Introduction to Continuous Integration

Revision 1.0 – 9/17/18

Brent Laster

IMPORTANT NOTES:

1. You should already have your VM image up and running per the setup doc.

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

3. Highlighted text is a description of what the solution will contain. Feel free to try and come up with the solution before looking at the answers.

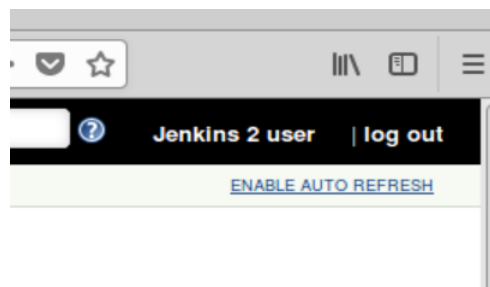**Lab 1 starts on the next page!**
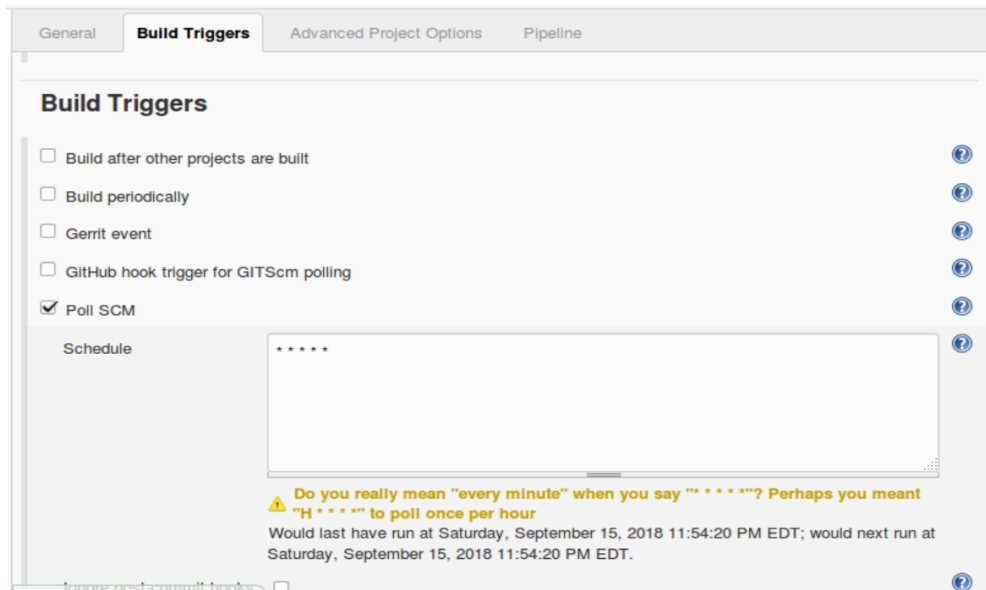
© 2018 Brent Laster
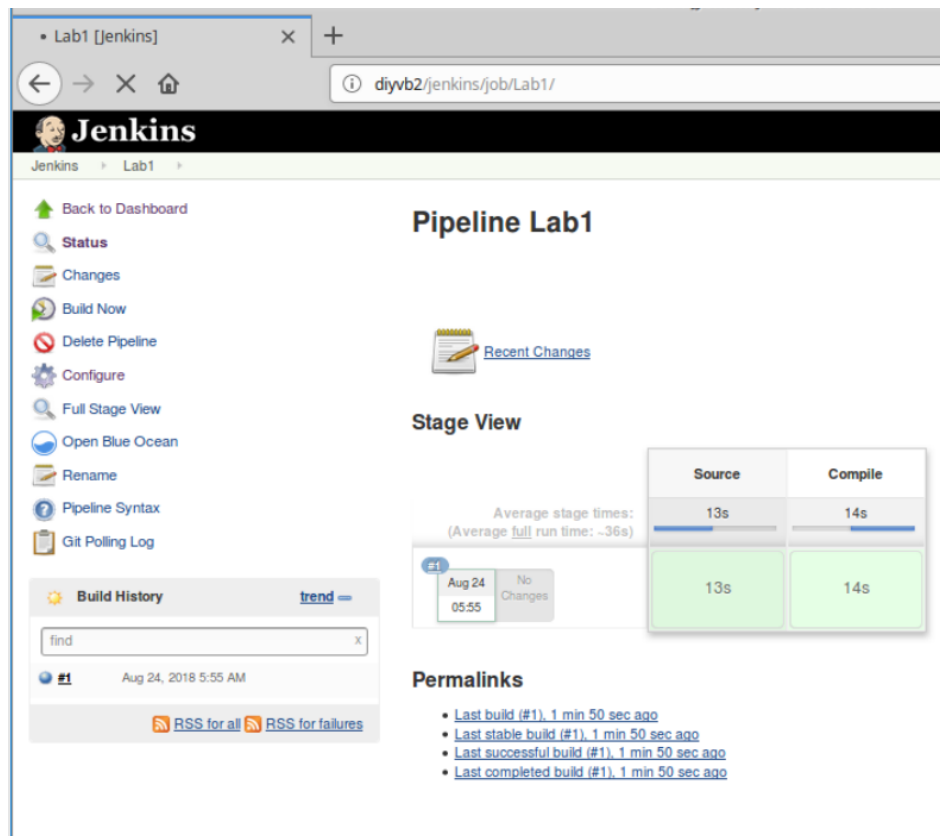
Lab 1 – First CI Pipeline

Purpose: In this lab, we'll get a quick start learning about CI by setting up a simple Jenkins pipeline job to build a project from our local repository on the VM.

1. If you haven't already, start up the CI virtual machine.

2. On the VM desktop, click on the Jenkins icon on the desktop and login to Jenkins with the userid and password supplied in the class.

3. Once logged in, click on New Item in the left menu. On the next screen, enter a name for the project such as "lab1" and select a project type of "Pipeline". Then select OK.
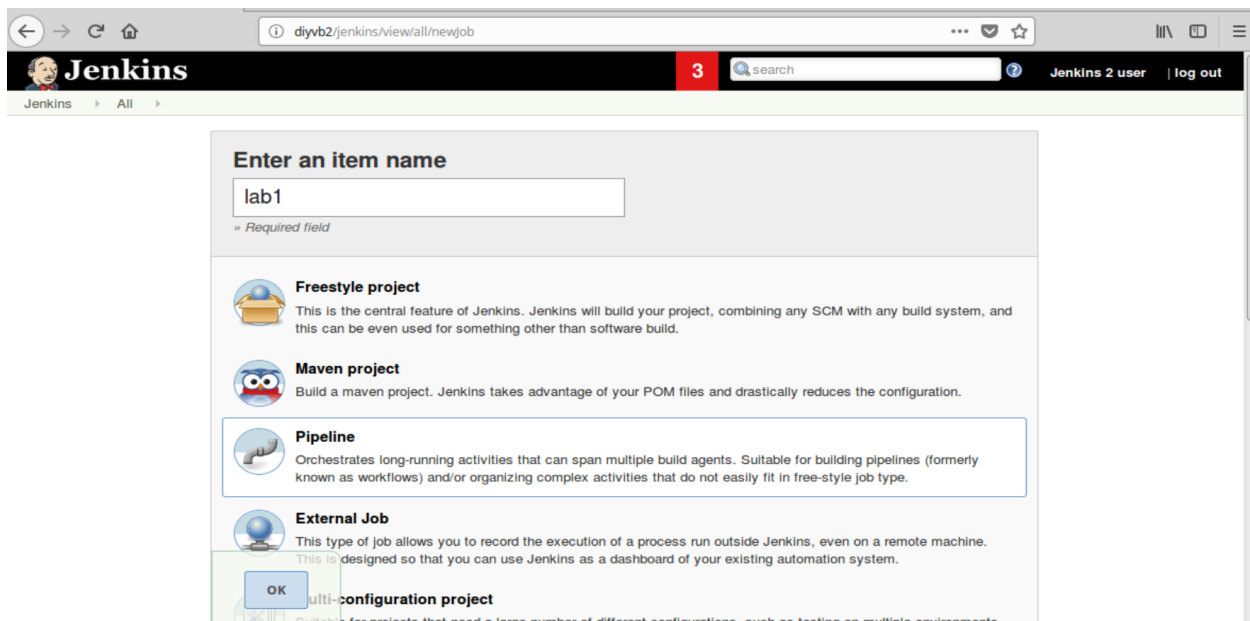
4. First, we will want to define what kind of events will trigger this build. For our purposes here, we'll just scan the source repository periodically. Scroll down to the "Build Triggers" section and select "Poll SCM".

