UberConf 2016

Creating a Deployment Pipeline with Jenkins

Brent Laster

Setup - Installing the Image

As mentioned in the workshop description, this workshop uses a pre-configured VM running under VirtualBox.   The image will be made available to you through USB drives to be passed around to the group and via a file server (location to be shared).

You will already need to have VirtualBox installed on your particular system.   (Copies of the various install programs may be available at the workshop but are not guaranteed. Your instructor will share details.)

Once you have Virtualbox installed and are ready to proceed, do the steps below to complete installation.
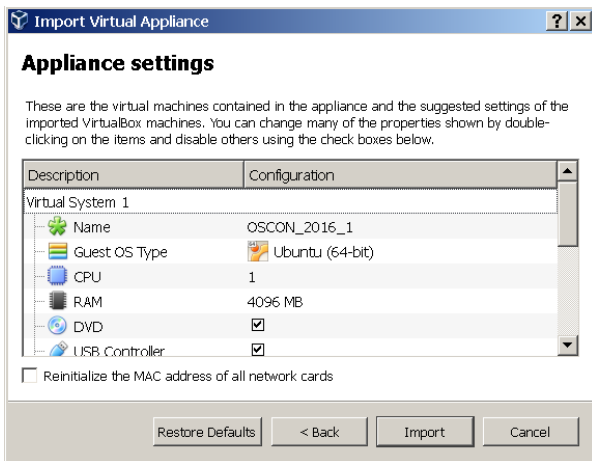
1. Open **VirtualBox** on your system.

2. From the **File** menu, select **"Import Appliance…".**
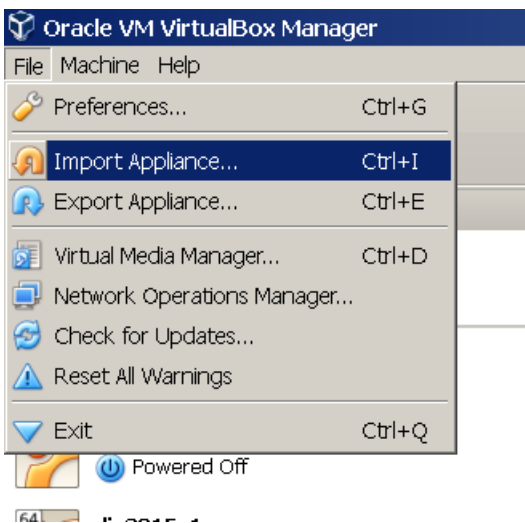
3.   From there, you can type in (or browse to, using the folder icon circled in the picture) the path of the **OSCON_2016.ova** file.

© 2016 Brent Laster
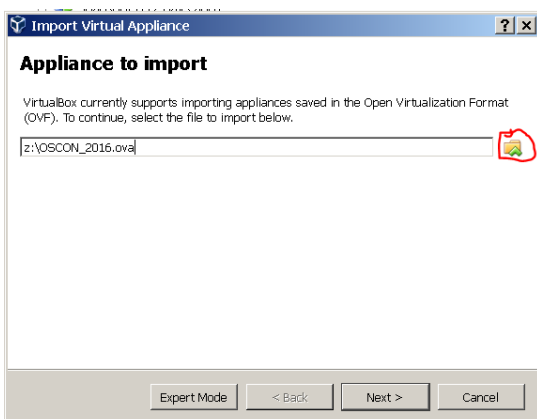
4. You can just accept the **Appliance Settings** and click the **Import** button.


5. After the import is finished, you should have a VM listed in VirtualBox named **OSCON_2016**. Start it up by right-clicking on it, selecting **Start** and then **Normal Start**. (Yours will not show Snapshot 7.)
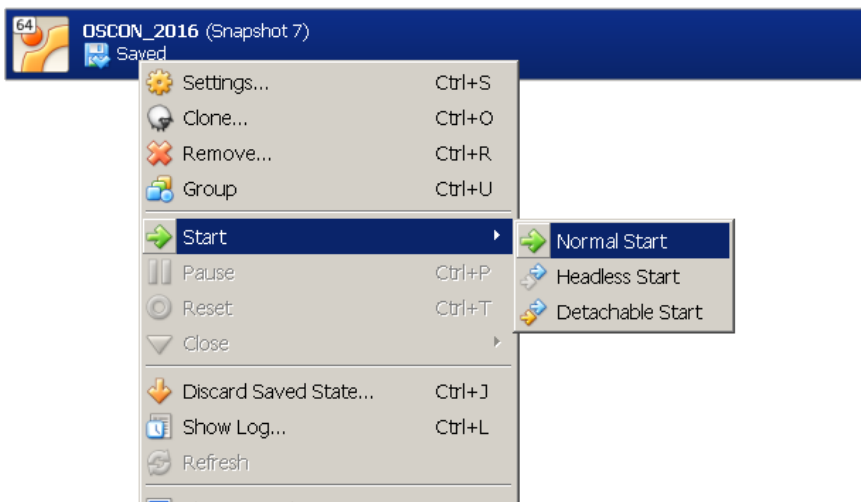
6. You may receive an error about network adapters here. If so, just click on the option given. You can also enable enable "**Auto Ethernet**" in the networking menu (click on double arrors in upper-right corner).
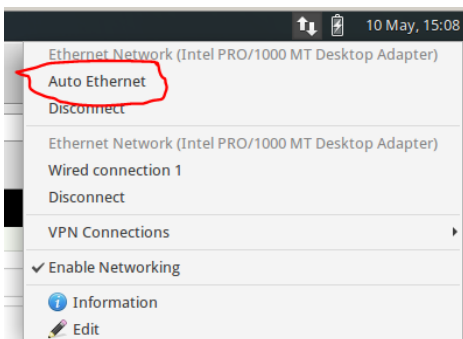

7. After starting up the VM, you should see the desktop of the VM.

© 2016 Brent Laster
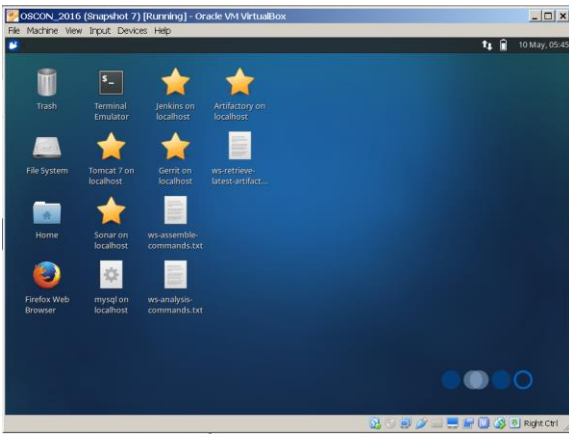
8. We need to check one more thing here. Occasionally, **Gerrit** may not start up correctly. Check to see if Gerrit is running by clicking on the bookmark named "**Gerrit**" in the bookmarks toolbar or by going directly to **http://localhost:8081.**

9. If you see the Gerrit homepage (below) things are good. If not, proceed to step 10.
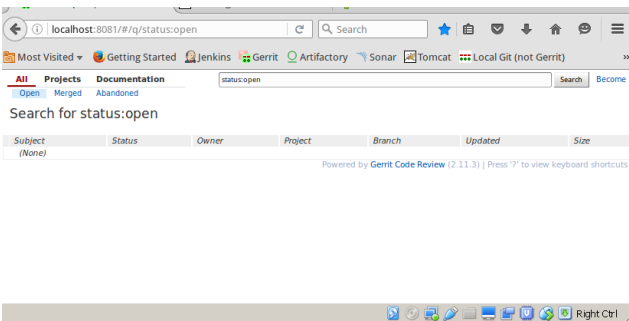
10. If Gerrit needs to be started, do the following.

11. Open up a terminal session by either clicking on the "**Terminal Emulator**" shortcut on the desktop or by selecting the "**Terminal Emulator**" selection from the system menu (drop down from the mouse in the upper-lefthand side). Then, in the terminal session, enter "**sudo service gerrit stop**". Wait for that to complete. Then enter "**sudo service gerrit start**".

© 2016 Brent Laster
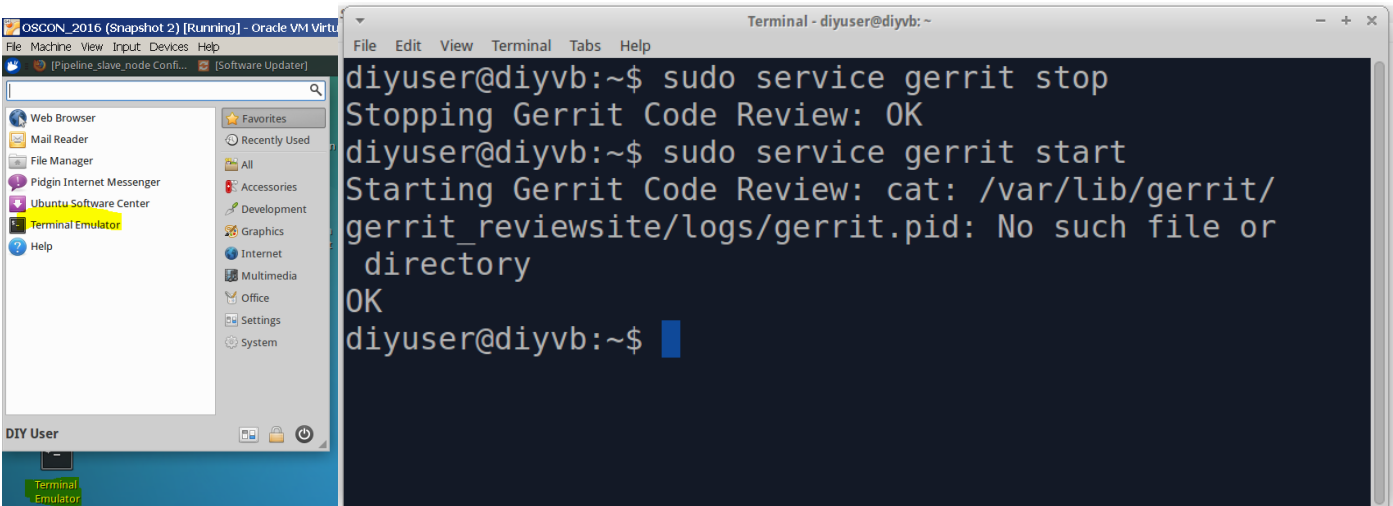
Lab 1 - Creating a Jenkins Slave Node

Purpose: In this first lab, we'll create a new Jenkins slave node to use for our pipeline "build" processes. (Note that you wouldn't normally create a slave on the same machine as the master but we'll do this here for simplicity.)

1. Start Jenkins by clicking on the "**Jenkins on localhost**" 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 = **diyuser** and Password = **diyuser**

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

3. Click on the "**Manage Jenkins**" link in the menu on the left-hand side. Next, look in the list of selections in the middle of the screen, and find and click on "**Manage Nodes**".

