Git in 4 Weeks - Part 3 Labs

Revision 1.6 - 06/10/24

Brent Laster for Tech Skill Transformations LLC

Note #1: If you are using an older version of Git, the default branch may be "master" instead of "main". If this is the case, you can substitute "master" where the labs specify "main". Or you can change the default branch to main by using the command below when you are on the master branch.

```
$ git branch -M main
```

Lab 7 - Using the Git bisect command

Purpose: In this lab, we'll learn how to quickly identify the commit that introduced a problem using Git bisect.

Note: If you are on windows, to run the "sum.sh" script you will need to do that through the Git Bash Shell. If you do not have the bash shell, you can just "type sum.sh" at each step and look at the code to determine if it would work or not.

1. Go back to your home directory (or choose a directory that does not already have a git repo in it) on your machine and clone the bisect project from GitHub.

```
$ git clone https://github.com/skillrepos/bisect
```

2. Change directory into the new bisect directory and run the sum.sh program to see what it does and see if you get a correct answer.

```
$ cd bisect
$ ./sum.sh
```

3. We need to figure out where the problem was introduced. Start the git bisect operation.

```
$ git bisect start
```

4. Git tells you that it is waiting for the good and bad commits (to set the bounds of the range of commits it should check). Tell Git that the current version is incorrect.

```
$ git bisect bad
```

5. We need to supply a bound for a good one. Get a list of the SHA1 commits in the branch.

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

6. Tag the original commit to make it easier to work with. (Note version 1 not version 12.)

```
$ git tag first <SHA1 of version 1 commit>
(example: git tag first d9d9585)
```

7. Get a checkout of the first version. (You can ignore any messages about "switch" etc.)

```
$ git checkout first
```

(Why didn't we have to specify a file here?)

8. Run the first version of sum.sh (1.01) and verify that it works as expected.

```
$ ./sum.sh
```

9. Mark that revision as good to set the other end point for the range to work with.

```
$ git bisect good
```

10. Git has now checked out a version of the file from the "middle" of the set of revisions in the branch. Which one did it checkout and is that version good? (Hint: look at the commit message that is shown.)

```
$ ./sum.sh
```
(Note the version number that is output.)

(Note that if you wanted to get a clearer idea of where this one was in the chain, you could always do "git log --oneline" and find the sha1 or HEAD in that list.)

11. Mark this version appropriately - depending on whether it works or not.

    $ git bisect good (if it returns the right answer)

    OR

    $ git bisect bad (if it doesn't return the right answer)

12. Repeat steps 10 and 11 (running the shell script and doing git bisect bad or git bisect good) for each commit Git provides until you see a message that says:

    "<SHA1> is the first bad commit" (Message will be displayed immediately under bisect command.)

13. Once you see the message about the "…first bad commit", reset to the commit that is prior to the bad one. The reason is so we can isolate the set of good commits in another branch. To do this, use the following command:

    $ git bisect reset refs/bisect/bad^

    (Here, the refs/bisect/bad is the first bad revision and the "^" on the end means "the one before".)

14. Create a new branch from this version and switch to it. This will just keep the good commits in it since that is the point where we are branching. Examine the history to make sure it looks correct.

    $ git checkout -b <branch-name>

    $ git log --oneline

==========

END OF LAB

==========

Lab 8 - 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 part of the log that pertains to the "web" path, 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".) This will bring up a warning first that you can ignore and then will still take several seconds to process and return output.

```
$ 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. Note the first 7 characters of the returned SHA1 – you'll need them for step 6.

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

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 found in the first part of step 5 and plug that into the step below.

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

And now look at the working directory. Notice that everything is as it was before you did the filter-branch (web is a separate subdirectory in the larger set).

```
$ 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 main branch just as a prevalent example file. You should be on the main 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 main 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'
```

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

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

==========

END OF LAB

==========

Lab 9: 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. (If you end up with output that stops and you can't quit it, hit the "Q" key.)

```
$ 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 10: 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.

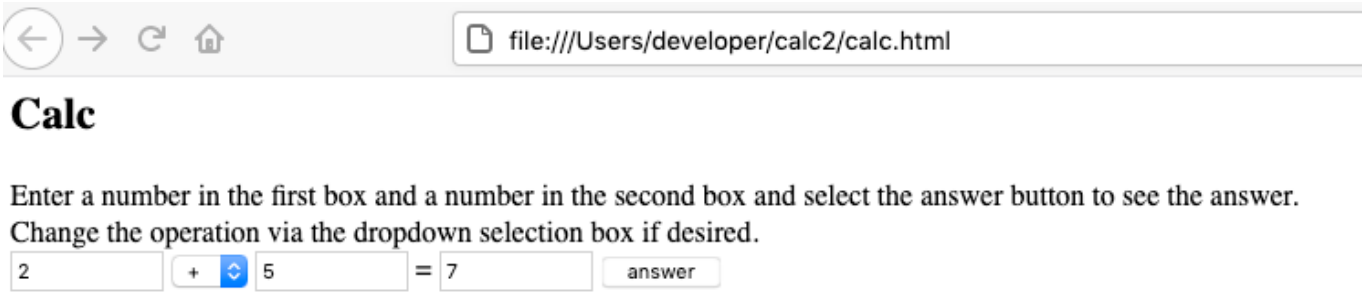
1. I have a simple little calculator program with several different branches on it to use for this example. Clone a copy down to your home directory or a directory that is not already a working directory for a repository.

```
$ 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 main branch - specific commits to the features branch.

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

```
* b372aa6 (origin/features) add avg function
* 3753e5a add min function
* 482365d add exp function
* d003b91 add max function
```

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. You should see the commit with the max function.

```
$ 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 that one.

```
$ 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…" near the top of the output.

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
```

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:

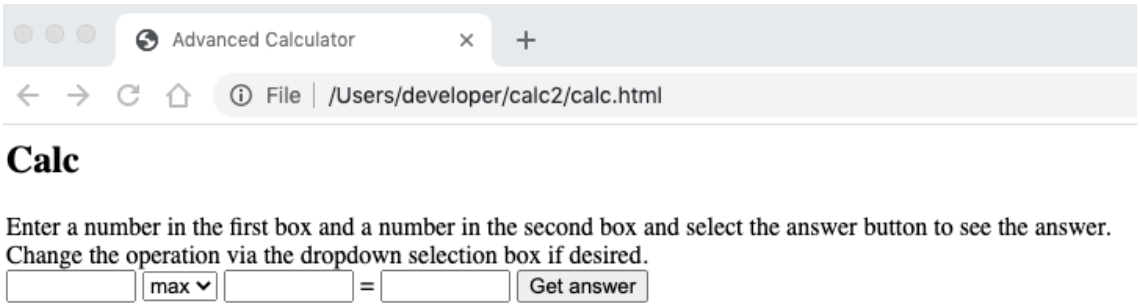
```
$ git cherry-pick –Xtheirs <sha1>
```

    (Note: The sha1 above is the same 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

==========