5. This will open up a text box labeled "Schedule". This is where we'll put in the representation for how often to scan the repository. Though we wouldn't normally do this in production, we'll tell it to scan every minute. In the field that pops up, enter * * * * *. (This is five asterisks separated by spaces.) This is short for "scan every minute of every day of the week of every day of every year. You can ignore the yellow warning message.

6. With the build trigger setup, we can now put our pipeline in place to retrieve the source code and do a build when the build is triggered by a change in the area we are scanning for. The pipeline code you need is already saved in a file named "lab-1.txt" on the desktop. Click on the icon for this file to open it. Then copy and paste the code from the file into the pipeline area "Script" box in your job.

7. Notice here that we have a source management stage with a "Source" statement to get the code from our remote repository and then a "Compile" stage with a call to the gradle build tool to attempt to build our code.

8. Save your changes to the pipeline code by clicking the "Save" button at the bottom of the screen. Click the "ENABLE AUTO REFRESH" link in the upper right corner. After a minute or two, Jenkins should try to automatically build your project.

© 2018 Brent Laster

9. Open a "Terminal Emulator" session (there is a shortcut on the desktop for that.)

10. We already have a copy of the repository referenced in our pipeline cloned down locally. In the terminal window change into the directory with that cloned copy.

    cd helloWorkshop

11. Open up the helloWorkshop.java file and make a simple change for it.

    gedit src/main/java/helloWorkshop.java

12. For example you can add a line like the following under the System.out… line

    System.out.println("Hello user!");

© 2018 Brent Laster

```
Open  ▼      +                          *helloWorkshop.java
                                        ~/helloWorkshop/src/main/java

File   Edit   View   Search   Tools   Documents   Help
/* Hello Workshop Java Program */
/* 8-12-18 */
class helloWorkshop  {
    public static void main(String[] args) {
        System.out.println("Hello CI Workshop!");
        System.out.println("Hello user");
    }
}
```

13. Save your changes and quit/exit the editor.

14. Before we push our code, we want to make sure that it builds cleanly in our local environment first. Back in the terminal window, run the gradle build command.

    gradle build

    After a moment, you should see Gradle start up and run through the tasks to build the project. At the end, you should see a "BUILD SUCCESSFUL" message. If not, go back and check your typing in the file.

15. Now that we've made a change and verified that it works in our working directory, we can push it over into the repository. We'll use the following Git commands to do this.

    git commit -am "Add a line"

    git push

16. Now, switch back to the stage view page of the Jenkins job (http://diyvb2/jenkins/job/Lab1/. After a minute or two, Jenkins will detect your change and build the project.   You can see how it polled the SCM by clicking on the "Git Polling Log" link in the left menu.

Pipeline Lab1
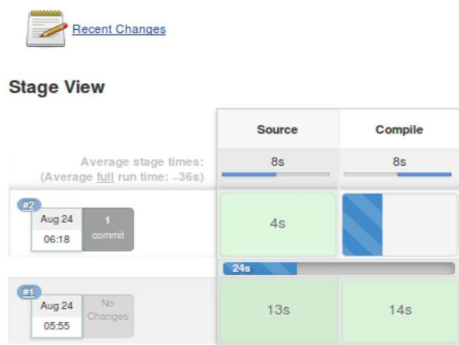
17. After it builds, you can click on the blue ball next to #2 in the "Build History" window to see the console output where the project was built.   Any further changes would be incorporated and built the same way.

© 2018 Brent Laster
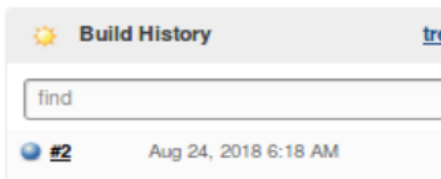
Lab 2 - Using pre-flight checks with Gerrit

In this lab, we'll see how to use pre-checks (such as code reviews and verification builds) through a tool called Gerrit. You'll set up a Gerrit project in Jenkins and then push a change through Gerrit to see it pass.

1. In Jenkins, go back to the Dashboard by clicking on the "Back to Dashboard" link in the upper left or entering "diyvb2/jenkins" in the address bar in the browser.

2. We're going to create a new job for accessing the code from our Gerrit repository and doing a verification build before it gets completely merged in. This job will start out as a copy of our existing job. Start out by clicking on "New Item" in the left menu.

3. Enter a name for the job, such as "lab2". Then scroll down to the bottom of the screen until you see the "Copy from" area. In the text area, enter the name of your first job (i.e. "lab1") and click "OK".
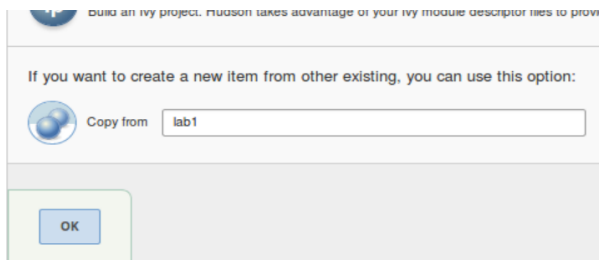
4. You should now be in the "Configure" screen for your new job. Scroll down to the "Build Triggers" section of the configuration.
5. We want this CI build to be initiated off of a change pushed to Gerrit. So, unclick the "Poll SCM" option and click the "Gerrit event" box here.
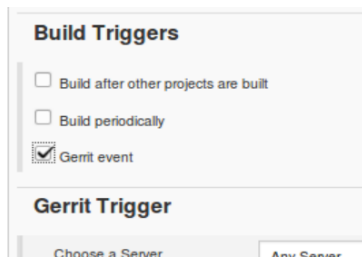
6. A new section will open up on the page for "Gerrit Trigger". First we need to select the server we already have configured on our system. In the "Choose a Server" section, click and then select the "diyvb2" entry.

© 2018 Brent Laster

7. We want to trigger the build when a new revision (patchset) is created in Gerrit. In the "Trigger on" section, click on the "Add" button and select the "Patchset Created" entry.
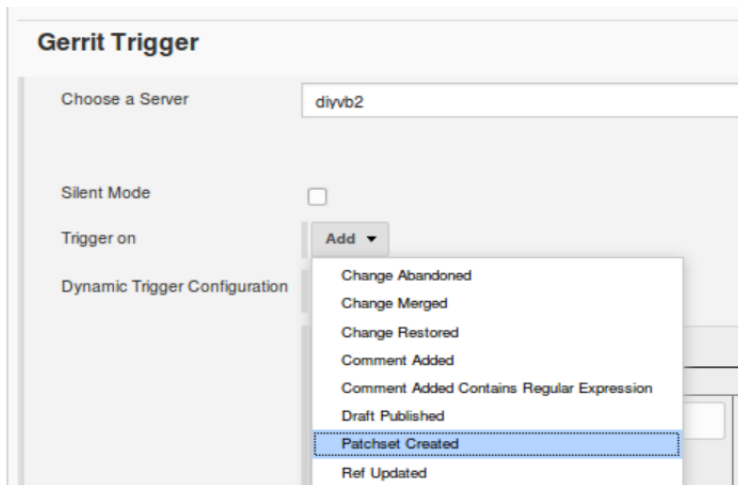
8. Now we need to point Jenkins to the Gerrit Project we want to use. In the "Gerrit Project" section, in the left block, select "Type" of "Plain" and for "Pattern", put in "helloWorkshop". Under the "Branches" section, for "Type" select "Path" and for "Pattern", put in "**".

