Git Fundamentals

Revision 1.6

9/3/20

Lab 1 - Creating and Exploring a Git Repository and Managing Content

In this lab, we'll create an empty Git repository on your local disk and stage and commit content into it.

Prerequisites

To complete this and all future labs in the course, you must have a working version of Git installed. We assume 2.0 or higher for the version. (To see which version you have, you can run **git --version**) If you don't have a working version of Git installed, then you should install it now.

Steps

1.) On your local disk, **create a new directory** and change (**cd**) into it. (This will be the directory we work in unless otherwise specified).

2.) In the new directory, initialize a new repo by running the following command:

```
git init
```

This command created a new git repository skeleton in a subdirectory named "**.git**" under the current directory - as indicated by the output message from the command. This means that you're now able to start using other Git commands in the current directory.

3.) Tell Git who you are by setting your basic identification configuration settings with the following commands (Note the double dashes before "global" since we are spelling out the option. Also, values only require quotes if there is a space in the value.)

```
git config --global user.name "First-name Last-name"

git config --global user.email emailAddress@provider
```

4.) Now let's create some content to put through the Git workflow. Note that for purposes of these initial labs, we just need files to work with - we don't really care what's in them. We can "cheat" and just echo something into a file via the ">" operator. In fact, the output of any command could be used to put content into a file via the ">" operator. Of course, if you prefer, you can certainly create files via your favorite editor instead.

Create two files - contents and names don't matter.

```
echo content > file1.c

echo content > file2.c
```

5.) Stage the files with the add command. (If you prefer you can add each separate file explicitly rather than use the ".")

```
git add .
```

Note: If you see any messages about end of line conversions, you can ignore them.

6.) Now commit the files. You can use whatever commit message (comment) you want. Note the single hyphen/dash before the short form of the option.

```
git commit -m  "<commit message>"
```

7.) Notice the output you get. There is the branch name - the default branch - **master**, followed by an indicator that this was the first (root) commit and then the first few characters of the SHA1 for the commit.

8.) Edit one of the files. (We can just use the ">>" to append something to the file's content.)

```
echo more >> file1.c
```

9.) Stage and commit the file with the shortcut. Note the combined short options "-am" for "-a" + "-m".

```
git commit -am  "<commit message>"
```

Lab 2 - Tracking Content through the File Status Lifecycle

In this lab, we'll work through some simple examples of updating files in a Local Environment and viewing the status and differences between the various levels along the way.

Prerequisites

This lab assumes that you have done Lab 1: Creating and Exploring a Git Repository and Managing Content. You should start out in the same directory as that lab.

Steps

1.) Starting in the same directory as you used for Lab 1, run the status command or the short form to see how it looks when you have no changes to be staged or committed.

`git status` (or git status –s)

2.) Create a new file and view the status.

`echo content > file3.c`

`git status` (or git status -s)

Is the file tracked or untracked?

Answer: It's untracked - we haven't added the initial version to Git yet.

3.) Stage the file and check status

`git add .` (or git add file3.c)

`git status` (git status –s if you want)

Is the file tracked or untracked? What does "Changes to be committed" mean?

(Answers: The file is now tracked - we've added the initial version to Git. "Changes to be committed" implies files exist in the Staging Area and the next step for them is to be committed into the Local Repository.)

4.) Edit the same file again in your Working Directory and check the status.

```
echo change > file3.c

git status
```

Why do we see two?

Where is the version that's listed as "Changes to be committed"? (Working Directory, Staging Area, or Local Repository)

Where is the version that's listed as "Changes not staged for commit"? (Working Directory, Staging Area, or Local Repository)

(Answers: We see two because there is one version of the same file in the Working Directory and another version in the Staging Area.

The version that's listed as "Changes to be committed" is in the Staging Area. The phrase implies that this version's "next step" or "next level for promotion" is to the Local Repository via a commit.

The version that's listed as "Changes not staged for commit" is in the Working Directory. The phrase implies that this version's "next step" or "next level for promotion" is to the Staging Area, since it's currently "not staged".)

5.) Do a diff between the version in the Working Directory and the version in the Staging Area.

```
git diff
```

6.) Go ahead and commit and do another status check.

```
git commit -m "<commit message>"

git status
```

Which version did we commit – the one in the Staging Area or the one in the Working Directory? (Hint: Which one is left – shows up in the status? Note the "Changes not staged for commit" part of the status message.)

(Answer: The version in the Staging Area was the one committed. The content goes through the Staging Area and then into the Local Repository.)

7.) Stage the modified file you have in your Working Directory and do a status check.

```
git add .

git status
```

8.) Edit the file in the Working Directory one more time and do a status check.

```
echo "change 2" > file3.c

git status
```

At this point, we have a version of the same file in the Local Repository (the one we committed in step 6), a version in the Staging Area (the one we staged in step 7), and a version in the Working Directory (step 8).

