Next-Level Git – Master your content

Revision 2.0 - 01/28/21

Brent Laster

Setup - Installing Git

If not already installed, install Git on your computer. Install packages for Windows and Mac are available via the internet.

Windows: (Bash shell that runs on Windows)

http://git-scm.com/download/win

Mac/OS X:

http://git-scm.com/download/mac

Linux:

http://git-scm.com/download/linux

After installing, confirm that git is installed by opening up the Git Bash shell or a terminal session and running:

git --version

You'll also need a free GitHub account from https://github.com.

Please ensure these prerequisites are done before the labs.

==========

END OF SETUP

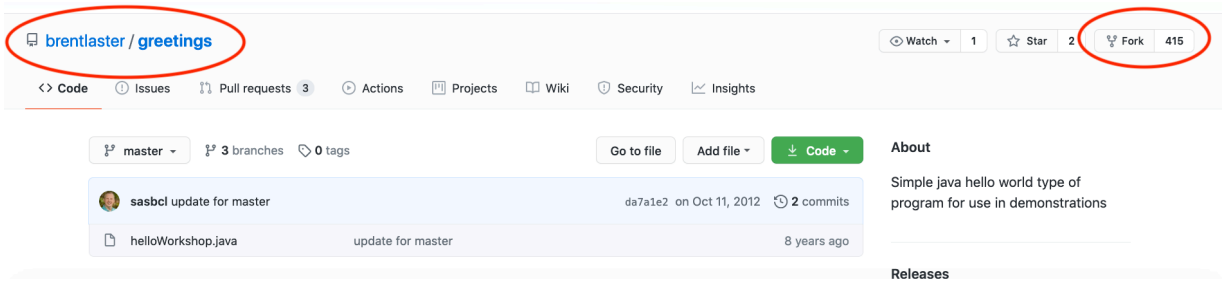
==========

Lab 1 - Using rerere

Purpose: In this first lab, we'll learn how to use the Git rerere functionality to teach Git how to resolve merges automatically.

1. Go to https://www.github.com
2. Log in to your own github account.
3. Browse to the greetings project at https://github.com/brentlaster/greetings
4. Click on the **Fork** button at the top and wait while the repository is forked to your userid.

   (Note: The **Fork** button is on the right in the same row as the brentlaster/greetings title.)



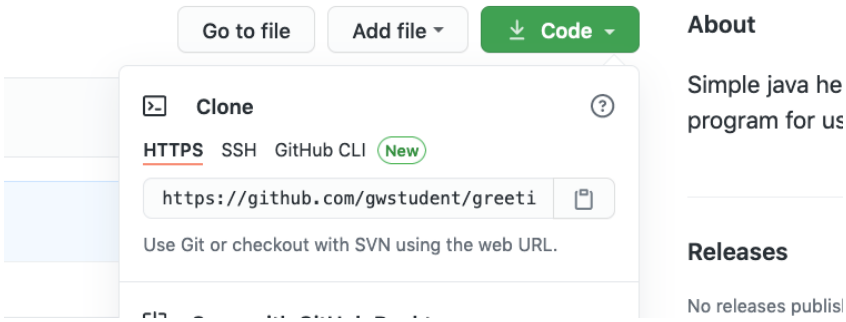
5. Open up a terminal/command line session on your local machine and start out in your home directory.

6. Configure your basic Git identifier information.

```
$ git config --global user.name "<first> <last>"
$ git config --global user.email <email address>
```

7. Clone your new repository down to your local system. You can just type the command below or copy the command from GitHub.

```
$ git clone https://github.com/<your github userid>/greetings.git
```



8. After the clone completes, change into the greetings directory.

```
$ cd greetings
```

Look around either locally or on github. Note that we have three branches, among them **master** and **topic1** - each with a different version on the tip of the branches. Each of these versions is also different from their common ancestor.

```
$ git branch -a
$ git log --graph --all
```

9. Now, enable the rerere functionality.

```
$ git config --global rerere.enabled 1
```

10. While the clone brought down all of the origin branches, we need to create a local branch to track **origin/topic1**. Execute the command below:

```
$ git branch topic1 --track origin/topic1
```

11. Now, let's merge **topic1** into **master**

```
$ git checkout master  (if not already on master)
$ git merge topic1
```

Note that there were merge conflicts, but you can also see the new message about "**Recorded preimage…**" Git is remembering the conflicts and resolution.

12. We can use options to git rerere to see what git rerere knows at this point.

```
$ git rerere status  (to see what files rerere is tracking)
$ git rerere diff  (to see resolution so far)
```

13. Edit the file and resolve the conflicts as you want. Then, stage and commit the changes.

```
<edit file and resolve conflicts in helloWorkshop.java>
$ git commit -am "<comment>"
```

Note the line in the output that mentions "**Recorded resolution…**" Git has now learned how to resolved this conflict in the future.