9. You configuration should look like this for the "Gerrit Project" section.
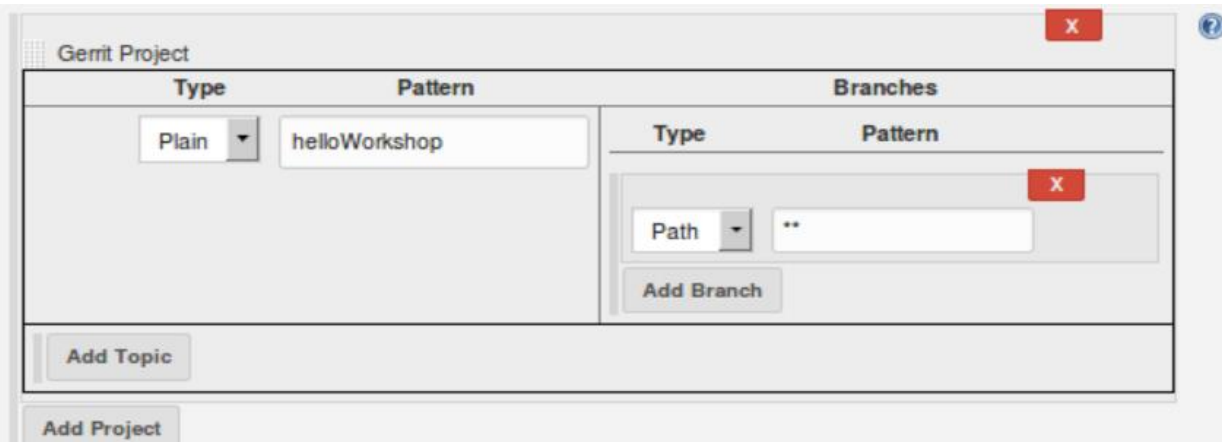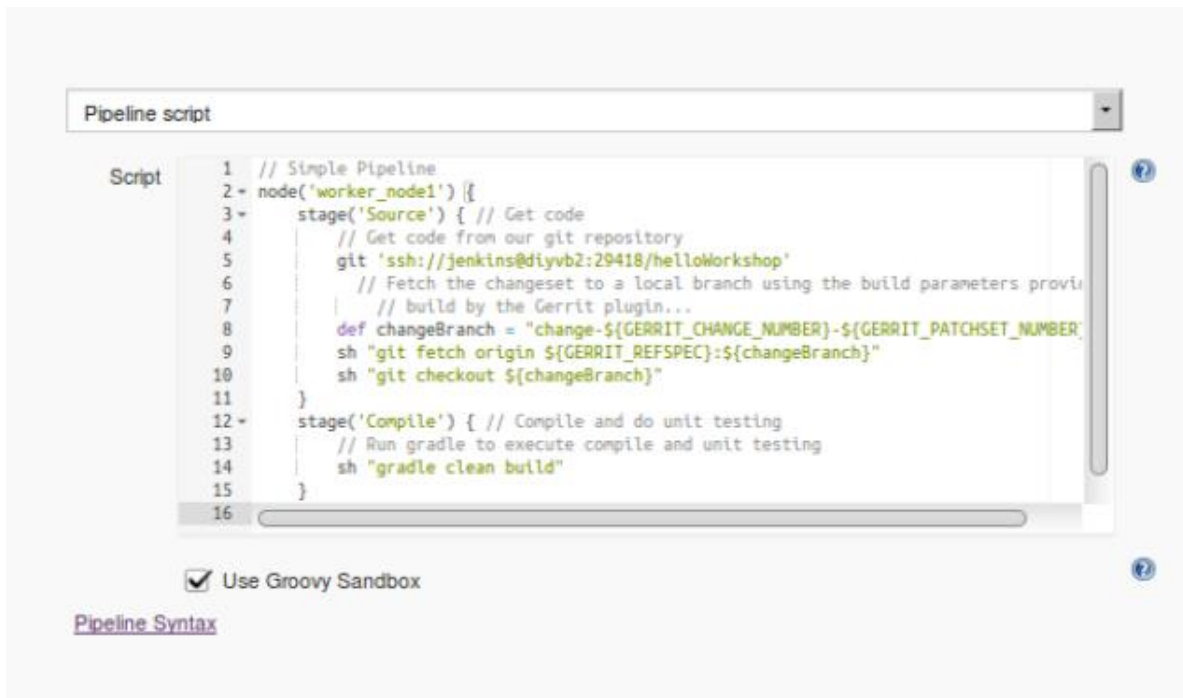
10. With the build trigger setup, we can now add the needed steps into our pipeline to get the change from Gerrit. The extra pipeline code you need is already saved in a file named "lab-2.txt" on the desktop. Click on the icon for this file to open it. Then copy and paste the code from that file into the pipeline "Script" box in your job. Put it under the "git" step. The code to add is shown in bold below.

```
stage('Source') { // Get code
    // Get code from our git repository
    git 'ssh://jenkins@diyvb2:29418/helloWorkshop'
    // Fetch the changeset to a local branch using the build parameters
    //provided to the build by the Gerrit plugin...
    def changeBranch = "change-${GERRIT_CHANGE_NUMBER}-${GERRIT_PATCHSET_NUMBER}"
    sh "git fetch origin ${GERRIT_REFSPEC}:${changeBranch}"
    sh "git checkout ${changeBranch}"
}
```

These extra "git fetch" lines after the main git call are needed to get the code from the intermediate place that Gerrit is holding it. The items starting with "${GERRIT" are references to values that Gerrit passes to Jenkins through environment variables.

11.   Afterwards, your pipeline script should look like the figure below:

Pipeline script

```
Script    1  // Simple Pipeline
          2 ▾ node('worker_node1') {
          3 ▾     stage('Source') { // Get code
          4             // Get code from our git repository
          5             git 'ssh://jenkins@diyvb2:29418/helloWorkshop'
          6             // Fetch the changeset to a local branch using the build parameters provi
          7             // build by the Gerrit plugin...
          8             def changeBranch = "change-${GERRIT_CHANGE_NUMBER}-${GERRIT_PATCHSET_NUMBER
          9             sh "git fetch origin ${GERRIT_REFSPEC}:${changeBranch}"
         10             sh "git checkout ${changeBranch}"
         11         }
         12 ▾     stage('Compile') { // Compile and do unit testing
         13             // Run gradle to execute compile and unit testing
         14             sh "gradle clean build"
         15         }
         16
```

☑ Use Groovy Sandbox

Pipeline Syntax

12. Now, we are going to push a change into Gerrit and watch it kick off a verification build.   Switch back to the terminal window and change to the copy of the project in the "gerrit-wd" working directory.

> cd ~/gerrit-wd/helloWorkshop

13. Open up the helloWorkshop.java file and make a simple change for it that will break the build.   For example you can add a line like the following:

> gedit src/main/java/helloWorkshop.java

> Add a line without a semicolon on the end under the last System.out… line like

> > System.out.println("Hello again")

> Save your changes and quit the editor.

14.   Now, stage and commit the change.

> git commit -am "update 2"

© 2018 Brent Laster

15. Set your environment to be able to use git review.

    git config --global gitreview.remote origin

16. Push your change to Gerrit with the git review tool.

    git review

17. Click on the icon to open up Gerrit or open a new tab in your browser and go to http://diyvb2/gerrit. Userid = "diyuser2" and password is the same. After a few moments, your change should show up on the Gerrit dashboard.
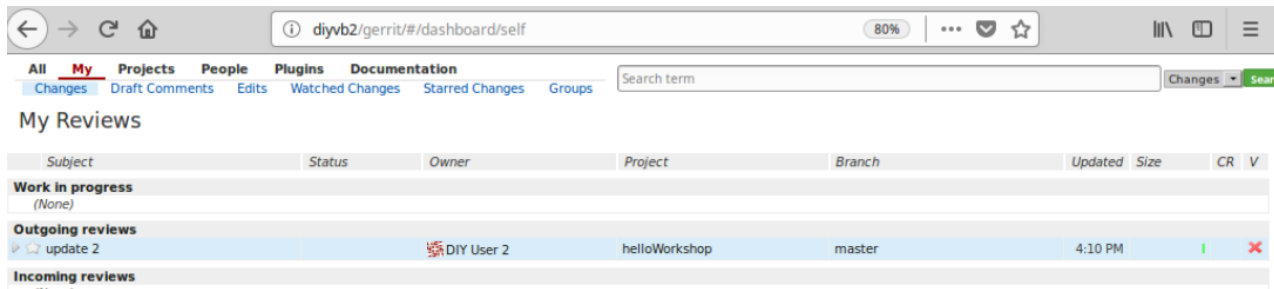