9.) Diff the version in the working area against the version in the Staging Area.

```
git diff
```

10.) Diff the version in the Staging Area against the version in the Local Repository.

`git diff --staged` (or git diff --cached) (note the -- is a double -)

11.) Diff the version in the working area against the version in the Local Repository (the one we committed earlier).

```
git diff HEAD
```

12.) Commit using the shortcut.

```
git commit -am "committing another change"
```

Which version got committed – the one in the Working Directory or the one in the Staging Area?

(Answer: Since we used the -am shortcut, the version from the Working Directory was staged (over the previous version in the Staging Area) and then that version was committed into the Local Repository.)

13.) Check the status one more time.

```
git status
```

Notice the output - we're back to a clean Working Directory - Git has the latest versions of everything we've updated.

Lab 3 - Working with Changes Over Time and Using Tags

In this lab, we'll work through some simple examples of using the Git log commands to see the flexibility it offers as well as creating an alias to help simplify using it. We'll also look at how to tag commits to have another way to reference them.

Prerequisites

This lab assumes that you have done Lab 2: Tracking Content through the File Status Lifecycle. You should start out in the same directory as that lab.

Steps

1.) Starting in the same directory as you used for Lab 3, let's first make another change to the repository to make the history more interesting. Add a line to the first file you committed into the repository and then stage and commit. Note that you can use the shortcut here.

```
echo new >> file1.c
git commit -am "add a line"
```

2.) Now, take a look at the history we have so far in our small repository. To do this we just run the log command. (In some terminals your history may be longer than the screen and need to hit a key to continue. If you run are paging through log output and want to end the listing, hit the "q" key.)

```
git log
```

3.) Often when looking at Git history information, users will only want to see the first line of each entry - the "subject line". This is why it is important to make that first line meaningful in a real-life use of Git.

To see only the first line of each log message, you can use the --oneline option. Try it now.

```
git log --oneline
```

4.) Let's try a more complex version of the log command that includes selected pieces of history information formatted in a specific way. Be careful of your typing - note the colon after "format", the double hyphens, and the double quotes.

```
git log --pretty=format:"%h %ad|%s %d[%an]" --date=short
```

Press **Enter** to see this execute.

5.) Since this is a bit much to type, let's create an alias to simplify running this command. We do this by configuring the alias name to stand for the command and its options. Enter the following, paying attention to the punctuation (double hypens, colon, vertical bars, single and double quotes, etc.)

```
git config --global alias.hist "log --pretty=format:'%h %ad | %s%d [%an] ' --date=short"
```

6.) Now run your new hist alias. You should see the same output as the original log command from step 3. If you encounter any problems, go back and double-check what you typed in step 4.

```
git hist
```

7.) We can also use the log command (and our hist alias) on individual files. Pick one of your files and run the hist alias against it.

```
git hist <filename>
```

8.) We're interested in seeing the differences between a couple of the revisions. But there are no version numbers. How do we pick revisions?

(Answer: We pick revisions via the SHA1 (hash) values (first 7 bytes are enough). It's the first column in the hist output.)

9.) Run the git hist alias again and find the SHA1 values of the earliest and latest lines in the history. (Yours will of course be different from mine in the example below.)

```
git hist
```

```
  latest -> 1db49cf 2016-08-20 | add a line (HEAD -> master) [Brent Laster]
            ece66a5 2016-08-20 | committing another change [Brent Laster]
            8103190 2016-08-20 | update [Brent Laster]
            581c751 2016-08-20 | another update [Brent Laster]
earliest  -> c6a82d2 2016-08-20 | first commit [Brent Laster]
```

10.) We can use these SHA1 values similarly to how we might use version numbers in other systems. Let's see the history between our earliest and latest commits. To do this, we'll run the hist alias and specify the range of values using the SHA1 values. Execute the command below, substituting the appropriate SHA1 values from the history in your repository.

Format: git hist <earliest SHA1>..<latest SHA1>

```
git hist c6a82d2..1db49cf
```

11.) You should see a similar history as you saw previously. One thing to note here is you don't see the original (first) commit. This is because when specifying ranges via the ".." syntax, Git defines that as essentially everything after the first revision. Note that you can also run this against an individual file. Try the command below with your SHA1 values and the first file you added in the repository.

Format: git diff <earliest SHA1>..<latest SHA1> <first file>

```
git diff c6a82d2..1db49cf file1.c
```

12.) This is useful, but finding and typing SHA1 values each time for operations like this can be cumbersome. To simplify this, we can use tags to point to commits, and then use those tag names instead of the SHA1 values in commands. Let's create tags for the earliest and latest commits in our repository. We'll use the tags "first" and "last" respectively. The commands are below.