14. Since this was only a test merge, undo it now.

```
$ git reset --hard HEAD~1
```

15. Now, assume that it's later and we want to rebase **topic1** onto **master**. Issue the following commands.

```
$ git checkout topic1
$ git rebase master
```

16. Note that even though there were conflicts, git states that it "**Resolved… using previous resolution**". (You'll have to scan through the other output to find this.) This is because we taught it how earlier.

17. Since the conflict is already resolved, stage the changes and let the rebase continue.

```
$ git add  .
$ git rebase --continue
```

==========

END OF LAB

==========

Lab 2 - Working with Filter-branch

Purpose: In this lab, we'll see how to work with the filter branch functionality in Git

1. For this lab, we'll need a more substantial project to work with. We'll use a demo project with several parts that I use in other classes. In your home directory (or a directory that is not a clone of a git project), clone down the roarv2 project from http://github.com/brentlaster/roarv2.

```
$ git clone https://github.com/brentlaster/roarv2
```

2. We are interested in splitting the web subdirectory tree out into its own repository. First, note what the structure looks like under web, as well as the log, for a reference point.

```
$ cd roarv2
$ ls -la web
$ git log --oneline
$ git log --oneline web
```

3. Now we can run the command to split the web piece out into its own repository. (Note the syntax at the end. It's the word "web" followed by a space, then a double hyphen, then another space, then "—all".)

```
$ git filter-branch -f --subdirectory-filter web -- --all
```

4. After the preceding step runs, you should see messages about "refs/heads/<branch name> being rewritten". This is the indication that the process worked. Do an ls and look at the history to verify that your repository now shows only the web pieces.

```
$ ls -la
$ git log --oneline
```

5. Now, let's restore the original larger repository back for the other parts of this lab. To do this, we'll find the original SHA1 value for HEAD from before we did the operation

```
$ cat .git/refs/original/refs/heads/master
```

We can also use the reflog to see the commit before the change.

```
$ git reflog
```

6. Now, take the first 7 characters of the value you find in the first step below and plug them in to the step below.

```
$ git reset --hard <first 7 characters from SHA1 output of step above>
```

And now look at the working directory and notice that everything is as it was before.

```
$ ls
```

7. Let's look at another application of filter-branch - removing files from the history. In this case, we'll use filter-branch to remove the file build.gradle from the master branch just as a prevalent example file. You should be on the master branch. First, let's see everywhere this file has been involved in a change. Run the command below to do this: (Note that there are spaces on each side of the last --).

```
$ git log --oneline --name-only -- build.gradle
```

8. Now, let's run the filter-branch command to remove the build.gradle file from all of the commits in this branch. (Note that these are single quote characters around the git rm command, not backticks.)

```
$ git filter-branch -f --index-filter 'git rm --cached --ignore-unmatch build.gradle'
```

9. Take a look now and notice that the file is no longer in the commits for this branch. (You will see no listing here.)

```
$ git log --oneline --name-only -- build.gradle
```

10. Let's do one more use case for the filter-branch command. We'll use the env-filter to change the email address for a few of the commits in the master branch. To do this, as part of the command, set the desired email address to your email address, and then we'll export it for the filtering. Here's the command to run:

```
$ git filter-branch -f --env-filter 'GIT_AUTHOR_EMAIL=<your email address>;
export GIT_AUTHOR_EMAIL'
```

12. After this change, you can run a simple formatted log command to see the updates:

```
$ git log --format="%h %ae"
```

==========

END OF LAB

==========

Lab 3: Working with Interactive Rebase

Purpose: In this lab, we'll see how to use interactive rebase to squash multiple commits into one.

1. Pick one of your directories that already has a git project in it. (You can use the roarv2 directory from the last lab for simplicity if you want.) In that working directory, make three new changes and commit each one separately. (Once you type the first one, you can just the recall function (up arrow) to bring up the line again and edit the last number if you want.)

```
$ echo data >> lab.txt; git add lab.txt; git commit -m "Update to lab.txt, version 1"

$ echo data >> lab.txt; git add lab.txt; git commit -m "Update to lab.txt, version 2"

$ echo data >> lab.txt; git add lab.txt; git commit -m "Update to lab.txt, version 3"
```

2. Do a git log to see the 3 separate commits showing up in the history.

```
$ git log  (or git log --oneline)
```

3. After doing the 3 commits, we'll initiate an interactive rebase for the last 3 commits. As stated, we specify the starting point as the one before the first one we want to change.

```
$ git rebase -i HEAD~3
```

4. Now, in the editor window that comes up with the list of commits (titled "git-rebase"), modify the action on the left to be "squash" for the bottom two entries. The format of your file should look similar to below (note that this is demonstrating the contents of the upper part of the file NOT commands you type in and execute):

```
pick  <sha 1>  "Update to lab txt, version 1"
squash <sha 1> "Update to lab.txt, version 2"
squash <sha 1> "Update to lab txt, version 3"
```

5. Save your file and exit the editor. Git will now start the interactive rebase running. You can watch it go through the steps. **After it first runs, it will stop and bring up the editor again.** This is so you can enter what you want the commit message to be for the squashed commit (since you won't have the 3 separate ones anymore). You can just put whatever you want in this field (or leave it as-is). Save any changes and exit the editor. The rebase should then continue and complete.

