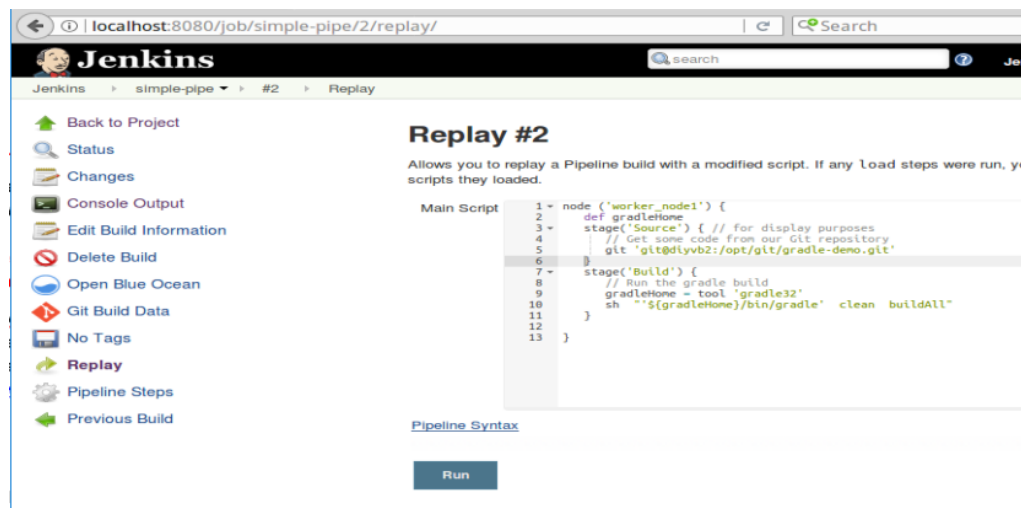
13. On the line that has the gradle command and the "**buildAll**" task, change "**buildAll**" to just "**build**". Then click the "**Run**" button. Notice that this kicks off another build, which should complete successfully after a few moments.

14. Click on the **Configure** menu item on the left. Scroll down to the **Pipeline** section and look at the code in the **Pipeline 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**.

© 2018 Brent Laster

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**, enter "**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 "**1.5**", so enter that. Note that while a branch name can be used here, updates to that may not be picked up over time.

Leave the **"Load implicitly"** checkbox unchecked. Loading a library implicitly is a convenience, but we don't want to have it loaded for every project we create.

Leave the "**Allow default version to be overridden**" box checked 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'))

 You can leave "**Include @Library changes in job recent changes**" checked as well.

For **Retrieval method**, select **Modern SCM**. This is allowing pulling the library from source control via the Git plugin that has been updated for pipeline.

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.

Library Name	Utilities
Default version	1.5
	Cannot validate default version until after saving and reconfiguring.
Load implicitly	☐
Allow default version to be overridden	☑
Include @Library changes in job recent changes	☑

Retrieval method

◉ Modern 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.

After selecting **Git,** enter git@diyvb2:/opt/git/shared-libraries.git in the **Project Repository** field.

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 load the shared library, 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. The gbuild line replaces the previous gradle line. (Lines referencing gradleHome are no longer needed and can be removed.)

@Library('Utilities@1.5')

import static org.conf.Utilities.*

node ('worker_node1') {

    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

        gbuild this, 'clean build'

    }

}

```
Pipeline script

Script    1   @Library('Utilities@1.5')
          2   import static org.conf.Utilities.*
          3 ▾ node ('worker_node1') {
          4 ▾     stage('Source') { // for display purposes
          5           // Get some code from our Git repository
          6           git 'git@diyvb2:/opt/git/gradle-demo.git'
          7       }
          8 ▾     stage('Build') {
          9           // Run the gradle build
         10           gbuild this, 'clean build'
         11       }
         12   }
         13   
```

© 2018 Brent Laster

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")**
>
> **}**


```
Pipeline script

Script
 1   @Library('Utilities@1.5')
 2   import static org.conf.Utilities.*
 3 ▾ node ('worker_node1') {
 4       def gradleHome
 5 ▾     stage('Source') { // for display purposes
 6           // Get some code from our Git repository
 7           git 'git@diyvb2:/opt/git/gradle-demo.git'
 8       }
 9 ▾     stage('Build') {
10           // Run the gradle build
11           gradleHome = tool 'gradle32'
12           gbuild this, 'clean build'
13       }
14 ▾     stage ('Verify') {
15           // Now load 'verify.groovy'.
16           def verifyCall = load("/home/diyuser2/shared-libraries/s
17           verifyCall("Please Verify the build")
18       }
19
```

☑ Use Groovy Sandbox

Pipeline Syntax

© 2018 Brent Laster

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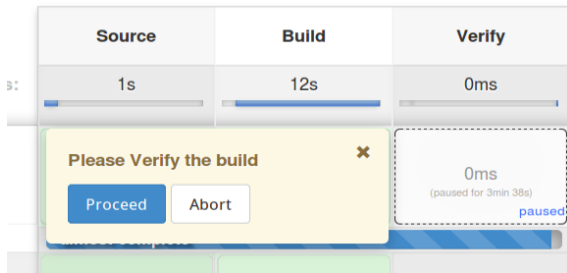
4. As well, if you look at the console output, you'll see the process waiting for you to tell it to **proceed** or **abort** (via clicking on one of the links).

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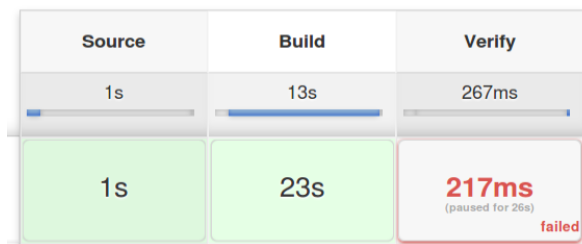
6. Let's add a timeout to make sure the build doesn't go on forever. To figure out the exact syntax, we'll use the **Snippet Generator** tool. Click on the **Configure** button for the job, then scroll to the bottom and click on the **Pipeline Syntax** link.

© 2018 Brent Laster

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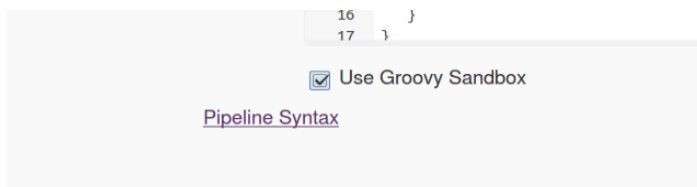
7. You'll now be in the Snippet Generator page. Here you can select what you want to do and then have Jenkins generate the Groovy pipeline code for you. To get our timeout code, select the following values:

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 t(
interested in from the list, configure it, click **Generate Pipeline Script**, and you will see a
with that configuration. You may copy and paste the whole statement into your script, or pi
parameters are optional and can be omitted in your script, leaving them at default values.)

Steps

Sample Step   timeout: Enforce time limit

Timeout   5

Unit   SECONDS

Generate Pipeline Script

```
timeout(time: 5, unit: 'SECONDS') {
    // some block
}
```

Global Variables

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.

===

© 2018 Brent Laster

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

Prerequisite: If you want to use the email functionality in this lab and have not already configured the email setup in Jenkins, you should do that now. (See instructions in the setup document.)

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.   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")

    }
  } // end node  (already in pipeline)
```

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

4. **Save** and **Build Now**. Try it with selecting **Proceed** and 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.

<div align="center">node ('worker_node1') {</div>

<div align="center">© 2018 Brent Laster</div>

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

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

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

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

==

END OF LAB

==