Format: git tag <tagname> <hash>

```
git tag first <your earliest SHA1 value>
git tag last  <your latest SHA1 value>
```

13.) Now that we have the tags, we can use them anyplace we used the SHA1 values before. Try out the hist alias with the tags.

```
git hist first..last
```

14.) You may not have thought about it, but this is giving us the history for all of the files in the repository. This is because a tag applies to an entire commit - not a specific file in the commit. To see this more clearly, add the --name-only option to the command and run it again.

```
git hist first..last --name-only
```

15.) What do we do if we only want to do an operation using a tag for one file? The answer is that simply add filename onto the command. Try out the example below.

```
git hist first..last --name-only file1.c
```

Lab 4 - Working with Branches

In this lab, we'll start working with branches by creating a new branch and making changes on it.

Prerequisites

This lab assumes that you have done Connected Lab 3: Working with Changes Over Time and Using Tags. You should start out in the same directory as that lab.

Steps

1.) Starting in the same directory as you used for Lab 3, take a look at what branches you have currently with the git branch command.

```
git branch
```

2.) You'll see a line that says "* master". This indicates that there is only one branch currently in your repository - master. The "*" next to it indicates that it is the current branch (the one you've switched to and are currently working in). If your terminal prompt is configured to show the current branch, it would also say "master".

3.) Now, before we work with a new branch, let's update the files in the master branch to indicate that these are the versions on master so it will be easier to see which version we have later. To do this, we can just use a short version of the same way we have been creating and updating other files.

On some non-Windows systems, run this command.

```
echo "master version" >> *
```

On Windows and some other systems, it will be necessary to issue the command for each file, such as:

```
echo "master version" >> file1
echo "master version" >> file2
…
```

4.) Stage and commit the updated files. Because these are files that Git already knows about, we can use the shortcut command here.

```
git commit -am "master version"
```

5.) When we work with branches, it can be helpful to see a visual representation of what's in the repository. To do this, we'll use the Gitk tool that comes with Git. Start up gitk in this directory and have it run in the background.

```
gitk &
```

6.) In gitk, create a new view. Follow the instructions below.

1. On the menu, select View, New View. Give it a name.
2. Check "Remember this view"
3. Check the four checkboxes under the "Branches & tags:" field. Click OK.
4. Switch to the new view under the View menu.

7.) We have a new feature to work on, create a feature branch with name "feature". Switch back to your terminal, and in the directory, run the command below.

```
git branch feature
```

8.) Notice that this command created the branch, but did not switch to. Let's check what branches we have and which is our current one.

```
git branch
```

9.) We can now see our new branch listed. Let's change into to the feature branch to do some work.

```
git checkout feature
```

10.) Verify that we're on the feature branch. Run the command below and observe that the "*" is next to that branch.

```
git branch
```

11.) Switch back to gitk, and refresh the screen to see what things look like visually now.

12.) Back in the terminal session, create a new file and then update the files in the feature branch to indicate that they are the "feature branch version".

```
echo "new file" > file4.c
```

On some non-Windows systems, you can use:

```
echo "feature version" >> *
```

On other systems, it will be necessary to issue the command for each file, such as:

```
   echo "feature version" >> file1
   echo "feature version" >> file2
…
```

13.) When you're done, stage and commit your changes.

```
git add .
git commit -m "feature version"
```

(Note: If you just used git commit -am, it wouldn't pick up your new file.)

14.) Refresh your view in gitk and take one more look around your feature branch.

15.) Switch back to the master branch.

```
git checkout master
```

16.) Verify you're on the right branch.

```
git branch
```

(Note: Should have a * by master.)

17.) Take a look at the contents of the files and verify that they're the original ones from master.

```
cat <filenames> (or type <filenames> on Windows)
```

Look for "master version" in the text.

18.) Refresh gitk and take a look around in it.

Lab 5 - Practice with Merging

In this lab, we'll work through some simple branch merging.

Prerequisites

This lab assumes that you have done Lab 4: Working with branches. You should start out in the same directory as that lab.