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.

Manage Credentials
Create/delete/modify the credentials that can be used by Jenkins and by jobs running in Jenkins to connect to 3rd pa

4. On the next screen, click on "**New Node**" on the menu on the left-hand side.

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

  pipeline_slave

 Click on the "**Dumb Slave**" radio button. Then click on "**OK**".

6. You should now be on the configuration screen for the new node. Find the "**Remote root directory**" text field.   Type in the following:

  /home/jenkins/pipeline_slave

7. Under that, fill in the "**Labels**" area with

  pipeline_slave

8. Leave the "**Launch method**" as "**Launch slave agents on Unix machines via SSH**".

9. Under the "**Launch method**" section, for "**Host**" enter the name of this machine:

     diyvb

   In the "**Credentials**" drop-down, select "**jenkins (ssh)**". (If you see more than one, use the first jenkins in the list.)

Name	pipeline_slave
Description	
# of executors	1
Remote root directory	/home/jenkins/pipeline_slave
Labels	pipeline_slave
Usage	Utilize this node as much as possible
Launch method	Launch slave agents on Unix machines via SSH
Host	diyvb
Credentials	jenkins (ssh)　　　　　　　　　　　　　❤ Add
Availability	Keep this slave on-line as much as possible

Node Properties

☐ Environment variables
☐ Tool Locations

[Save]

10. Click on **Save**.

11. You should see a note that indicates the slave is being launched.   If not, in the upper right corner, click the "**Enable Auto Refresh**" link to have the screen automatically updated. As long as the red square with the "x" in it disappears, the node is good.

S	Name ↓	Architecture	Clock Difference	Free Disk Space	Free Swap Space	Free Temp Space	Response Time	
💻	jenkins_slave1	Linux (amd64)	In sync	1.50 GB	4.00 GB	1.50 GB	30ms	🔧
💻	master	Linux (amd64)	In sync	1.50 GB	4.00 GB	1.50 GB	0ms	🔧
💻	pipeline_slave	Linux (amd64)	In sync	1.50 GB	4.00 GB	1.50 GB	1791ms	🔧
	Data obtained	6 min 17 sec	6 min 17 sec	6 min 17 sec	6 min 17 sec	6 min 17 sec	6 min 17 sec	

[Refresh status]

12. Click on "**Back to Dashboard**" (upper left) to get back to the Jenkins Dashboard. There are several different "views" already setup for the jobs we will use.   Currently, the "**(1) Workshop Pipeline View**" should be selected. If not, click that tab.

Lab 2 - Setting up the Review phase

Purpose: One of the easiest ways to setup a new job in Jenkins is to copy an existing one that's similar. For the first phase of our Workshop pipeline, the review stage, we'll set up a verify job (ws-verify) by copying the one from the Reference pipeline and tweaking it as needed.   This will also highlight making a job run on a particular slave node and Gerrit integration in Jenkins jobs.

1. We want to create a new job for the verify phase of our workshop pipeline. We'll do that by copying the reference verify job (**ref-verify**) and modifying it as needed.

2. On the Jenkins dashboard at **http://localhost:8080**, click on "**New Item**" in the upper left corner.   For item-name, enter

      ws-verify

Select "**Copy existing item**". In the "**Copy from**" field, enter

      ref-verify

Item name	ws-verify

○ **Freestyle project**
    This is the central feature of Jenkins. Jenkins will build your project, combining any SCM with any build system, and this can be even used for something other than software build.

○ **Maven project**
    Build a maven project. Jenkins takes advantage of your POM files and drastically reduces the configuration.

○ **External Job**
    This type of job allows you to record the execution of a process run outside Jenkins, even on a remote machine. This is designed so that you can use Jenkins as a dashboard of you
    details.

○ **Multi-configuration project**
    Suitable for projects that need a large number of different configurations, such as testing on multiple environments, platform-specific builds, etc.

◉ **Copy existing item**
    Copy from | ref-verify

OK

3. Click on the **OK** button. You will now be in the configure screen for the new **ws-verify** job.

4. Find the section on "**Advanced Project Options**".   Right above that line, you'll see the "**Restrict where this project can be run**" option.

In the "**Label Expression**" text box,   replace the "**jenkins_slave1**" value with the name of the slave node we created in Lab 1.

      pipeline_slave

(Note: If you happened to name your slave node something different, then you would use that name here. If you did that, you'll need to modify the name in **each** job to be the name you used.)

☑ Restrict where this project can be run

Label Expression   pipeline_slave

Label is serviced by 1 node

Advanced Project Options

5. For the **workshop pipeline**, we will be using a different source management repository so that needs to be updated. Find the **Source Code Management** section and change the path of the source code repository by replacing "**reference**" with "**workshop**". The path should look like

      ssh://jenkins@localhost:29418/workshop

⦿ Git
Repositories

Repository URL   ssh://jenkins@localhost:29418/workshop

Credentials   jenkins (jenkins - ssh)   🔑 Add

6. Locate the "**Gerrit Trigger**" section under the "**Build Triggers**" section on the page. This section is telling Jenkins to kick things off based on a new patchset being created in Gerrit (thus the "**Patchset Created**" setting in the "**Trigger on**" area.

What we need to change here is the project that Gerrit will be using and that Jenkins will trigger on.

7. Locate the "**Gerrit Project**" section (just below "**Dynamic Trigger Configuration**"). Under "**Type**" on the left-hand side (next to the red "**Delete**" button), replace "**reference**" in the "**Pattern**" field with

      workshop

Gerrit Project

Delete	Type	Pattern		Branches	
	Plain	workshop		Type	Pattern
				Path	**	Delete
				Add Branch	
Add Topic					
Add File path					
Add Forbidden File path					

☐ Disable Strict Forbidden File Verification

8. Find the "**Build**" section and notice that we're invoking a local version of Gradle as the build step and passing it a task of "**build**". We'll be talking more about Gradle and how this all fits in as we go along.

9. Save your changes by clicking on the Save button at the bottom of the screen.


===

END OF LAB

===

Lab 3 - Pushing to Gerrit for Review

Purpose: In this lab, we'll exercise the "review stage" of our pipeline, by changing a simple html file in our demo webapp, pushing the change to Gerrit which will trigger Jenkins to do a verification build *before* the change is actually merged into the underlying remote Git repository.

1. Open up a terminal session by either clicking on the "**Terminal Emulator**" shortcut on the desktop or by selecting the "**Terminal Emulator**" selection from the system menu (drop down from the mouse in the upper-lefthand side).
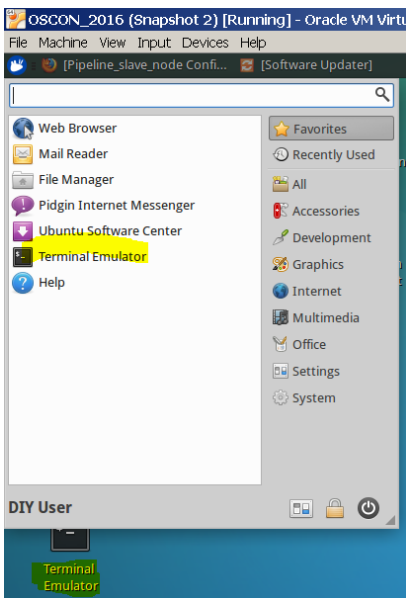
2. Once the terminal session opens, cd to the **workshop** directory.

      cd workshop

3. In this directory, we already have a cloned version of the project that we will update and push out to Gerrit for the Jenkins verify build and code review.

4. Under the "**workshop**" directory, in the "**web/src/main/webapp**" subdirectory, edit the "**agents.html**" file.

© 2016 Brent Laster

(You can use **mousepad** or **gedit** editors.)

> **gedit web/src/main/webapp/agents.html**

Find the line for the main header (about 20 lines down).

> `<h1>R.O.A.R (Registry of Animal Responders) Agents</h1>`

Add a second header line after it substituting in your name. Be sure to include the html tabs
(`<h2>` and `</h2>`).

> `<h2> Managed by ( your name here ) </h2>`

5. Save your file (**File->Save**) and **quit/exit** the editor.

6. <u>From the workshop directory</u>, do a local build to make sure things compile.

> **gradle build**

> This should finish with a "**BUILD SUCCESSFUL**" message.

7. Run the local instance to make sure the changes look as expected. Do this by running the **jettyRun** task (note case) and then opening a browser to **http://localhost:8086/com.demo.pipeline** in a new tab.

> **gradle jettyRun**

(Note: This will get to some percentage and then start the system running. Don't kill this - it is working.)

```
:api:jar UP-TO-DATE
:web:compileJava UP-TO-DATE
:web:processResources UP-TO-DATE
:web:classes UP-TO-DATE
> Building 93% > :web:jettyRun > Running at http://localhost:8086/com.demo.pipe
```

8. <open in new browser tab> **http://localhost:8086/com.demo.pipeline (or click on "Local jettyRun" bookmark)**

NOTE: note that the URL is **pipeline** on the end not just pipe as shown. You should see something like the following:

© 2016 Brent Laster

9. Close the browser tab (not the entire session) and kill the running job in the terminal **(Ctrl-C) .**

10. Since we've verified that things work as expected, push the change over to the Gerrit remote. First configure your **user.name**. (**user.email** is already configured to match our Gerrit user.) Be careful of the typing. Note that there are two dashes/hyphens before **global** and **user.name** is two words with only a period inbetween. Also the name string should be enclosed in quotes since it has a space.