18. Notice the red "x" on the far right. (If you don't see it refresh the screen.) This is an indication that our Jenkins verification build failed. Switch back to Jenkins and take a look at the latest output for lab 2. You can see that the most recent build failed.
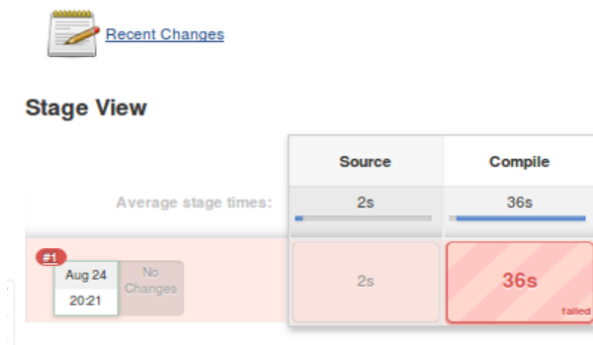
19. Click on the red ball next to the latest run to see the console output. Notice that it was triggered by a new patchset pushed to Gerrit.
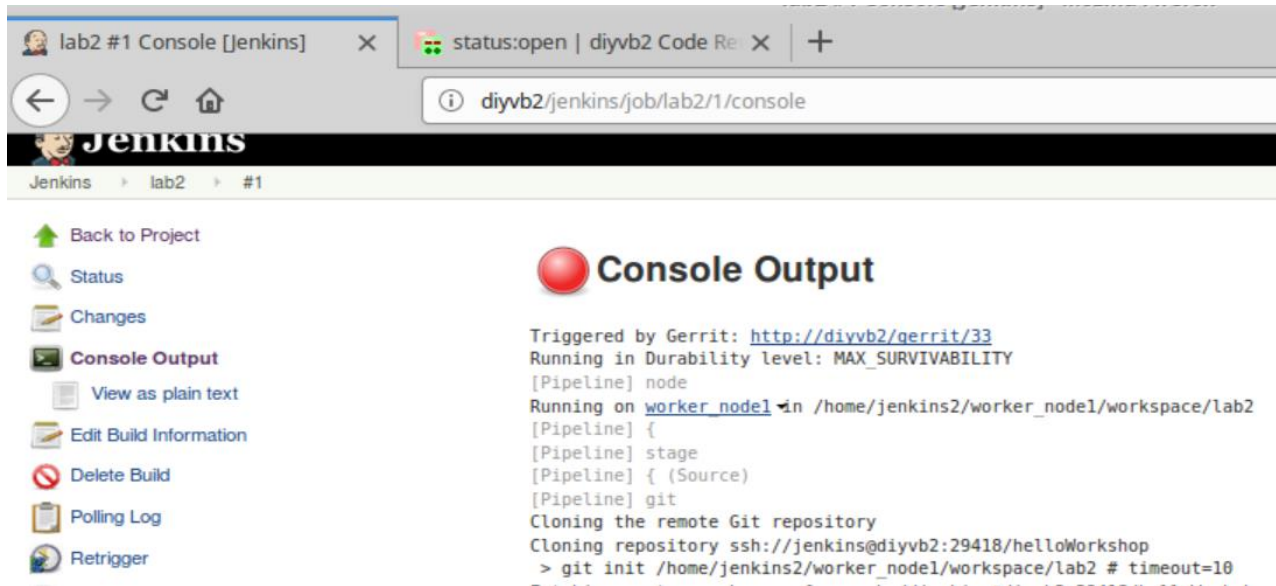
Lab 3. Further CI with Gerrit

In this lab, we'll see how to correct our failed verification build, how to setup a job to build our change after it is merged, and how to complete the Gerrit lifecycle by merging our code into its Git repository.

1. Switch back to the browser tab with Gerrit open in it. Open the outgoing review from lab 2 by clicking under the "Subject" field in the line.
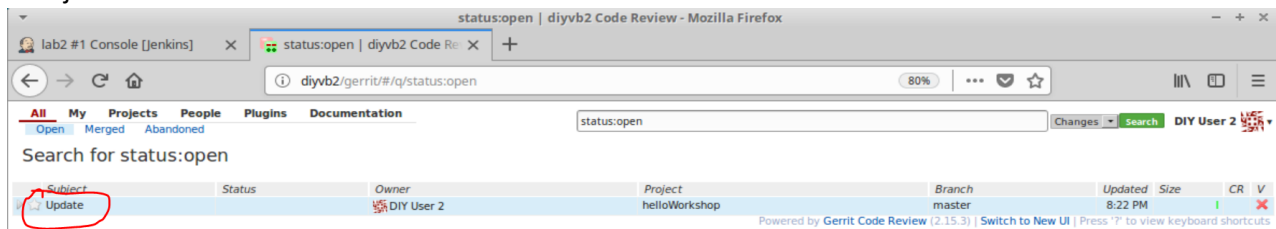
2. Notice in the upper right section, there is a line that says "Verified -1". This is telling you that Jenkins ran a test build and it failed.

© 2018 Brent Laster
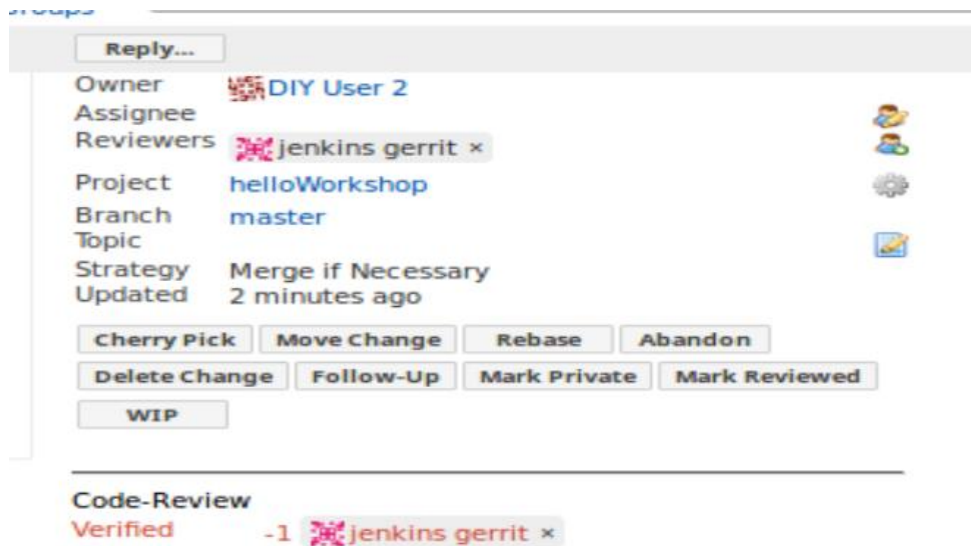
If you look at the bottom of the screen, you'll see more log output from Gerrit where the Jenkins build started and finished. The links in here actually work to take you to the Jenkins job.

3. Let's fix our error. Go to the terminal session. Open up the helloWorkshop.java file and add the missing semicolon.

    gedit src/main/java/helloWorkshop.java

        System.out.println("Hello user");

4. Now we need to update our change in Gerrit with a new patchset (revision). To do that, we first need to put this change in the staging area.

    git add .

5. Now we need to commit it, but we need to do that with the --amend option so it changes the same changeset instead of creating a new one.

    git commit --amend

    This will open up the default editor in case you want to update the commit message. You can just change "update" to "update 2" or something like that at the top.

    Then hit Ctrl-O to tell the editor to write the file out. Hit return for the filename. Then hit Ctrl-X to exit the editor.

© 2018 Brent Laster

6. This has put the change in our local Git repository. Now we need to push it over to Gerrit. We can just use Git review again.

> git review

7. After a minute or so, the updated change should trigger a new build in Jenkins. You can switch back to the Jenkins session and watch the new build in the browser. It should be successful.

8. Gerrit should also be updated after the build. You should see a "+1" value in the Verified section now indicating the test build was successful. (You may need to refresh the screen to see the change.)

9. Our verification job for Gerrit was successful. Let's create a job to build the code after it is merged. In Jenkins, go back to the Dashboard by clicking on the "Back to Dashboard" link in the upper left or entering "diyvb2/jenkins" in the address bar in the browser.

10. We're going to create a new job for accessing the code from our Gerrit repository and doing a verification build after the code is merged in. This job will start out as a copy of our existing job. Start out by clicking on "New Item" in the left menu.

11. Enter a name for the job, such as "lab3". Then scroll down to the bottom of the screen until you see the "Copy from" area. In the text area, enter the name of your second job (i.e. "lab 2") and click "OK".

12. You should now be in the "Configure" screen for your new job. Scroll down to the "Build Triggers" section of the configuration.

13. In the "Gerrit Trigger" section, we want to change from triggering on a new patchset revision being created to triggering on our code actually being merged.