6. Finally, do another git log and notice that there is now a single (squashed) commit where there were previously 3.

```
$ git log (or git log --oneline).
```

==========

END OF LAB

==========

Lab 4: Working with Cherry-pick

Purpose: In this lab, we'll see how to work the cherry-pick command, merge conflicts with it and an alternative merge strategy.with two kinds of custom filters to modify file contents automatically on checkout and commit.

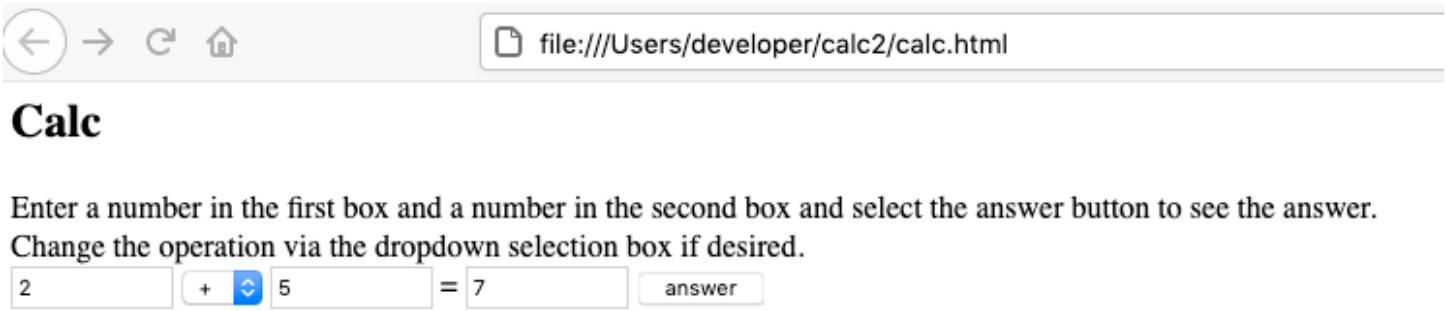
1. I have a simple little calculator program with several different branches on it to use for this example. Clone a copy down to your home directory or a directory that is not already a working directory for a repository.

```
$ git clone https://github.com/brentlaster/calc2
```

2. Change into the cloned area and take a look at the branching relationship we have there with the UI branch, the features branch, and the docs branch.

```
$ cd calc2
$ git log --graph --all --oneline
```

3. If you want, you can open/start the calc.html file in your browser to see it in practice.



4. Now we want to merge changes from the feature and ui branches into a new branch. Create a local cpick branch to work in and also a local UI branch from the remote one.

```
$ git branch features origin/features
$ git branch ui origin/ui
$ git checkout -b cpick
```

5. Let's see what's in the features branch that's NOT in the master branch - specific commits to the features banch.

```
$ git log features ^master --oneline
```

6. For simplicity, let's cherry-pick the commit that added the max function. Find the SHA1 abbreviation for that commit from the log listing above – write it down (i.e. the 7 characters in front of the "add max function" comment).

7. Do a quick log of your current branch.

```
$ git log cpick --oneline
```

8. Issue the command to cherry-pick that commit's SHA1 (the one from step 4) onto our current branch.

```
$ git cherry-pick <sha1 of "add max function" commit>
```

9. Assuming there were no errors, do another quick log of your current branch.

```
$ git log cpick --oneline
```

10. Now, find the SHA1 of the commit on the ui branch with the comment "update title and button text". Make a note of it.

```
$ git log ui --oneline
```

11. Attempt to cherry-pick that SHA1 into your current branch.

```
$ git cherry-pick <sha1 of ui commit>
```

12. Did you encounter errors about "could not apply"? Which file is in conflict? Use the git status command to find out.

    $ git status
    (Hint: Remember what "both modified" means?)

    Notice the part about "You are currently cherry-picking…"

13. What are the conflicts?

```
$ git diff
```

14. We are in the cherry-picking state till we complete or abort the operation. In this case, we just really want the changes from the ui branch, so we'll force it to merge those in instead of reconciling it ourselves. First, cancel this cherry-pick.

```
$ git cherry-pick --abort
```
    Notice your branch prompt now.

15. Now, since git is defaulting to the recursive merge strategy here (you can read more about merge strategies in git merge --help), we just need to supply the merge strategy option to take the changes from the source branch.