> **git config --global user.name "Firstname Lastname"**

11. Now add and commit the change. This assumes again that you are still in the **workshop** directory. (Ignore any messages about line ending changes.)
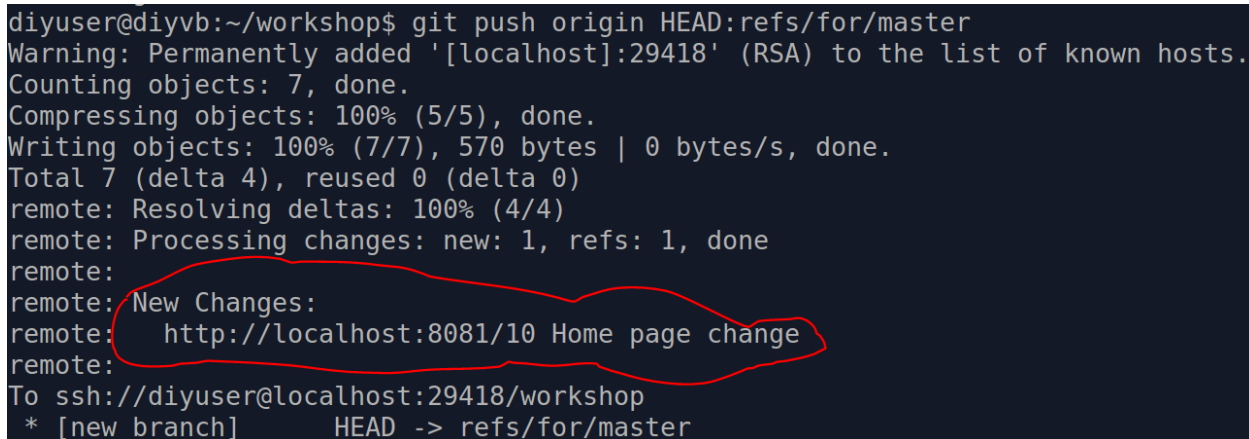
> **git add web/src/main/webapp/agents.html**

> **git commit -m "home page change"**

12. Now push the change to Gerrit. Note the syntax - that's "HEAD", then a colon, then "refs/for/master" - all together.

> **git push origin HEAD:refs/for/master**

This will create a new Gerrit change with a first patchset (revision). This is not in the remote repository yet. It's being held by Gerrit for review. You can find the full link to the new change and it's first patchset (revision) in the push commands' output. Look for something like shown below.

```
diyuser@diyvb:~/workshop$ git push origin HEAD:refs/for/master
Warning: Permanently added '[localhost]:29418' (RSA) to the list of known hosts.
Counting objects: 7, done.
Compressing objects: 100% (5/5), done.
Writing objects: 100% (7/7), 570 bytes | 0 bytes/s, done.
Total 7 (delta 4), reused 0 (delta 0)
remote: Resolving deltas: 100% (4/4)
remote: Processing changes: new: 1, refs: 1, done
remote:
remote: New Changes:
remote:    http://localhost:8081/10 Home page change
remote:
To ssh://diyuser@localhost:29418/workshop
 * [new branch]      HEAD -> refs/for/master
```

13. Open up the Gerrit website in a separate tab in the browser at **http://localhost:8081**. (There's also a shortcut in the bookmarks toolbar named "**Gerrit**"(Switch to a new tab first.)). This session is configured in a debug mode that lets us easily switch between preconfigured users.

14. In the upper right corner, click on the **Become** link. Then click on "**Workshop User**".

© 2016 Brent Laster

Sign In

Username:	[ ] [Become Account]
Email Address:	[ ] [Become Account]
Account ID:	[ ] [Become Account]
Choose:	Local Administrator
	Spock (Contributor)
	McCoy (Reviewer)
	Kirk (Committer)
	Workshop User
	Jenkins

15. You should see the change you just pushed under the "**Outgoing reviews**" section.   (If you don't see it, click on **My->Changes** in the menu.)

16. Click on the commit message there ("**home page change**") to open up the review.

17. On this page, near the middle right, find the section with the verification checks for **Code-Review** and **Verified**. The **Verified +1** means that Jenkins did a verification build on this change. (If you don't see that yet, wait a moment and refresh the screen.) The absence of any vote next to **Code-Review** means it is waiting on a code-review. (We will take care of that in the next lab.)

18. Look at the history at the bottom of this change window.   There are lines for the Jenkins verification build.   Click on the line with the Jenkins "**Build Started**" link to expand it and then **click on that link.**
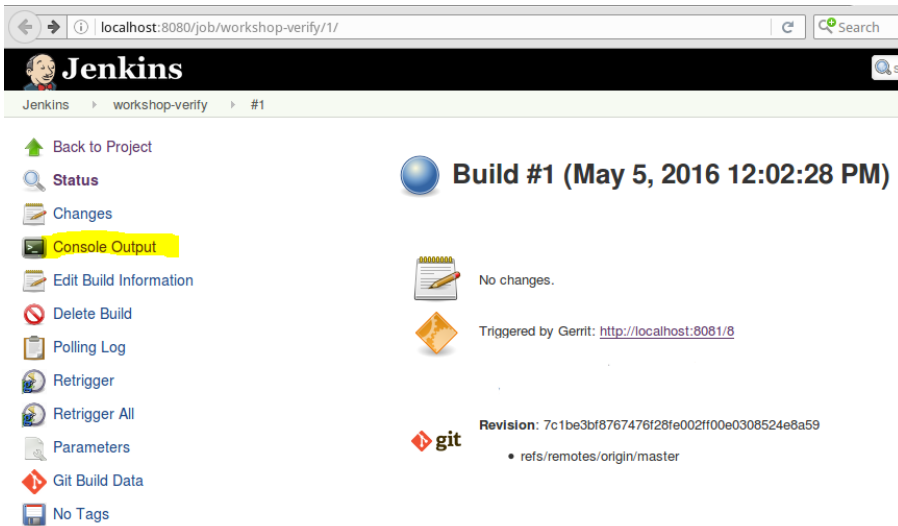
This will take you to a build output page in Jenkins for that run of your workshop-verify job.
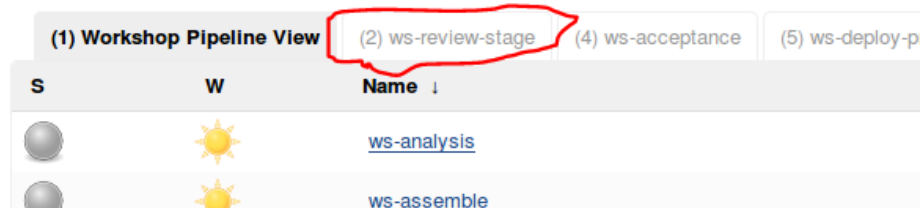
© 2016 Brent Laster

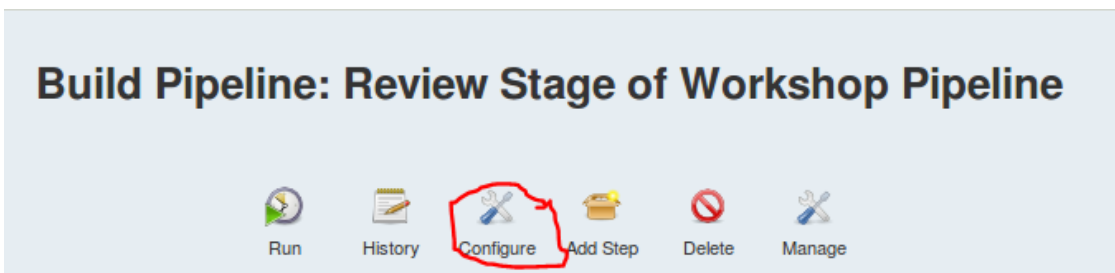**19.** You can click on the "**Console Output**" link to see the actual output of the build step.

20. One last step here is that we want to update Jenkins view that represents the review stage of our **Workshop pipeline**. Since we created the **ws-verify** job, we want to make that the starting job in our the **ws-review-stage** view. Back on the Jenkins homepage, in the row of tabs above the jobs, select the tab for "**(2) ws-review-stage**".
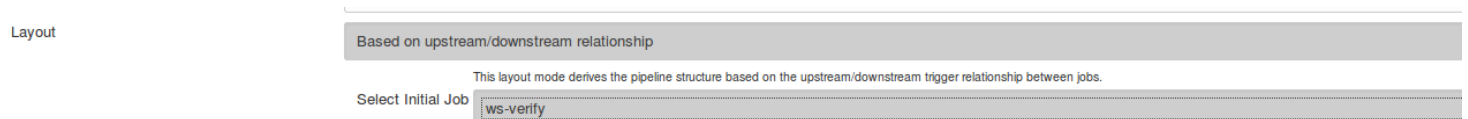
21. On the next screen, select the "**Configure**" icon.
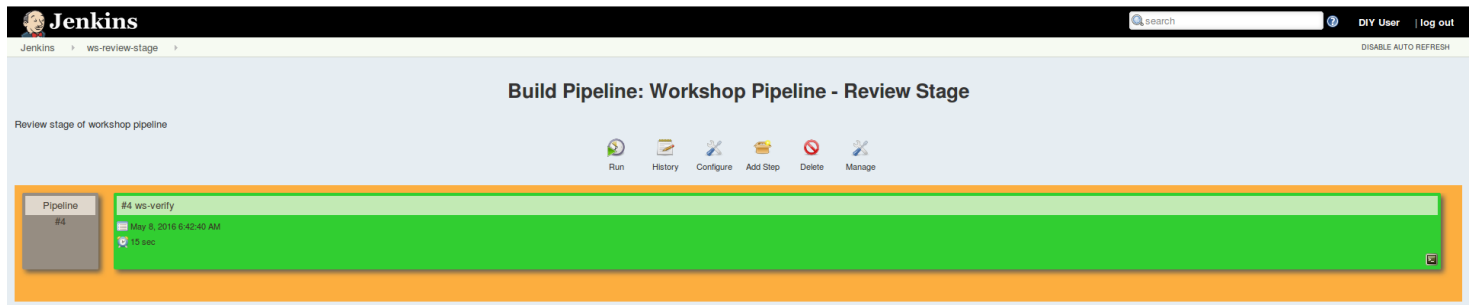

22. On the configuration page, find the section for "**Select Initial Job**". Change that from "**ref-verify**" to "**ws-verify**" (may have to scroll down to see it).

23. Save your changes by clicking on the "**OK**" button at the bottom.

24. You should now be able to see the **pipeline view** of the **review stage** with **ws-verify** in it.

===

END OF LAB

===

Lab 4 - Setting up the Compile Job

Purpose: In this lab, we'll fill in the pieces to get our "compile" job running in the workshop pipeline. The purpose of the compile job is to compile our code and run any unit tests that we have.   It is the first piece of our commit stage and is triggered by the code we have under review in Gerrit being approved, submitted, and merged into the underlying Git repository.

1. Start out on the Jenkins dashboard and click the link in the job list to open up **ws-compile**. This is the job in our pipeline that will compile our code and run unit tests against it.