Click on the red "x" to get rid of the "Patchset Created" trigger.
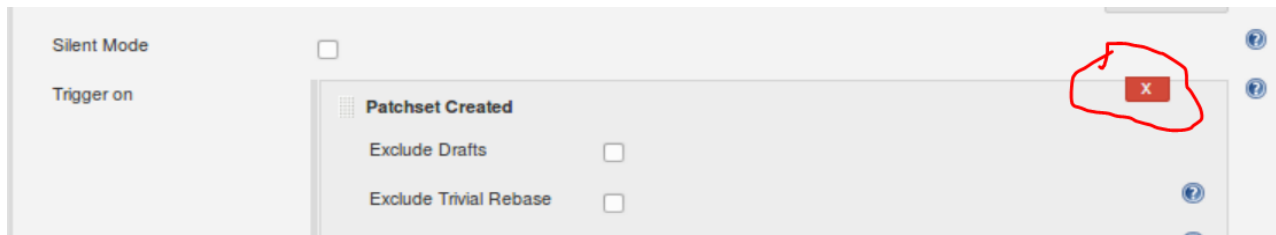
Click on the Add button and select "change merged".
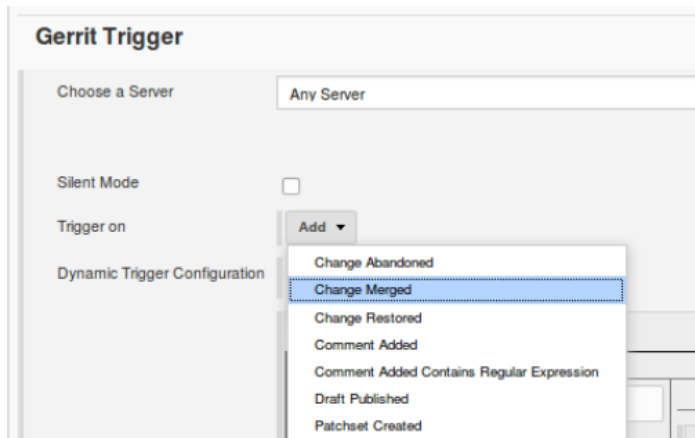
14. Scroll down to the pipeline section and remove all the extra statements in the "Source" stage beyond the initial "git" call. So remove the lines you added in lab 2.

```
// Fetch the changeset to a local branch using the build parameters provided to the
 // build by the Gerrit plugin...
def changeBranch = "change-${GERRIT_CHANGE_NUMBER}-${GERRIT_PATCHSET_NUMBER}"
sh "git fetch origin ${GERRIT_REFSPEC}:${changeBranch}"
sh "git checkout ${changeBranch}"
```

Afterwards, your pipeline will look like this:

© 2018 Brent Laster

```
Pipeline script                                                          ▾

Script      1  // Simple Pipeline
            2 ▾ node('worker_node1') {
            3 ▾     stage('Source') { // Get code
            4            // Get code from our git repository
            5            git 'ssh://jenkins@diyvb2:29418/helloWorkshop'
            6        |
            7        }
            8 ▾     stage('Compile') { // Compile and do unit testing
            9            // Run gradle to execute compile and unit testing
           10            sh "gradle clean build"
           11        }
           12 }

            ✔ Use Groovy Sandbox
Pipeline Syntax
```
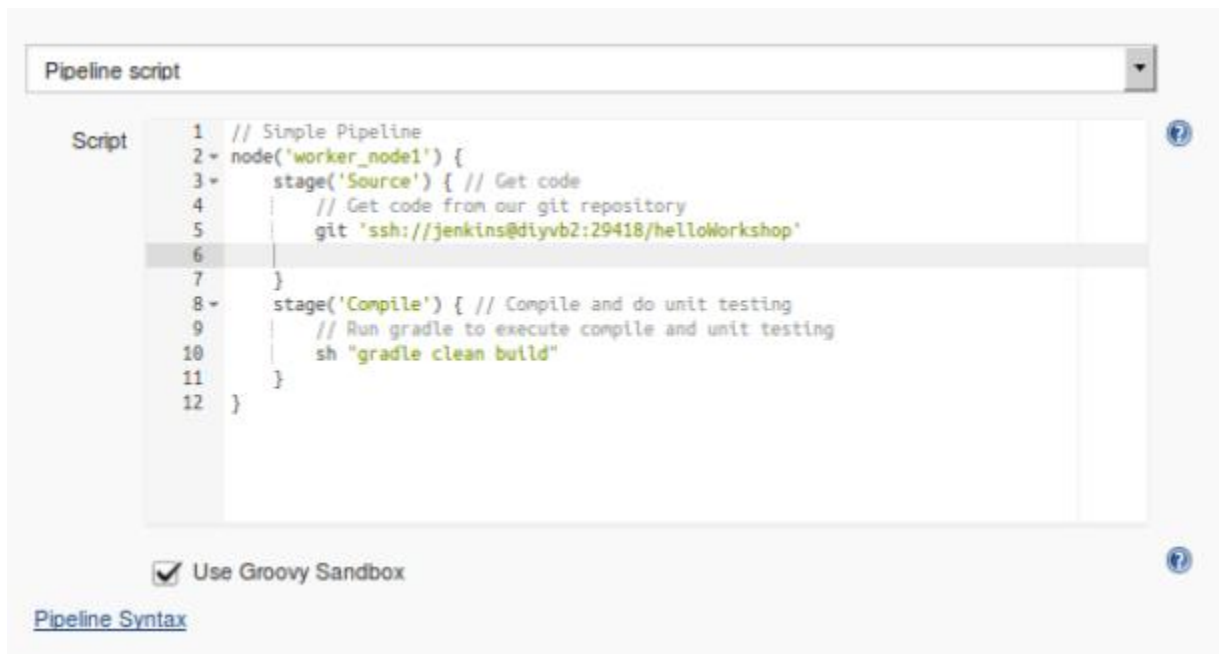
15. Save your changes.
16. Now that we have our job ready to build our code when we merge it, let's finish the review process. Switch back to the browser tab with Gerrit. You should still be on the screen for your change.

17. Notice that above the "Verified +1" section is another category for "Code Review". This is waiting for a review and approval. In this case, we'll be our own reviewer.   In the middle of the change screen, find the file listing and click on "src/main/java/helloWorkshop.java".
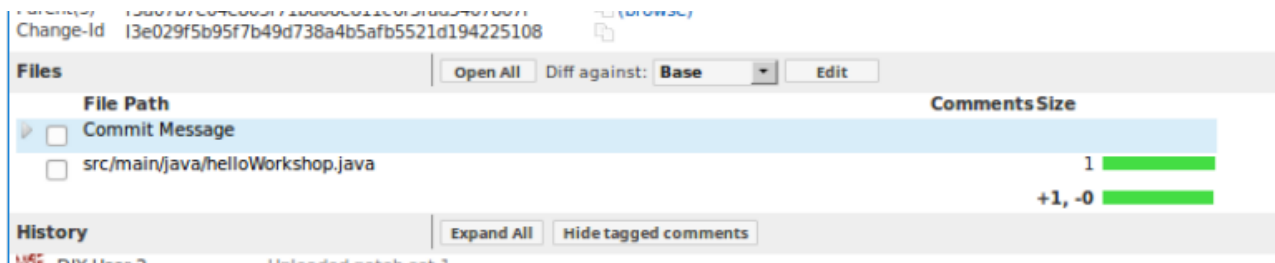
18. This will open up a side-by-side comparison between the previous version on the left and the current version on the right. If we wanted to, we could make comments in here about the changes. For our purposes here, you can just go back to the change by clicking on the green up arrow in the upper left part of the screen.

© 2018 Brent Laster
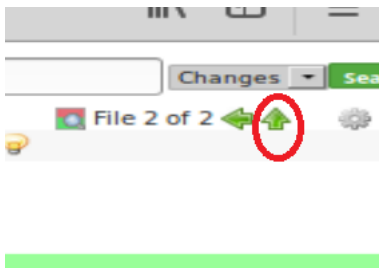
19. Now that we have reviewed the change, we can indicate our approval by giving it a "+2" value. To do this, click on the "Code-Review+2" shortcut button at the top of the screen.

20. After clicking on that button, a new "Submit" button will appear on the screen that you can use to merge the code. Click on the "Submit" button to complete the cycle.

21. After the Submit button is pushed, the status of the change in the upper left should show "Merged".
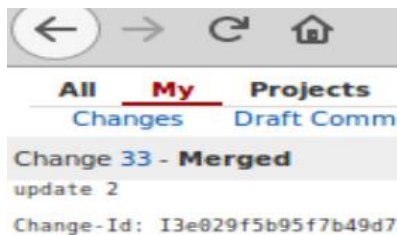
22. At this point, lab 3 should be kicked off and build the merged change. You can view the results in Jenkins.

Lab 4. Working with GitHub and Multibranch Pipelines

In this lab, we'll see how to contribute to projects on GitHub and utilize Jenkins for another form of Continuous Integration.

1. To start, sign in to your GitHub account on http://github.com.
2. Once logged in, in the address bar, enter http://github.com/brentlaster/filedump to get to the project we will be working from in this lab.
3. After changing to that project, click the "Fork" button at the top of the page to make your own copy of the repository in your GitHub namespace.
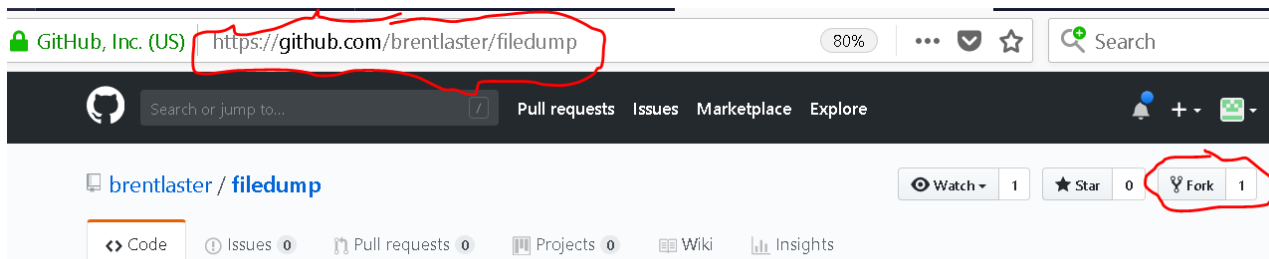
When this operation finishes, you'll have your own copy of this project in your GitHub namespace.