Steps

1.) Starting in the same directory as you used for Lab 4, make sure you don't have any outstanding or modified files (nothing to commit). You can do this by running the status command and verifying that it reports "working directory clean".

```
git status
```

2.) Now, create a new one-line file with a line that identifies it as the master version.

```
echo "New content" > file5.c
```

3.) Stage it and commit on the master branch.

```
git add .
git commit -m "adding new file on master"
```

4.) Start up gitk if it's not already running.

```
gitk &
```

5.) Create a new branch but don't switch to it yet. (You can use whatever branch name you want.)

```
git branch newbranch
```

6.) Change the same line in the new file (still on master)

```
echo "Update on master" > file5.c
```

7.) Stage and commit that change (still on master)

```
git add .
git commit -m "update on master"
```

8.) Switch to your new branch.

```
git checkout newbranch
```

9.) Now on newbranch, make a change to the same line of the same file.

```
echo "Update on newbranch" > file5.c
```

10.) Stage and commit it on newbranch.

```
git commit -am "update on newbranch"
```

11.) Switch back to the master branch.

```
git checkout master
```

12.) Merge your new branch back into master. (This will attempt to merge newbranch into master.)

```
git merge newbranch
```

13.) Check the status of things. Notice that we have a conflict.

```
git status
```

14.) Also take a look at the local file and notice the conflict markers.

```
cat file5.c (or type file5.c on Windows)
```

15.) "Fix" the conflict in the file in the working directory. (For simplicity you can just write over it.)

```
echo "merged version" > file5.c
```

16.) Stage and commit the fixed file.

```
git commit -am "Fixed conflicts"
```

17.) Check the status to make sure the merge issue is resolved

```
git status
```

18.) Refresh gitk and take a look at the changes.

19.) We're done with your new branch, so get rid of the branch.

```
git branch -d newbranch
```

20.) Refresh gitk and take a look at the most recent changes.

Lab 6 - Using the Overall Workflow with a Remote Repository

In this lab, you'll get some practice with remotes by working with a Github account, forking a repository, cloning it down to your system to work with, rebasing changes, and dealing with conflicts at push time.

Prerequisites

This lab assumes that you have internet access.

Steps

1.) Go to **https://github.com** and sign in to your Github account.

2.) Browse to the calc3 project at https://github.com/brentlaster/calc3

3.) Click on the **Fork** button (top right) and the repository will be forked to your userid. (Your URL should change to https://github.com/<github userid>/calc3.)

4.) Open up a terminal session (that you can run git in) on your local machine and **cd** back to your home directory (or at least out of any directories that have git repositories in them).

5.) On the screen, near the middle right, should be a green button, labelled "*Clone or download*". Click on that. A new window pops up populated with the URL path you can use to clone this project down via the https protocol. To the right of that command, you'll see a "clipboard" icon. Click on that to copy the path to your clipboard. This saves you from having to construct the path yourself.

6.) Switch back to your terminal session. CD back up a level if needed to make sure you are not in one of the existing projects from the other labs.

Then clone the project down by typing "git clone" and then pasting the path from the clipboard. Hit Enter.

```
git clone https://github.com/<your github user id>/calc3.git
```

7.) You should see some messages from the remote side and then the project will be cloned down into the calc3 directory.

Change (**cd**) into the calc3 directory.

8.) You can now browse around the calc3 directory. There is only one file in there, but if you look at the hidden files, you'll see the **.git** repository that was cloned down from the remote. You can also run commands like branch against them to see the set of branches. Also try the commands below to see the list of remote branches and information about the most recent set of changes in each.

```
git branch -r
git branch -av
```

9.) You can then run the remote -v operation to see the remote.

```
git remote -v
```

10.) (Optional) Let's see what features our calculator already has. Open up the calc.html program in a browser and take a look. You'll notice we have the basic arithmetic functions there: addition, subtraction, multiplication, and division.

11.) We want to incorporate some other features into our calculator program from the features branch. First, we'll setup a local features branch. Create a local branch tracking the features remote branch.

```
git branch features origin/features
```

12.) Let's get a look at what's in the master branch and what features are available for us to use in the features branch.

```
git log --oneline              (to see what's on master)
git log --oneline features     (to see what's on features)
```

13.) We want to merge in the max, exp, and min functions to add to our calculator. We'll do this with a rebase to get the history as well. Still in your master branch, run the rebase command as follows.

```
git rebase features
```

14.) Once that finishes, you can do a quick log of your current branch (master) and see that the history records show up there now.

```
git log --oneline
```

(Optional) You can also open up the calc.html program in a browser and verify that the functions are there as well.

15.) Assuming the rebase was successful, push the updates out to the remote.

```
git push origin master
```

16.) At this point, you'll see an error message about Git not being able to do a fast-forward merge. To correct this, we need to merge in the "more up-to-date" content from the remote. To try this, we can just do a pull operation from the remote.

```
git pull
```

Git will want to create a new "merge commit" for this and will prompt you for a commit message (via the editor). You can change it if you want, but when you are done, close the editor.

17.) In this case, the merge was fairly simple and should have succeeded. Now that we are up-to-date, we can try the push again. This time it should succeed without problems.

```
git push origin master
```