2. Click on the **Configure** button in the left-hand menu.

© 2016 Brent Laster
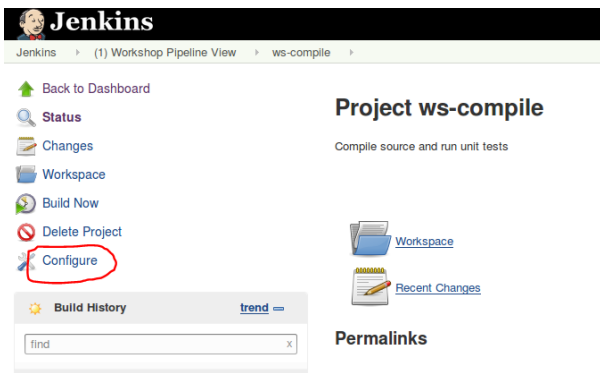
3. On the configuration page find the "**Build Triggers**" section. In this area, under the "**Gerrit Triggers**" section, notice that we are again triggering based on a **Gerrit event**. But this time we are using the "**Change Merged**" event for when content gets reviewed, approved, submitted and merged into the repository. Also notice that under "**Dynamic Trigger Configuration**" and under "**Gerrit Project**", we are looking at the "**workshop**" project and all branches. Nothing needs changing here.
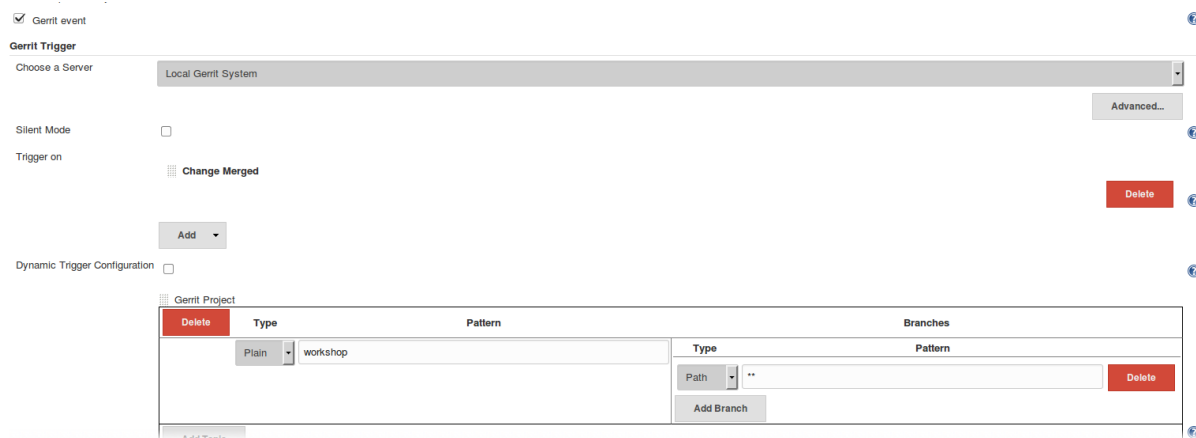
4. Find the "**Build Environment**" section. Previously, when we built locally, we relied on cached versions of the dependencies we needed in the local gradle cache on the disk. Now we want to use "official" versions of artifacts out of our **Artifactory** instance. We will let Jenkins handle the **Gradle-Artifactory integration** rather than us doing it in our Gradle script. To do this, enable (check) the "**Gradle-Artifactory Integration**" checkbox.

Check the box for **"Gradle-Artifactory Integration".**

5. Under "**Deployment Details**", (if not already set) go ahead and set the "**Artifactory deployment server**" to our local one at

> **http://localhost:8082/artifactory**

(Even though we aren't deploying anything in this job, it will avoid an error message.)

© 2016 Brent Laster

6. Under "**Resolution Details**", (if not already set) set the "**Artifactory resolve server**" to the same value:

http://localhost:8082/artifactory

7. Since we will be resolving artifacts from our Artifactory system, we need to specify a repository to get the artifacts from. In the "**Resolution repository**" field, select the "**Different Value**" button at the end and then type in

libs-release

Build Environment

- ☐ Ant/Ivy-Artifactory Integration
- ☐ Generic-Artifactory Integration
- ☑ Gradle-Artifactory Integration

Artifactory Configuration

Deployment Details

Artifactory deployment server   http://localhost:8082/artifactory

Publishing repository

Custom staging configuration

☐ Override default credentials

Resolution Details

Artifactory resolve server   http://localhost:8082/artifactory

Resolution repository   libs-release

☐ Override default credentials

More Details

7. Under "**More Details**" uncheck all the check boxes since we're not publishing anything.

More Details

- ☐ Project uses the Artifactory Gradle Plugin
- ☐ Capture and publish build info
- ☐ Publish artifacts to Artifactory
- ☐ Enable isolated resolution for downstream builds (requires Artifactory Pro)
- ☐ Enabled Release Management

8. Under the "**Invoke Gradle build script**" section in the "**Build**" section, enter the following for "**Tasks**"

© 2016 Brent Laster

clean compileJava test -x artifactoryPublish

(What we are doing here is telling it to clean any previous builds (clean), compile our Java code (compileJava), run our unit tests (test), but don't run the default artifactory publish (-x artifactoryPublish).   We're not trying to publish anything to Artifactory yet, just resolve dependencies from it.)

Build

⊞ **Invoke Gradle script**

◉ Invoke Gradle

Gradle Version

> gradle27

◯ Use Gradle Wrapper

Build step description

Switches

Tasks

> clean compileJava test -x artifactoryPublish

10. Scroll down to "**Post-build Actions**". The last thing to notice here is that we have an action setup after the build runs to "**Trigger parameterized build on other projects**".

11. What this is doing is triggering another job to start after this one is done AND passing it one or more parameters.   In this case, when we use the **Persistent_Workspace=${WORKSPACE}** string, we are telling it to set the variable **Persistent_Workspace** to the value of the current step's **workspace**.

We're going to pass this workspace along to the next step so it can just reference the workspace directory with all the compiled code in it and not have to recompile it again in its own workspace.

Post-build Actions

⊞ **Trigger parameterized build on other projects**

Build Triggers

Projects to build

> ws-integration-tests

Trigger when build is

> Stable

Trigger build without parameters ☐

⊞ **Predefined parameters**

Parameters

> Persistent_Workspace=${WORKSPACE}

12. **Save** your changes by clicking on the **Save** button at the bottom.

==

END OF LAB

==

© 2016 Brent Laster

Lab 5 - Setting up the Analysis job.

Purpose: This job in our pipeline is responsible for scanning our code for any issues and gathering metrics and reporting back.   It's a quality measure.   We can set acceptable thresholds for some metrics and code coverage. For our example pipeline, we only have limited unit and integration tests as examples of what can be done. The main point of this is the integration and reporting through Sonar and Jacoco.

1. Note that we skipped setup on the **Integration Tests** job which was next in line. Since there isn't very much interesting here, this one is already done for you.   However, you can open it up and look around if you want.

Switch back to the Jenkins dashboard. Click on the **ws-integration-tests** job name and then on the **Configure** button again on the left-hand side.

Under the "**Restrict where this project can be run**" setting, we have this set to "**pipeline_slave**" again to run on our new node.

Under the "**Build**" section, we have an "**Execute shell**" command that fires up mysql and creates a test database for us to use in our integration tests by inputting an sql file with all the commands. (You can look at this file out in the "workshop" directory if you want.)

Then, still in the "**Build**" section, we invoke Gradle with our "**integrationTest**" task.

Finally, as a "**Post-build Action**", we are invoking the next job in line "**ws-analysis**" and passing our workspace location on to it as we did in the **ws-compile** job.

When done reviewing this, you can just use the back-arrow key to go back since we haven't made any changes.

2. Now, let's work on the ws-analysis job. On the Jenkins dashboard (homepage), click on the **ws-analysis** job name and then on the **Configure** button again on the left-hand side.

3. The first part of this job is configuring parameters for the sonarqube metrics we'll be measuring and using.   Find the one labeled "**emailRecipients**" and enter an email address where you can get mail in the "**Default Value**" field.

4. Find the "**Build**" section on the page and the "**Execute shell**" box with the comment about inserting the sonar-runner command. This is where we'll enter the command to run the sonar-runner. Type in:

/opt/sonar-runner/bin/sonar-runner -X -e

© 2016 Brent Laster

Build


```
Execute shell
Command  # place sonar-runner command here
         /opt/sonar-runner/bin/sonar-runner -X -e
```

See the list of available environment variables

5. Next, in the step to "**Invoke Gradle script**", we'll turn on the jacoco coverage reports.

In the "Tasks" entry, type

jacocoTestReport


```
Invoke Gradle script
 ● Invoke Gradle
Gradle Version              gradle27

 ○ Use Gradle Wrapper
Build step description

Switches

Tasks                       jacocoTestReport

Root Build script
```
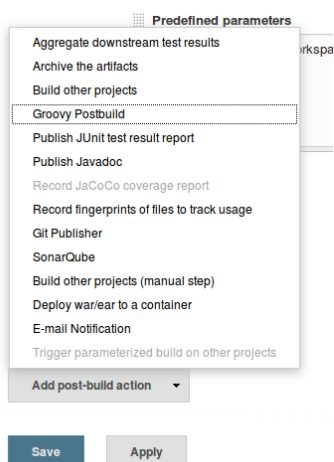
6. Now, we need to add in a post-build step to process the metrics and email the report. Scroll down to the bottom of the screen and find the button for "**Add post-build action**".

7. Click on that button and select "**Groovy Postbuild**".


```
Predefined parameters
    Aggregate downstream test results        orkspa
    Archive the artifacts
    Build other projects
    Groovy Postbuild
    Publish JUnit test result report
    Publish Javadoc
    Record JaCoCo coverage report
    Record fingerprints of files to track usage
    Git Publisher
    SonarQube
    Build other projects (manual step)
    Deploy war/ear to a container
    E-mail Notification
    Trigger parameterized build on other projects

 Add post-build action    ▼

 Save      Apply
```

8. Scroll back up and just under the "**Postbuild Actions**" section, you'll see a new entry for "**Groovy Postbuild**".   In the "**Grooy Script**" textbox, you can just copy in the text from the file on your desktop named **ws-analysis-commands.txt**. (Recommended approach - open file on desktop and copy and paste.)

OR, if you prefer to type them in, here they are.

> **sonarReportsScript="/var/lib/jenkins/scripts/groovy/JenkinsSQReports.groovy"**
>
> **println "Executing script for Sonar report generation from ${sonarReportsScript}"**
>
> **evaluate(new File(sonarReportsScript))**

What this is doing is running a Groovy program to read the parameter values we entered, check them against the generated SonarQube metrics, and send a results email for pass or fail.

9. Change the "**If script fails**" setting to "**Do nothing**" (if not already set that way). We would normally set this to fail the build. But in a workshop environment where the connectivity may be questionable, sending the final email could fail and we don't want that to fail the build.

Post-build Actions

Groovy Postbuild

Groovy Script
```
sonarReportsScript="/var/lib/jenkins/scripts/groovy/JenkinsSQReports.groovy"
println "Executing script for Sonar report generation from ${sonarReportsScript}"
evaluate(new File(sonarReportsScript))
```

☐ Use Groovy Sandbox

Additional classpath   **Add entry**

If the script fails: Do nothing

10. Notice that in the step below that we have the setup for the Jacoco integration between our projects and Jenkins and then a step to invoke the Assembly ("**ws-assemble**") job.

11. **Save** your changes.

===

END OF LAB

===

© 2016 Brent Laster

Lab 6 - Setting up the Assemble Job

Purpose: In this job, we'll do the assembling of our artifacts or packages - the things we'll put into the repository. For demonstration purposes, we'll attach a file in the war. We'll also add versioning information in the style of Semantic Versioning.