© 2018 Brent Laster

4. Take a look around the project in the GitHub interface. Notice that we have a Jenkinsfile here that defines our pipeline. We first need to create credentials in Jenkins that we can use to access GitHub. Start out by going back to the Jenkins dashboard and selecting "Credentials" in the left menu. Then, click on "System" under the Credentials item in the left menu.

5. On the next screen, click on the "CDPipeline" domain.

6. On the next screen, click on "Add Credentials" in the left menu.
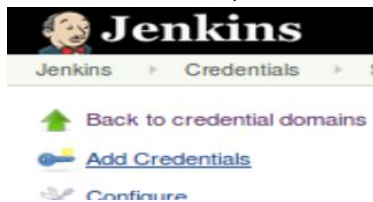
7. Next, select the credential type of "Username with password" and fill in the information with your GitHub userid and password. The ID and Description fields can be something like "GitHub". Click OK when done.

© 2018 Brent Laster

8. Now, we'll create a new project to build our code based on the Jenkinsfile in GitHub. Go back to the Jenkins dashboard and select "New Item". Enter a name for the new project, such as "lab4" and select a type of "Multibranch Pipeline" and click on "OK".

9. In the configuration for the new job, scroll down to the "Branch Sources" section, click the "Add" button, and select "GitHub".
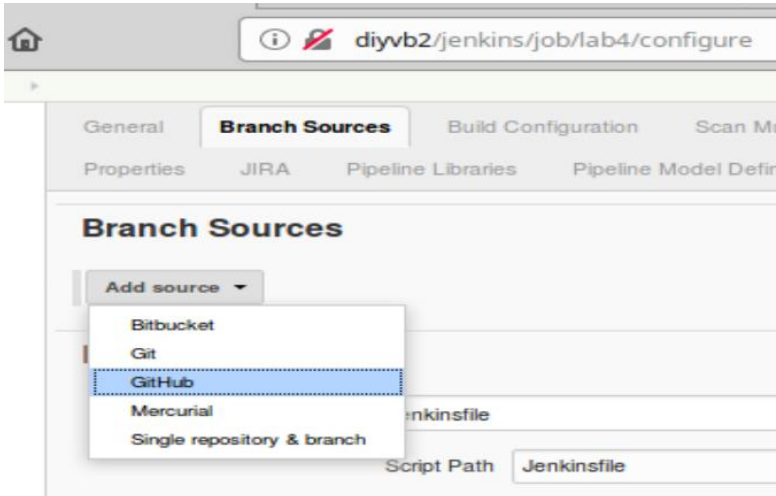
10. After selecting GitHub as the source, select your credentials from the drop-down list in the "Credentials" section.

    Next, in the "Owner" area, type in your GitHub user id.

    After this, the "Repository" section should be populated with a list of projects that are in your GitHub area. An example is shown below. Select the "filedump" project from the list.
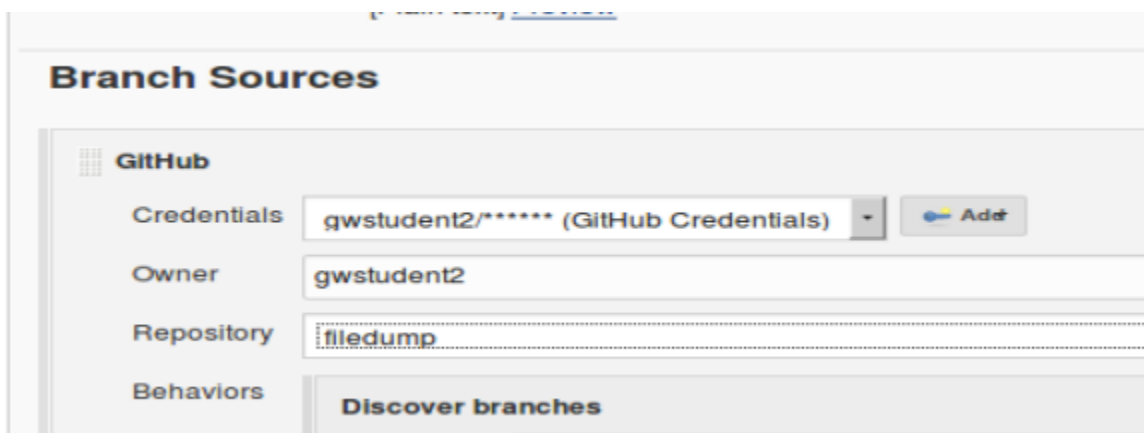
11. Scroll down to the "Scan Repository Triggers" section. Check the box that tells it to scan "Periodically if not otherwise run and set the interval to "1 minute".

© 2018 Brent Laster

Scan Repository Triggers

☑ Periodically if not otherwise run

Interval 1 minute

12. Save your changes. After saving, Jenkins will scan each branch of the repository checking for Jenkinsfiles. It will find one in the master branch and create a Jenkins job for it.

lab4

Very simple repo to use in Intro to CI course

Branches (1)		Pull Requests (0)				
S	**W**	**Name** ↓	**Last Success**	**Last Failure**	**Last Duration**	**Fav**
🔴	⛈	master	N/A	3 min 19 sec - #1	56 sec	🔁 ☆

Icon: S M L

Legend 🔊 RSS for all 🔊 RSS for failures 🔊 RSS for just latest builds

13. Now we'll clone the repository, make a simple change in another branch and create a push request for that. First, change to a terminal session, and clone down a copy of your project from GitHub.

    cd ~

    git clone https://github.com/<your github userid>/filedump

14. Let's add unit testing into our pipeline via a pull request. Change into the project directory and switch to the test branch. It already has most of the structure there to support unit testing.

    cd filedump

    git checkout test

15. It needs a Jenkinsfile to be able to build in Jenkins. Create one by simply copying the file "lab-4.txt" from your desktop into the directory where you cloned the project from GitHub.

    cp ~/Desktop/lab-4.txt   Jenkinsfile

16. Now, stage, commit, and push your changes into your GitHub space.

    git add .

    git commit -m "Add Jenkinsfile for testing"

    git push origin test:test

Lab 5: The Pull Request CI cycle

In this lab, you'll learn about creating Pull Requests and automatic verification tests for them.

© 2018 Brent Laster

1. Now switch back to your GitHub browser window for the forked filedump project. In the dropdown above the filelist for "Branch", select "test" to change to that branch.
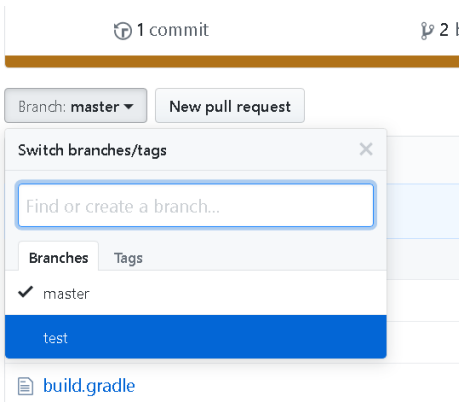
2. You'll now see several new buttons/links here to initiate a "Pull Request".   Click on the green one for "Compare and Pull Request".
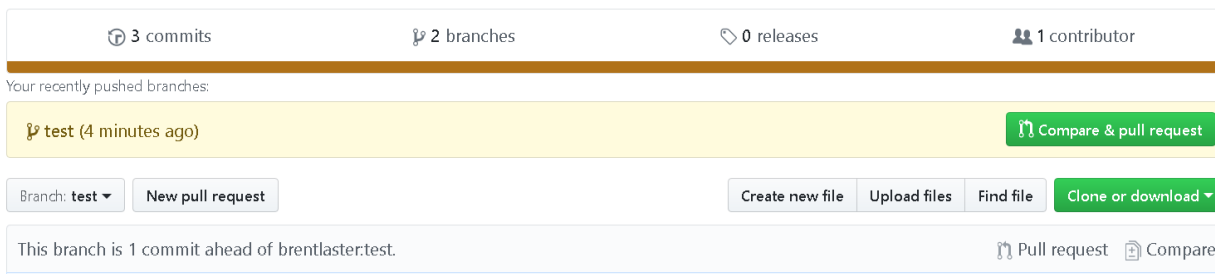
3. By default, this assumes that you want to do a Pull Request back to the project you forked from. In this case, we want to do a simple Pull Request between branches in the same project. To set this up, we will change the selected values in the "base" section on the left in the row of boxes. First go to the "base: test" box and change the branch "test" to "master".
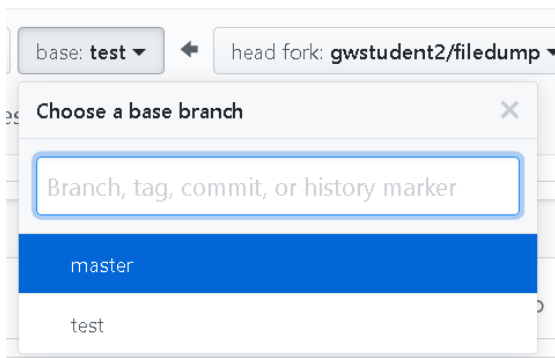
4. Now change the leftmost box for "base fork" to select your project (not brentlaster/filedump) .