That option is called “theirs” - meaning coming from the other branch. If we wanted to take the one from our branch instead, we would pass “ours”. Execute the following command:

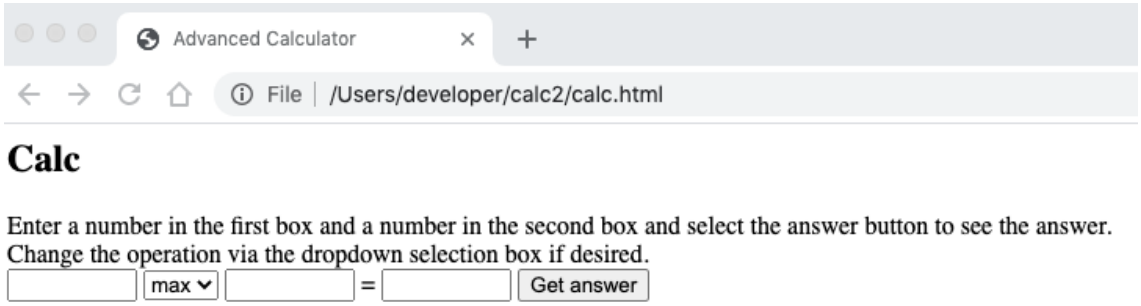
```
$ git cherry-pick –Xtheirs <sha1>
```

(Note: The sha1 above is the one from the UI branch for “update title and button text.”)

10. Startup calc.html and verify that it looks as expected.

start or open calc.html in browser

Note that the button text should now read “Get answer” and the title in the title bar should be “Advanced Calculator”. Also the “max” function should be available in the drop down list of operations.



==========

END OF LAB

==========

OPTIONAL/BONUS Lab 5: Using a Git Attributes File to Create a Custom Filter

Purpose: In this lab, we’ll see how to work with two kinds of custom filters to modify file contents automatically on checkout and commit.

Setup:

Suppose that you have a couple of HTML files that you use as a header and footer across multiple divisions in your company. They contain a placeholder in the form of the text string %div to indicate where the proper division name should be inserted.

You want to use the smudge and clean filters to automatically replace the placeholder with your division name (ABC) when you check the file out of Git, and set it back to the generic version if you make any other changes and commit them.

1. Pick one of your directories that already has a git project in it. (You can use the roarv2 directory from the recent labs for simplicity if you want.) Create two sample files to work with. Commands to create them are below (you may use an editor if you prefer).

```
$ echo "<H1>Running tests for division:%div</H1>" > div_test_header.html

$ echo "<H1>Division:%div testing summary</H1>" > div_test_footer.html
```

2. Go ahead and stage and commit these new files into your local Git repository.

```
$ git add div*.html
$ git commit -m "adding div html files"
```

3. Now, we'll create a custom filter named "insertDivisionABC" that will have the associated "clean" and "smudge" actions. This is done by using git config to define this in our Git configuration. Execute the following two config commands:

```
$ git config filter.insertDivisionABC.smudge "sed 's/%div/ABC/'"

$ git config filter.insertDivisionABC.clean "sed 's/ision:ABC/ision:%div/'"
```

4. To see how this is setup in the config file, take a look at your .git/config file and find the section for the insertDivisionABC filter:

```
$ cat .git/config
```

Look for the section starting with "[filter "insertDivisionABC"]

5. Now, we just need to create a Git attributes file with a line that tells Git to run your filter for files matching the desired pattern. We will create a very simple one-line file for this.

```
$ echo "div*.html   filter=insertDivisionABC"  >  .gitattributes
```

6. Now, we'll remove our local copies of the files and check them out again so they will get the filter applied.

```
$ rm div*.html
$ git checkout div*.html
```

7. Take a look at the contents of the two files after the checkout has applied the smudge filter.

```
$ cat div*.html
```

Notice that we have the division name (ABC) inserted into the files now.

8. Edit the html files and make a simple modification. For example, you might change “testing summary” to “full testing summary” and/or “tests for division” to “all tests for division”. Save the changes.

9. Now do a git diff on the files. Notice that with the clean filter in place, the diff takes into account the filter and just shows you the differences without the substitution being applied.

```
$ git diff
```

10. Add the files back into Git and run a status command to see the state.

```
$ git add div*.html
$ git status
```

11. Run a git diff command and note that it shows no differences. Then cat the local versions of the files. Notice that they still show the substitution from the smudge.

```
$ git diff
$ cat div*.html
```

12. We know that the clean filter should have removed the substitution when the files were staged. To verify that, we’ll use a special form of the show command to see the versions of the files in the staging area.

```
$ git  show  :0:div_test_header.html
$ git  show  :0:div_test_footer.html
```

Notice that the change to insert the division has been “cleaned” per the clean filter.