1. On the Jenkins dashboad (homepage), click on the **ws-assemble** job name and then on the **Configure** button again on the left-hand side.

2. We start out by defining values for the components of the artifact using parameters. If you want to change the version number for the war, you can change the values for MAJOR_VERSION, MINOR_VERSION here under the "**This build is parameterized**" section. To change these values, enter a different value in the "**Default Value**" section. Or you can leave the default values. **(NOTE: If you change the numbers, make them higher (not 0).)**

3. We want to get these numbers to Gradle so it can use them in the name of the war file we're creating. Normally, we'd do this by passing them as gradle parameters (via the **-P** option on the command line). The Gradle plugin for Jenkins doesn't support that very well, so instead we'll use some shell commands to put them into the **gradle.properties** file which Gradle will see and use.

4. Now, we'll add the commands to put the version number in the gradle properties file. Under the "**Build**" section, in the "**Execute Shell**" section, enter the text from the **ws-assemble-commands.txt** file on the desktop (open the file and copy and paste - recommended approach). Or if you prefer to type them in, they are:

sed -i '/MAJOR_VERSION/c\MAJOR_VERSION='$MAJOR_VERSION gradle.properties
sed -i '/MINOR_VERSION/c\MINOR_VERSION='$MINOR_VERSION gradle.properties
sed -i '/PATCH_VERSION/c\PATCH_VERSION='$PATCH_VERSION gradle.properties
sed -i '/BUILD_STAGE/c\BUILD_STAGE='$BUILD_STAGE gradle.properties

© 2016 Brent Laster

5. Underneath that, in the "**Invoke Gradle script**" section, add the following in the "**Tasks**" section.

   -x test build assemble

  The **assemble** task is responsible for assembling the war file in Gradle.

6. Notice that we are next going to invoke the Publish Artifact job to publish our war file to Artifactory.

7. **Save** your changes.

==

END OF LAB

==

Lab 7 - Setting up the Publish Artifact Job

Purpose: The point of this job in our pipeline is to publish the artifacts we've built, tested, and assembled to an artifact repository for later processing.   To do this, the main thing we need to do is enable Artifactory integration with publishing.

1. On the Jenkins dashboad (homepage), click on the **ws-publish-artifact** job name and then on the **Configure** button again on the left-hand side.

2. Find the "**Build Environment**" section.   We're going to add the Gradle-Artifactory integration to be able to get artifacts from it.

Check the box for "**Gradle-Artifactory Integration**".

Brent Laster

3. Under "**Deployment Details**", go ahead and set (if not already set) the "**Artifactory deployment server**" to our local one at

> **http://localhost:8082/artifactory**

4. Since we will be publishing artifacts from our Artifactory system, we need to specify a repository to put the artifacts into. In the "**Publishing repository**" field, select the "**Different Value**" button at the end and then type in

> **libs-snapshot-local**

5. Under "**Resolution Details**", set the "**Artifactory resolve server**" to the same value as the deployment server (if not already set):

> **http://localhost:8082/artifactory**

6. Since we will be resolving artifacts from our Artifactory system, we need to specify a repository to get the artifacts from. In the "**Resolution repository**" field, select the "**Different Value**" button at the end and then type in

> **libs-release**

7. Under the "**More Details**" section, make sure the following options are checked. (Note: The screen may jump around a bit as you click some of these.)

> **Capture and publish build info**
>
> **Include environment variables**
>
> **Allow promotion of non-staged builds**

Publish artifacts to Artifactory

Publish Maven descriptors

Use maven-compatible patterns

In the "**Exclude patterns**" field, type

*.jar

You can leave the other values as-is.

More Details

☐ Project uses the Artifactory Gradle Plugin

☑ Capture and publish build info

☑ Include environment variables

Include Patterns [ ]

Exclude Patterns [*password*,*secret*]

☑ Allow promotion of non-staged builds

☐ Allow push to Bintray for non-staged builds

☐ Run Artifactory license checks (requires Artifactory Pro)

☐ Run Black Duck Code Center compliance checks (requires Artifactory Pro)

☐ Discard old builds from Artifactory (requires Artifactory Pro)

☑ Publish artifacts to Artifactory

☑ Publish Maven descriptors

☐ Publish Ivy descriptors

☑ Use Maven compatible patterns

Ivy pattern [[organisation]/[module]/ivy-[revision].xml]

Artifact pattern [[organisation]/[module]/[revision]/[artifact]-[revision](-[classifier]).[ext]]

Include Patterns [ ]

Exclude Patterns [*.jar]

☑ Filter excluded artifacts from build Info

Deployment properties [ ]

☐ Enable isolated resolution for downstream builds (requires Artifactory Pro)

☐ Enabled Release Management

8. In the "**Build**" section, for the step to "**Invoke Gradle script**", add the following in the **Tasks** field:

build -x test install

(Note that we have the install target to cause the Maven plugin integration in Gradle to produce the pom.)

Build

Invoke Gradle script

◉ Invoke Gradle
Gradle Version

    gradle27

○ Use Gradle Wrapper
Build step description

Switches

Tasks

    build -x test install

9. **Save** your changes.

===

END OF LAB

===

Lab 8 - Setting up the Commit Stage of the Pipeline

Purpose: In this lab, we'll set up the pipeline view for the set of jobs we just finished configuring.

1. Change back to the jenkins dashboard if not already there (http://localhost:8080)

2. Above the list of jobs, on the line with the various tabs, click the "+" sign in the last tab.

Jobs that are part of the Workshop Pipeline

| (1) Workshop Pipeline View | (2) ws-review-stage | (4) ws-acceptance | (5) ws-deploy-production | (6) Reference Pipeline Jobs | (7) ref-review-stage | (8) ref-commit-stage | (9) ref-acceptance | 10 ref-deploy-production | All | + |
| S | W | Name ↓ | | | Last Success | | | Last Failure | | Last Duration |
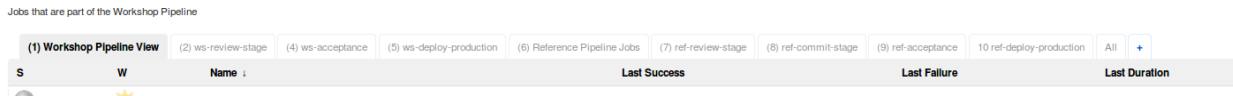
3. You'll now be on the configuration page for setting up the view. For the "**View Name**", you can type in

     (3) ws-commit-stage

(Note: The "(3)" here is optional but, if present, should sort the tab into the desired place in the list of tabs. This does not always work though.)

4. Select the button for **Build Pipeline View** and select **OK**.

5. On the next page, add a description in the "**Description**" field if you want.

6. Find the "**Build Pipeline View Title**" field and enter something like:

   Workshop Pipeline - Commit Stage

7. For "**Layout**", select "**Based on upstream/downstream relationships**".

8. Set the "**Select Initial Job**" to be

   ws-compile

9. The suggested value for "**No. of Displayed Builds**" is **1**.

10. Select "**Yes**" for the first two radio buttons: "**Restrict triggers to most recent successful builds**" and "**Always allow manual trigger on pipeline steps**"

© 2016 Brent Laster

11. Click on **OK** to save the changes.

12. Since these jobs have never been run, you will see a message like:

"This view has no jobs associated with it. You can either add some existing jobs to this view or create a new job in this view."

This is ok and will be resolved once the job is run.

===

END OF LAB

===
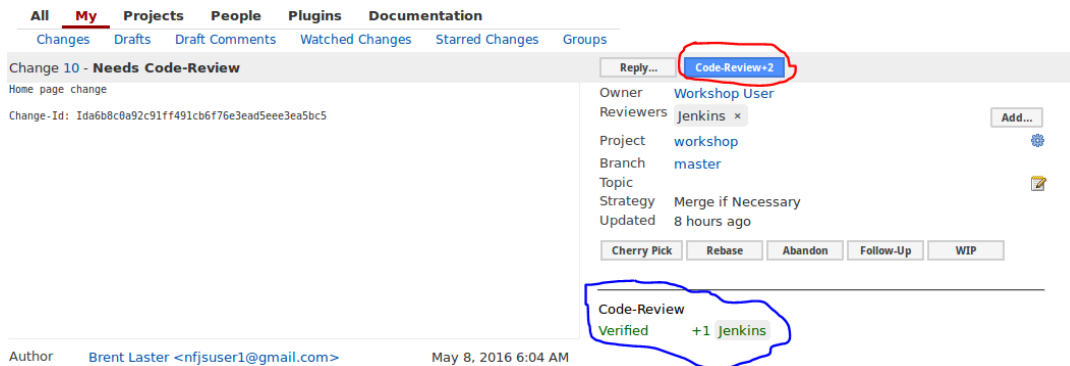

Lab 9 - Running a change through the Commit Stage

Purpose: In this lab, we'll review and submit our change in Gerrit and see it move through the Commit Stage of the pipeline.

1. Change back to the tab with your Gerrit instance or go to **http://localhost:8081**.

2. You should be running as "**Workshop User**" - that should be shown in the upper right corner.

3. If you are already on the page for your change (**http://localhost:8081/#/c/<number>**) then you are where you need to be. If you are on the homepage (**http://localhost:8081**), click on the single outgoing review you have. (Click on the Subject.)


4. You should now be on the page for the change. We need to review and approve this change.   For this case, we'll be the author of the change AND the reviewer (this is not typical).

5. First, let's look at the changes as we would do during a typical code review. Down around the middle of the page, click on the name of the file (**agents.html**).   This will bring up a screen to let you see the differences being put forward.

© 2016 Brent Laster

5. You'll see a screen that will look similar to the one below. For the sake of time, we won't do any commenting or further review here. After you look at the differences, click on the green "up arrow" in the upper right corner to get back to the previous page.

6. Back on the change screen, on the right center section, find the "**Verified +1 Jenkins**" line. This means that Jenkins did a test build (our **ws-verify** project) and it was successful (+1). We now need to complete the other validation - "**Code-Review**".

7. Gerrit provides a shortcut to approve a change - the blue button at the top.

Click on the "**Code Review+2**" button. You'll notice that the Code-Review check turns green and indicates that the Workshop User has approved this with a +2 score.

© 2016 Brent Laster

8. Now that we've "reviewed" the code change, and approved it (via pushing the button), you'll see that the "**Code-Review**" row now has a "**+2 Workshop User**" next to it. Since both **Code-Review** and **Verified** are green and have positive values, we are ready to tell Gerrit to try and merge it in to the repository.