© 2018 Brent Laster

Open a pull request

Create a new pull request by comparing changes across two b

base fork: **brentlaster/filedump** ▾   base: **test** ▾   ◄

Choose a Base Repository   ✕   atical

Filter repos

✓ brentlaster/filedump

gwstudent2/filedump

Leave a comment
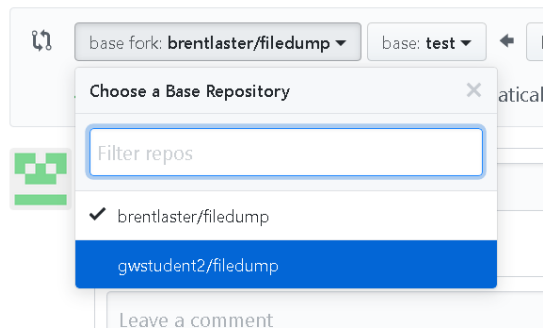
5. Afterwards, the section across the top should look like this:

Open a pull request

Create a new pull request by comparing changes across two branches. If you need to, you can also compare across forks.

base: **master** ▾   ◄   compare: **test** ▾   ✓ **Able to merge.** These branches can be automatically merged.
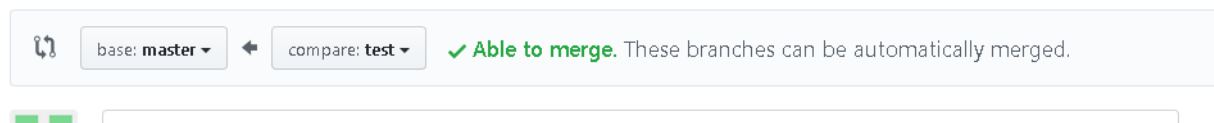
6. You can scroll down to see all the differences in files (similar to the Gerrit diffs).
7. Now enter a comment if you want for the Pull Request and then click on the "Create Pull Request" button.
8. After a moment, you'll see a section on the GitHub page where a check is running back on your Jenkins system to do a "test" build of this pull request.

Add more commits by pushing to the **test** branch on **gwstudent2/filedump**.

Projects

None yet

Some checks haven't completed yet   Hide all checks
1 pending and 1 successful checks

Milestone

No milesto

● 　 continuous-integration/jenkins/pr-merge *Pending — This commit is being built*   Details

✓ 　 continuous-integration/jenkins/branch — This commit looks good   Details

Notificatic

✓ **This branch has no conflicts with the base branch**
Merging can be performed automatically.

You're rece
you autho

Merge pull request   ▾   You can also open this in GitHub Desktop or view command line instructions.

2 participa

9. Switch back to the lab4 project in Jenkins and notice that now, instead of two branches, we have the master branch and the pull request.
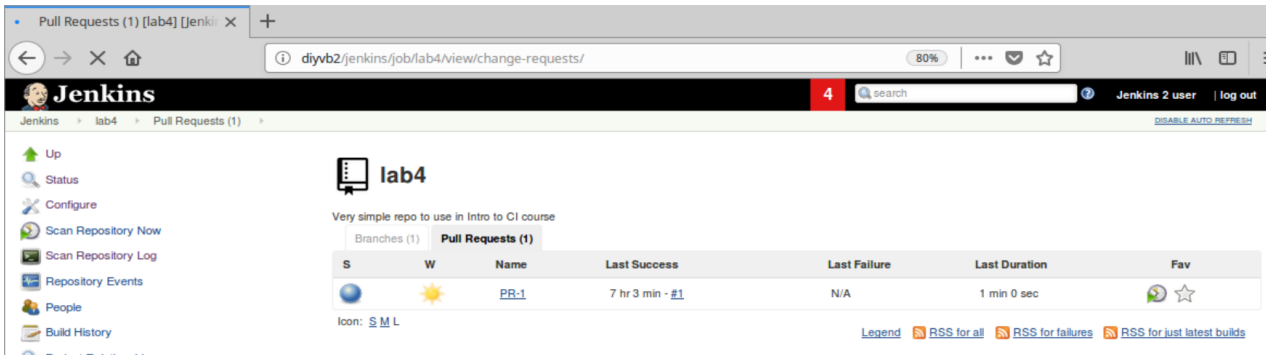
10. And, after a bit of time, if you look at the Console Output for that Pull Request, you'll see that it's completed and, at the end of the output is a notice that GitHub has been notified of the result.
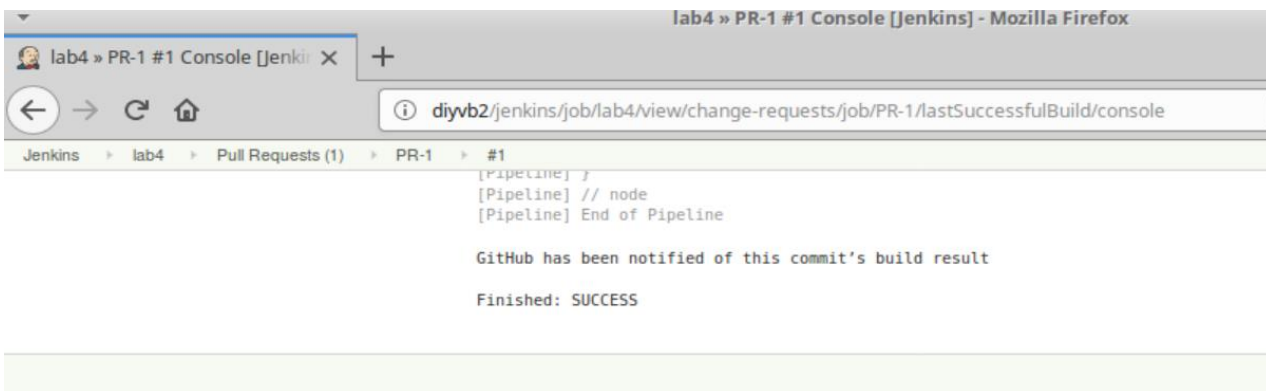
11. If you now look back on the Pull Request page on GitHub, you'll see that the checks have been successful. Now click on the green "Merge pull request" button to initiate the merge.
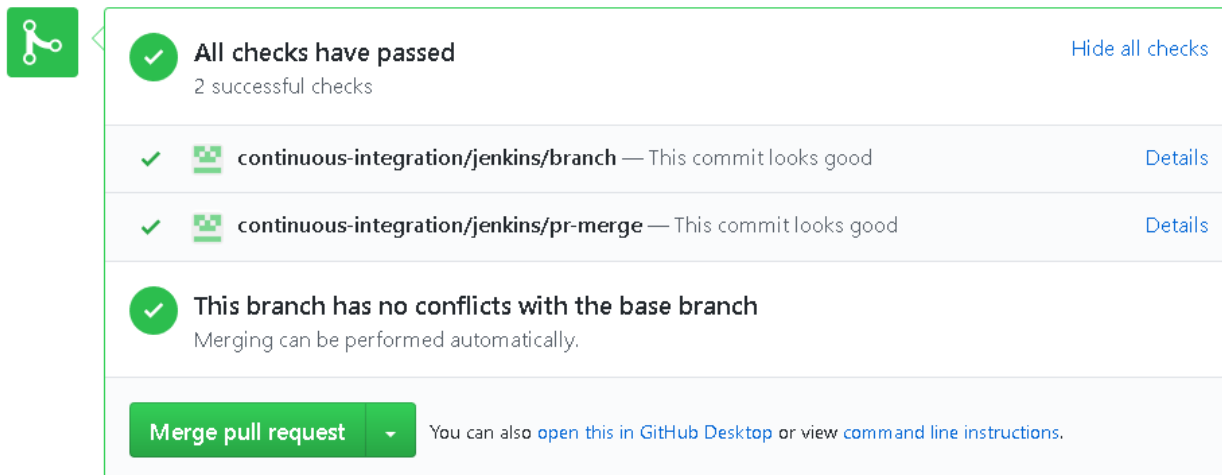
12. Click on the "Confirm merge" button to confirm we want to do this.