To do this, press the **Submit** button.
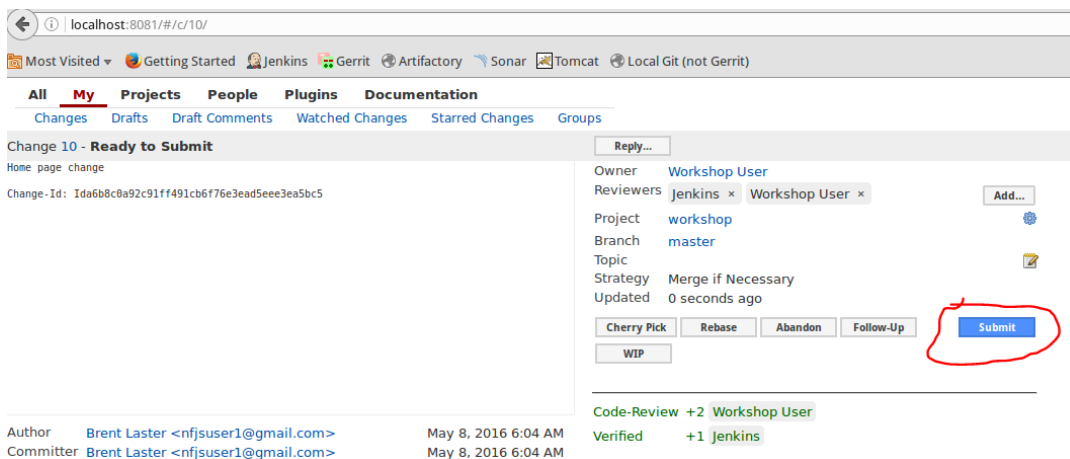
9. Notice that in the upper-left corner, this change's status has changed to "**Merged**". This means it has now been successfully merged into the remote Git repository by Gerrit. More importantly, it should have kicked off the **commit-stage** of our pipeline.

10. Switch back to the Jenkins dashboard. Click on the "(3) ws-commit-stage" tab at the top to see the commit-stage part of the pipeline.
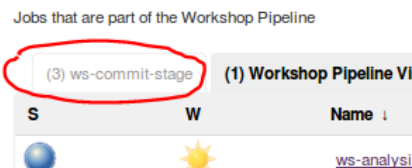

11. Depending on when you catch it in progress, you'll see the different stages running and turning yellow while running, and green when successfully done. **(Note: It will pause for a few seconds between each job.)**

© 2016 Brent Laster
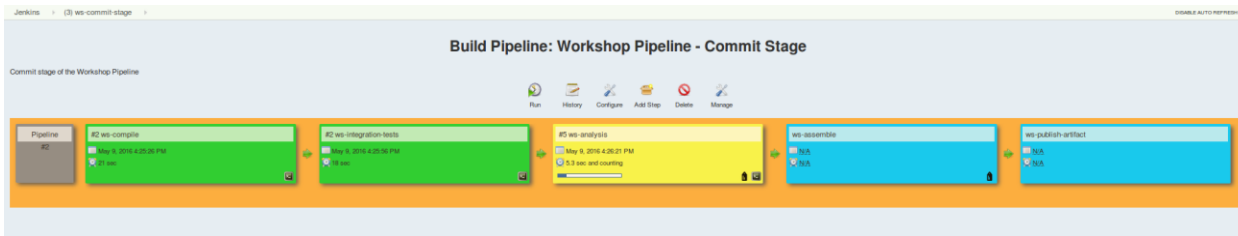
12. Once the commit-stage has completed successfully (all green boxes), you can look at a couple of interesting things. You can look at the integration test reports that Gradle produces in Jenkins' workspace. Open a new tab and click on the "Integration Test Reports" link in the bookmarks toolbar.
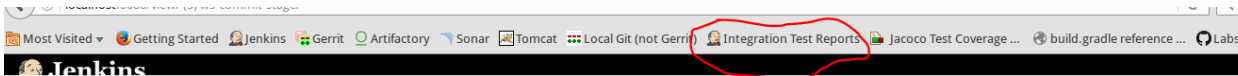
to see this:
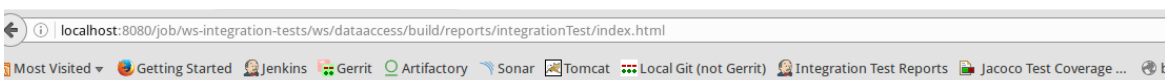
Test Summary

6	0	0	2.624s	**100%**
tests	failures	ignored	duration	successful

Packages   Classes

Packages

Package	Tests	Failures	Ignored	Duration	Success rate
com.demo.dao	6	0	0	2.624s	100%

Classes

Class	Tests	Failures	Ignored	Duration	Success rate
com.demo.dao.MyDataSourceTest	1	0	0	0.584s	100%
com.demo.dao.SchemaRegistryTest	5	0	0	2.040s	100%

Generated by Gradle 2.7 at May 9, 2016 4:26:14 PM

13. You can look at code coverage information for one of the sample projects by (opening a new tab and) clicking on the link in the bookmarks toolbar.
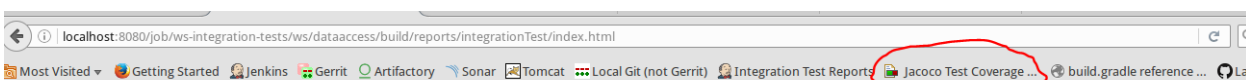
to see this:

util

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
⊕ com.demo.util		49%		56%	14	18	20	49	0	2	0	1
Total	132 of 257	49%	14 of 32	56%	14	18	20	49	0	2	0	1

14. You can see combined code coverage by going back to the Jenkins dashboard, clicking on the **ws-analysis** project, and then on the **Coverage Trend** link and then click on the big graph there.

15. You can see the **Sonar** report by (opening a new tab and) clicking on the shortcut in the toolbar (or going to **http://localhost:9000/sonar**).
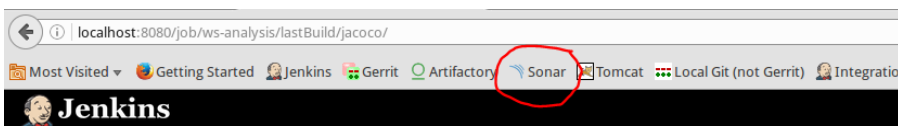
Thenk click on the **(Workshop) Pipeline Demo** project to drill in.

16. Finally, you can go to **Artifactory** and find your artifact by (opening a new tab and) clicking on "**Artifactory**" in the bookmarks toolbar (or going to **http://localhost:8082/artifactory**) and then drilling down.

The sequence is: **1.** Open **Artifactory**.   **2.** Click on the **Artifacts** link in the left column.   **3.** Expand the link for the **libs-snapshot-local** area and then expand the **com/demo/pipeline** area under that.   **4.** Expand the **web** area (for our web project).   **5.** Find the area for the artifact with the version numbers from the **ws-assemble** job.   By default this is **1.1.<build number>-SNAPSHOT**.   From there you can expand that and see the **war** file and the **pom**.

as shown in the figure below to find the latest artifact.

© 2016 Brent Laster

==
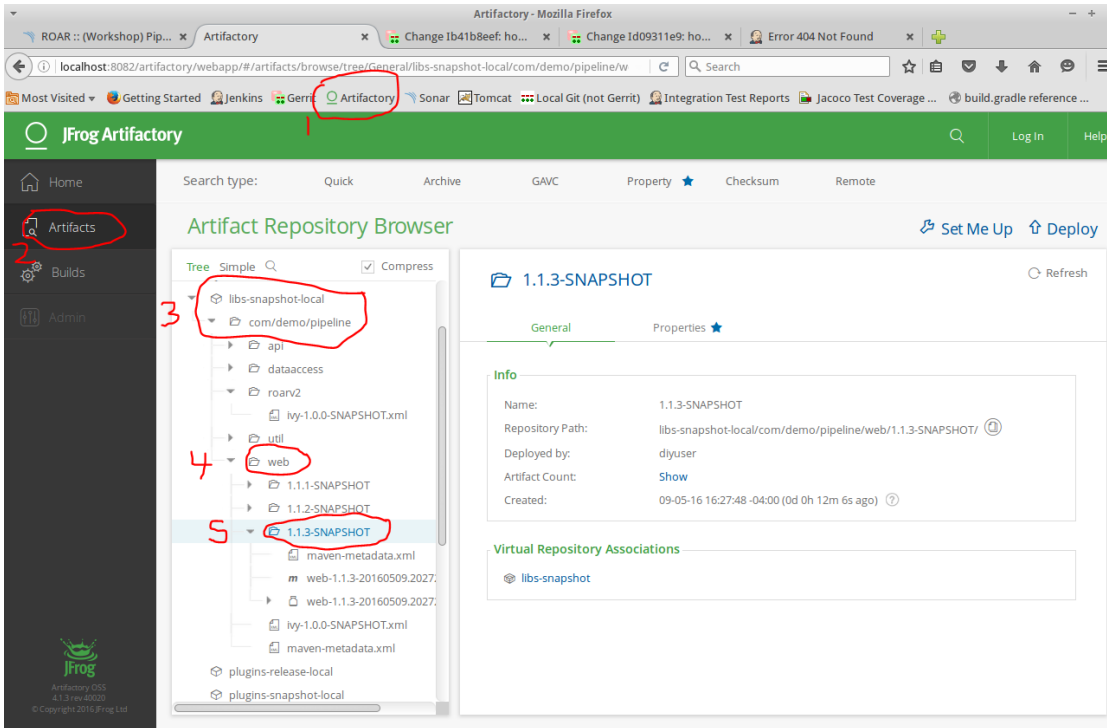
END OF LAB

==

Lab 10 - Creating the job to retrieve the latest artifact and running the acceptance stage

Purpose: The purpose of this job is to retrieve the latest artifact from Artifactory. There are easier ways to do this via Artifactory Pro if you buy it. Since we are going with the free version, we will use some scripting to examine the POM files and extract the latest. Once that is complete, we'll be able to watch the rest of the pipeline in action.

1. On the Jenkins dashboad (homepage), click on the **ws-retrieve-latest-artifact** job name and then on the **Configure** button again on the left-hand side.

2. Scroll down to the "**Build**" section.

3. Find the "**Execute shell**" command field (has this line in it - # insert commands from **ws-retrieve-latest-artifact-commands.txt** on the Desktop). Open up that file on the Desktop of your machine and copy and paste the commands into it. (It is not recommended that you type this one in since there is specific formatting involved.

© 2016 Brent Laster

Build

Execute shell

Command
```
# insert commands from ws-retrieve-latest-artifact-commands.txt on the Desktop
# remove any existing wars in workspace
if [ -e *.war ]; then rm *.war; fi

# Artifactory location
server=http://localhost:8082/artifactory
repo=libs-snapshot-local

# Maven artifact location
name=web
artifact=com/demo/pipeline/$name
path=$server/$repo/$artifact
version=`curl -s $path/maven-metadata.xml | grep latest | sed "s/.*<latest>\([^<]*\)<\/latest>.*/\1/"`
build=`curl -s $path/$version/maven-metadata.xml | grep '<value>' | sort -t- -k2,2nr | head -1 | sed "s/.*<value>\([^<]*\)<\/value>.*/\1/"`
war=$name-$build.war
url=$path/$version/$war

# Download
echo $url
wget -q -N $url
|
```

See the list of available environment variables

4. Add a **Post-build Action** to archive the artifacts.

Click on the "**Add post-build action**" button (near the bottom) and select "**Archive the artifacts**". (The reason we are doing this is to save off a copy of the artifact so that other jobs can use it in Jenkins.) A new step will appear in the **Post-build Actions** section. This should be the first step under the "**Post-build Actions**" section (before the "**Trigger parameterized builds on other projects**" step). If it is not, then drag the step by the handle up before the "Trigger…" step. The handle is shown in red in the figure below.

5. Fill in the "**Files to archive**" field with

   **/*.war

6. Click the "**Advanced**" button.

The boxes for "**Archive artifacts only if build is successful**" and "**Use default excludes**" should be checked.

Archive the artifacts

Files to archive   **/*.war

Excludes

☐ Do not fail build if archiving returns nothing

☑ Archive artifacts only if build is successful

☐ Fingerprint all archived artifacts

☑ Use default excludes

☐ Treat include and exclude patterns as case sensitive

7. After dragging the command up to the correct position (if needed), **save** your changes.

© 2016 Brent Laster

8. You should now be able to run the **ws-retrieve-latest-artifact job** (click on the **Build Now** link).

9. Once the build starts, go back to the Jenkins dashboard and click on the tab for the acceptance stage, "ws-acceptance".
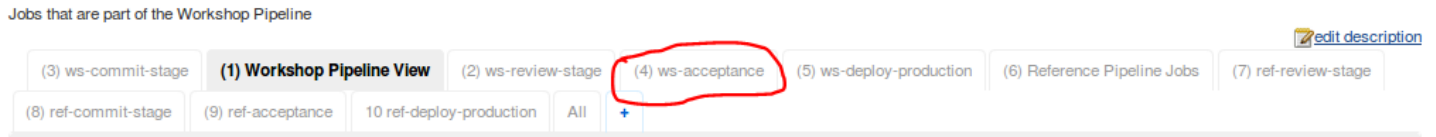
10. You should then see something like this once the jobs are completed.
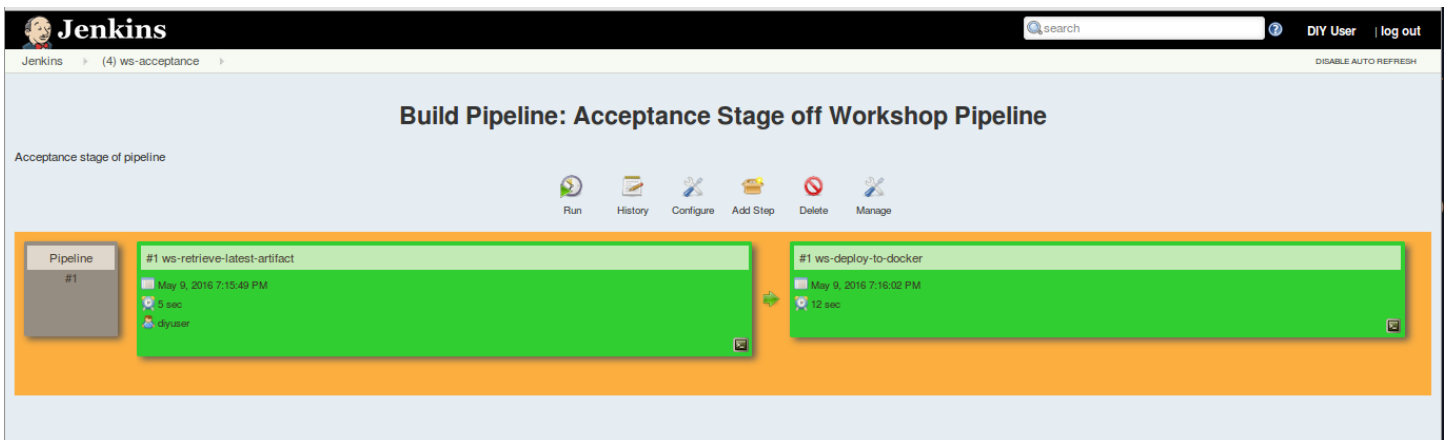
11. Now we want to look at the console log for the deploy-to-docker job. (We will discuss the contents of the job in the lecture.) But we need to disable the auto-refresh function so we can keep it open. In the upper right corner, click the link to "DISABLE AUTO REFRESH ". Then open up the console for the deploy-to-docker job by clicking on the small square in the lower right corner of the green rectangle. (See picture below - #2.)
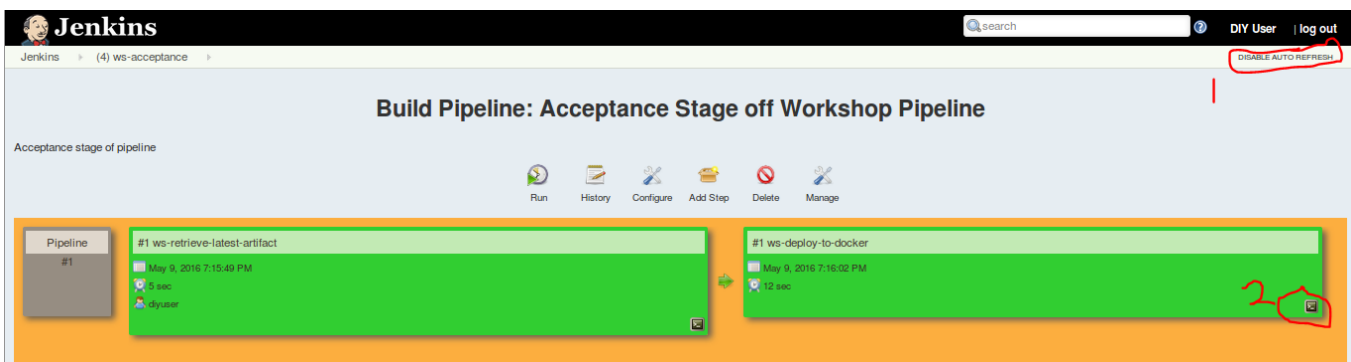
12. Clicking on that small square is a link to the console output for that job. For the **deploy-to-docker** job, we have a step that prints out a link to the docker container with the instance of our webapp running. Scroll to the bottom of the

© 2016 Brent Laster

console output. Find the link. Notice that this is running on a different ip address (different system - the docker container).
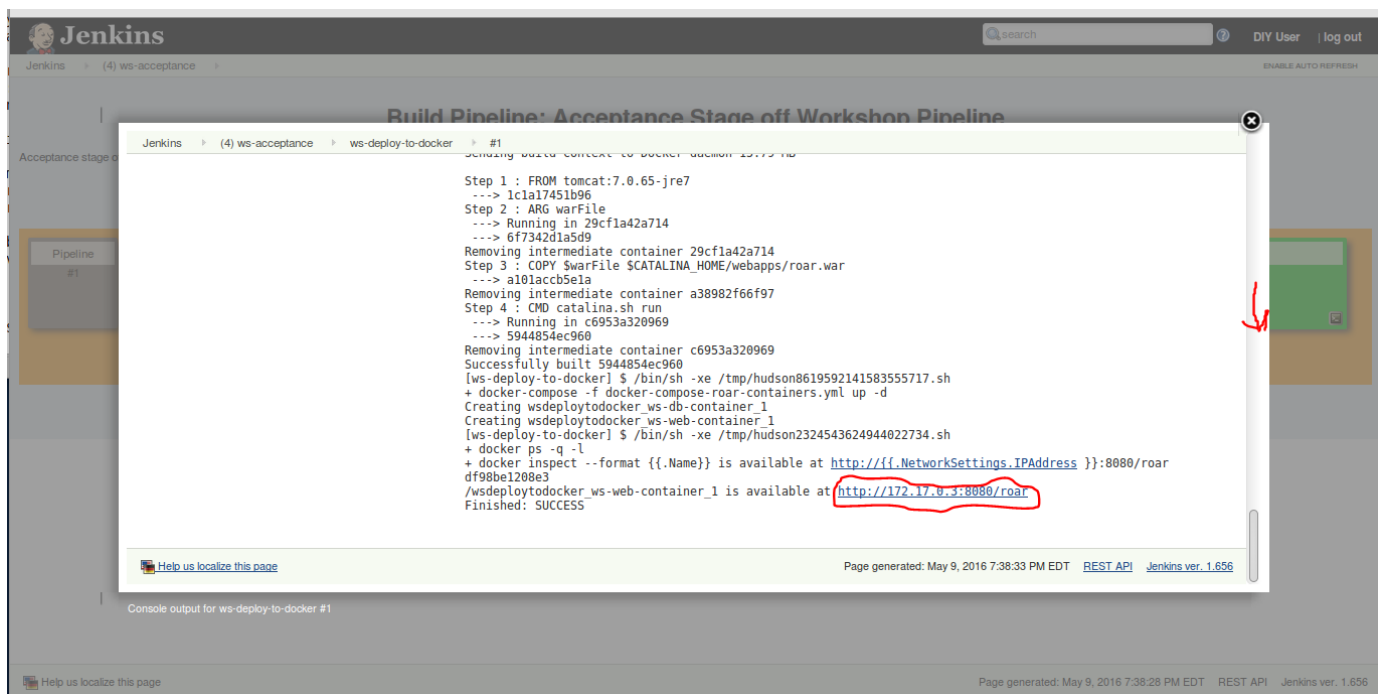
13. Open the web page on that container by copying and pasting the link into a new tab in the browser (or just click on it). This opens up a browser to the docker system running an image with our artifact deployed on it. (We'll discuss more about what's happening here in the lecture part of the workshop.)

==

END OF LAB

==

© 2016 Brent Laster

Lab 11 - Running the job to deploy the latest artifact to "production"

Purpose: The purpose of this job is simply to simulate deploying our artifact to a production system. In this case, production means a Tomcat instance running on our system.   We'll take a quick look at the job, fire it up and see the results - in the pipeline and the browser.

1. On the Jenkins dashboad (homepage), click on the **ws-deploy-artifact** job name and then on the **Configure** button again on the left-hand side.