© 2018 Brent Laster

Add more commits by pushing to the **test** branch on **gwstudent2/filedump**.

Merge pull request #2 from gwstudent2/test

Test

Confirm merge   Cancel
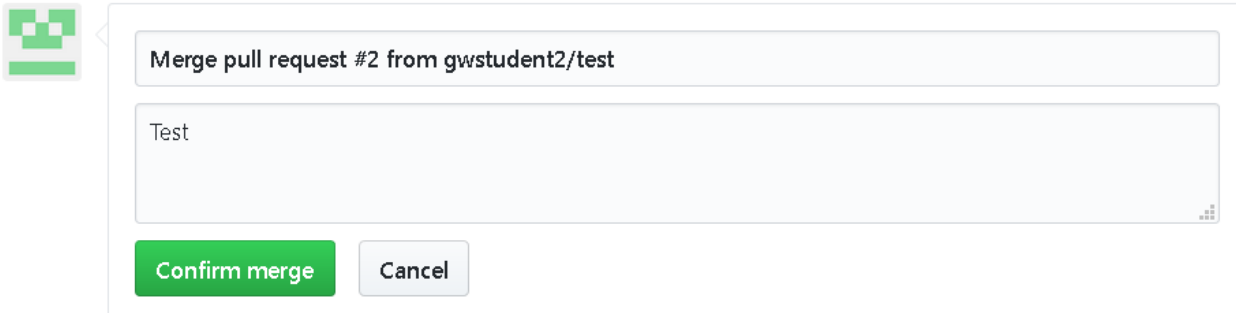
13. After a moment, you should see confirmation on the page that your Pull Request has been merged. At this point, you can choose to delete the branch or not.

gwstudent2 merged commit 234cccb into master 2 minutes ago   **View details**   **Revert**

2 checks passed

Pull request successfully merged and closed
You're all set—the test branch can be safely deleted.   **Delete branch**

Write   Preview   AA B i   " <> ⊘   ≔ ≣ ✓≣   @ 🔖 ↰
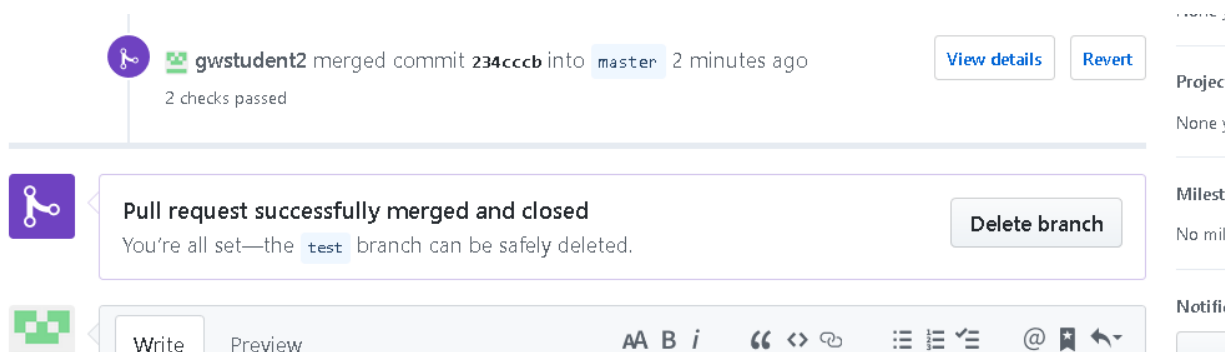
14. If you look back in Jenkins, you'll see we no longer show any Pull Requests and we are back to the original branch(es).

Very simple repo to use in Intro to CI course

Branches (2)   Pull Requests (0)

S	W	Name ↓	Last Success	Last Failure	Last Duration	Fav
	☀	master	11 hr - #1	N/A	1 min 10 sec	
	☀	test	N/A	N/A	N/A	

Icon: S M L

Legend   RSS for all   RSS for failures   RSS for just latest builds

Lab 6. Doing a Pull Request to another user

In this lab, you'll create a new branch, make a change and then see how it can be incorporated into another user's space on GitHub.

1. Go back to a terminal window and make sure you are in the filedump directory where you cloned your project.

2. Create a new branch from master – you can call it whatever you like. For our example, we'll call it userdata. (Make sure to include "master" on the end as the origin point.)

© 2018 Brent Laster

git checkout -b userdata master

3. You are now on the "userdata" branch. Change into the userdata directory and create a new file named <your github userid>.txt. You can put whatever contents you want in it.

cd userdata

gedit <your github userid>.txt

4. Stage, commit, and push your change to your GitHub repository.

git add .

git commit -m "add new user file"

git push origin userdata:userdata

5. Change back to your GitHub page in the browser. Click on the "Code" tab (above the project description). Change to the new userdata branch.

6. Click on the green "Compare & Pull Request" button. On the next screen it should show that you are initiating a pull request from <your github userid>/filedump, branch userdata -> brentlaster/filedump, branch master.
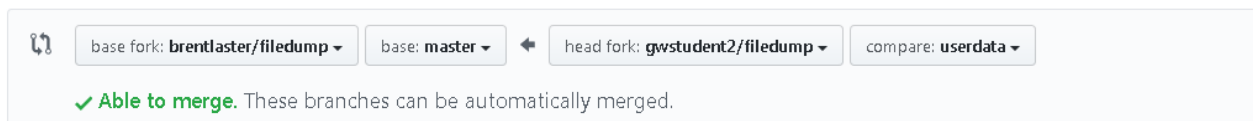
7. Enter a comment if you want and click on the green button to "Create Pull Request".

© 2018 Brent Laster

<> Code   🏷 Pull requests **0**   📋 Projects **0**   📖 Wiki   📊 Insights   ⚙ Settings

Open a pull request

Create a new pull request by comparing changes across two branches. If you need to, you can also compare across forks.

🔄   base: master ▾   ◆   compare: test ▾   ✔ **Able to merge**. These branches can be automatically merged.

Add in unit testing		Reviewers ⚙
		No reviews

Write   Preview   AA B *i*   66 <> 🔗   ☰ ☰ ☰   @ 🔖 ↩

Add Jenkinsfile

Attach files by dragging & dropping, selecting them, or pasting from the clipboard.

📖 Styling with Markdown is supported   **Create pull request**

Reviewers ⚙
No reviews

Assignees ⚙
No one—assign yourself

Labels ⚙
None yet

Projects ⚙
None yet

Milestone ⚙
No milestone

◈ **2 commits**   📄 **4 files changed**   💬 **0 commit comments**   👥 **1 contributor**

📑 Commits on Aug 17, 2018

◇   🖼 nfjsuser1   update for testing   002f01d

📑 Commits on Aug 18, 2018

◇   🖼 nfjsuser1   Add Jenkinsfile   0003394

📄 Showing **4 changed files** with 48 additions and 1 deletion.   Unified | Split

21 ■■■■□ Jenkinsfile			View 🖥 ⌄

```
      @@ -5,11 +5,30 @@ node ('worker_node1') {
5    5       try {
6    6         stage('Source') {
7    7           checkout scm
     8   +         stash name: 'test-sources', includes: 'build.gradle,src/test/'
8    9         }
9   10         stage('Build') {
10  11           // Run the gradle build
11  12           sh 'gradle clean build'
12   -         }
    13   +         }
```

8. Notice now that back on the original project's page your pull request shows up.

9. Now when signed in, the other user will see something like this:

And, clicking on the new pull request, the user can merge it if they choose.

new user file *#2*

🍴 **Open**   **gwstudent** wants to merge 1 commit into `brentlaster:master` from `gwstudent:userdata`

💬 Conversation **0**   ⊷ Commits **1**   📑 Checks **0**   📄 Files changed **1**

gwstudent commented 6 minutes ago   First-time contributor   +😀 •••

example pull request to another project

⊶ ⬛ new user file   575d676
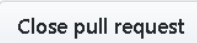
Add more commits by pushing to the **userdata** branch on **gwstudent/filedump**.

✅ **This branch has no conflicts with the base branch**
Merging can be performed automatically.

Merge pull request ⌄   You can also open this in GitHub Desktop or view command line instructions.

| Write | Preview |

AA **B** *i*   " <> 🔗   ☰ ☰ ✓   @ 🔖 ↩⌄

Leave a comment

Attach files by dragging & dropping, selecting them, or pasting from the clipboard.

Ⓜ Styling with Markdown is supported   **Close pull request**   **Comment**

© 2018 Brent Laster