2. Scroll down to the "**Build**" section.   The "interesting" parts of this one are the build step where we copy the artifact we archived from the **ws-retrieve-latest-artifact** job and then the **Post-build Action** step where we deploy our war file out to the Tomcat 7 container.   Nothing to change here but you can look at what's happening to see how this works.
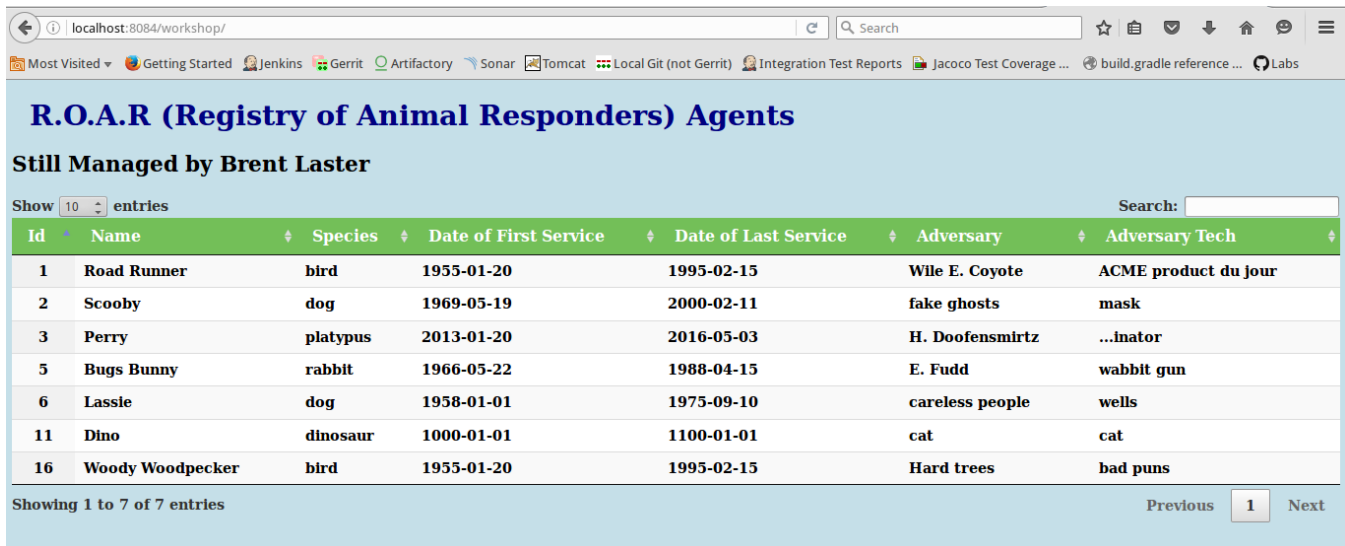
© 2016 Brent Laster

3. Finally, to "**deploy this to production**", you can click on the "**Build Now**" link in the upper left part of the page.   As the job is running, you can switch to the "**ws-deploy-production**" view and see that last stage running.
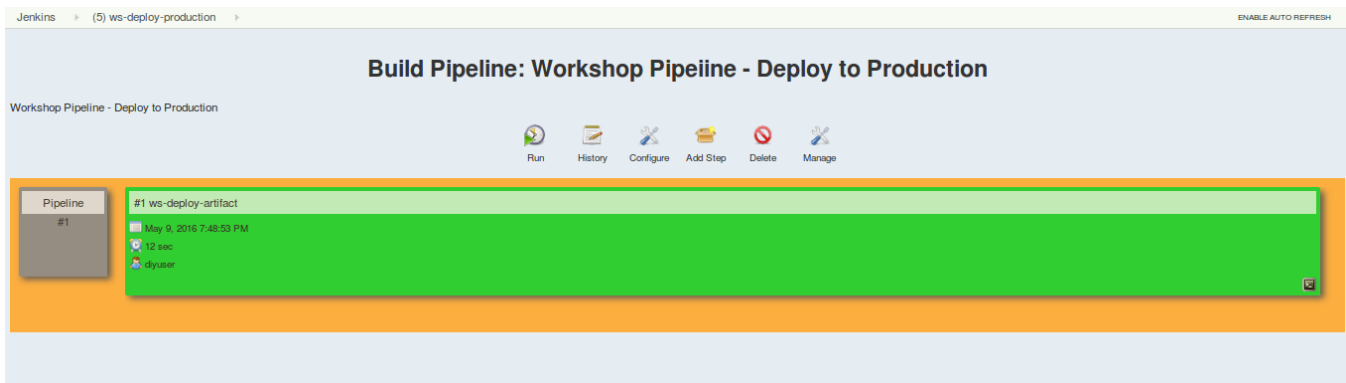
4. Now, you can open up the "**production**" website at **http://localhost:8084/workshop** and view the production instance of our webapp.

==

END OF LAB

==

OPTIONAL: Alternate approach for the ws-verify/ws-compile steps.

Purpose: If, for some reason, your submit and merge for the ws-compile step does not succeed, you can use an alternate job that does not go through Gerrit.   The ws-compile-without-gerrit-merge step is already setup to use. You would just push directly from the working directory and integrate it with the ws-commit-stage as outlined below.

© 2016 Brent Laster

1. On the Jenkins dashboard, click on the **ws-compile-without-gerrit-merge** job name. (If you don't see it, click on the "**All**" tab.)   This job is currently Disabled since it is setup to poll for SCM changes.   To use it, we will need to enable it.
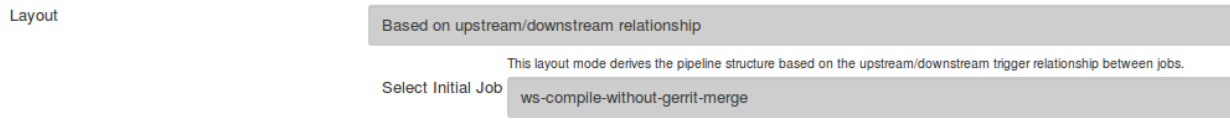
Click on the **Enable** button to do this.
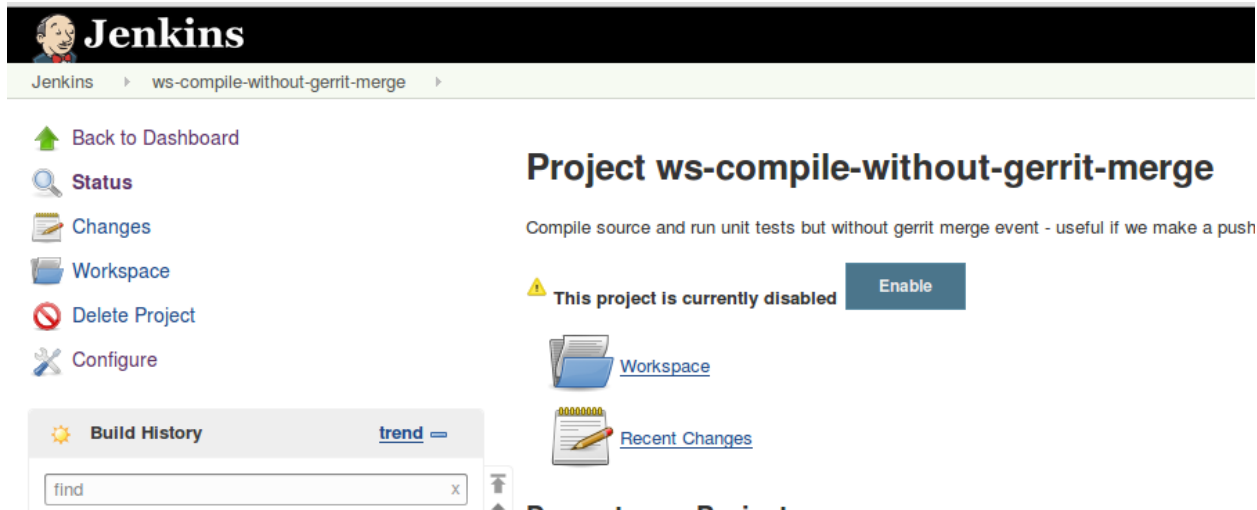
2. Now we need to modify our pipeline view to start with this job.   Switch back to the dashboard. On the top, in the list of views, click on the "**ws-commit-stage**" tab at the top.

3. Then, click on the "**Configure**" icon.

4. In the Layout section, find the "Select Initial Job" field and change it to use "ws-compile-without-gerrit-merge".

5. If you already have a terminal session open, switch to it. If not, open up a terminal session by either clicking on the "Terminal Emulator" shortcut on the desktop or by selecting the "Terminal Emulator" selection from the system menu (drop down from the mouse in the upper-lefthand side).
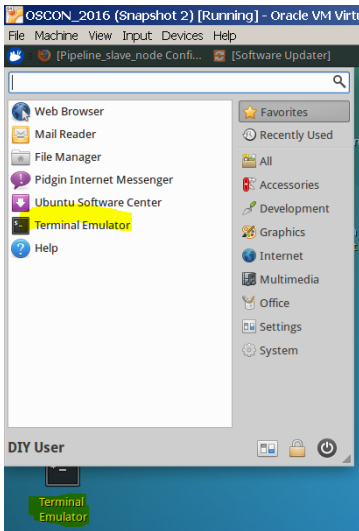
6. In the terminal session, if not already there, cd to the **workshop** directory.

     cd workshop

7. Under the "workshop" directory, in the "web/src/main/webapp" subdirectory, edit the "agents.html" file.

 (You can use mousepad or gedit editors.)

     gedit web/src/main/webapp/agents.html

Make some change to one of the header lines. Either

     `<h1>R.O.A.R (Registry of Animal Responders) Agents</h1>`

     `or`

     `<h2> Managed by ( your name here ) </h2>`

8.   Save your file (File->Save) and quit/exit the editor.

9.   <u>From the workshop directory</u>, do a local build to make sure things compile.

     gradle build

     This should finish with a "BUILD SUCCESSFUL" message.

10. Run the local instance to make sure the changes look as expected. Do this by running the jettyRun task (note case) and then opening a browser to **http://localhost:8086/com.demo.pipeline**.

     gradle jettyRun

(Note: This will get to some percentage and then start the system running. Don't kill this - it is working.)

© 2016 Brent Laster

```
:api:jar UP-TO-DATE
:web:compileJava UP-TO-DATE
:web:processResources UP-TO-DATE
:web:classes UP-TO-DATE
> Building 93% > :web:jettyRun > Running at http://localhost:8086/com.demo.pipe
```

<open in browser tab> **http://localhost:8086/com.demo.pipeline**

NOTE: note that the URL is pipeline on the end not just pipe as shown.

11.   Close the browser tab (not the entire session) and kill the running job in the terminal **(Ctrl-C) .**

12.   Now add and commit the change. This assumes again that you are still in the workshop directory. (Ignore any messages about line ending changes.)

   git add web/src/main/webapp/agents.html

   git commit -m "home page change"

13.   Now push the change to Gerrit. This will use the standard Gerrit syntax - not the syntax for Gerrit.

   git push origin master

==

END OF OPTIONAL LAB

==

NOTES ABOUT THE IMAGE:

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

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

Gerrit: Running on http://localhost:8081 Just switch to desired user: Workshop User or Local Admin through Become link

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

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

Artifactory: Running on localhost:8082/artifactory login: admin password: admin

© 2016 Brent Laster