Git Fundamentals

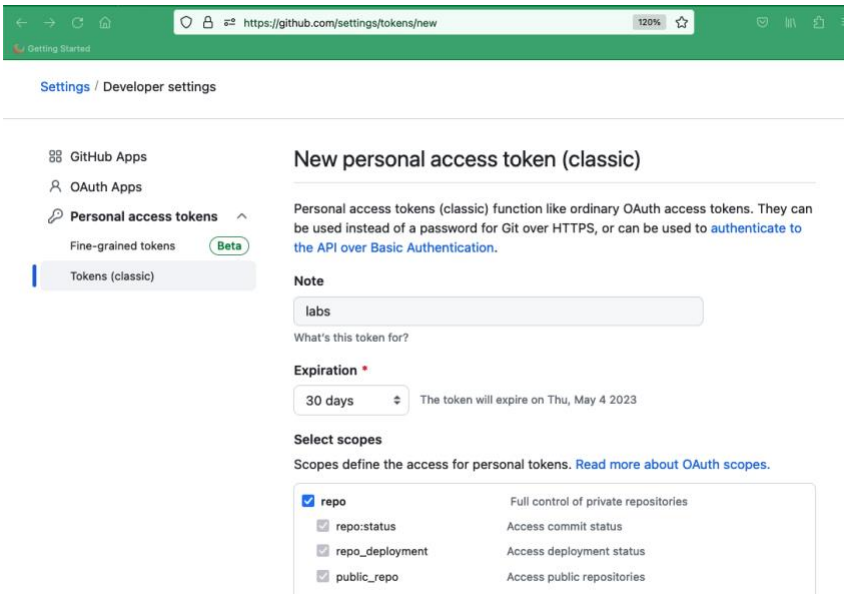
Revision 3.6

06/17/24

Note #1: Prior to lab 6, you will need to have a GitHub free account and a GitHub Personal Access Token (PAT). Follow the instructions at the link below to generate the token. Keep a copy of it somewhere so you can copy and paste it to use in lab 6.

https://docs.github.com/en/authentication/keeping-your-account-and-data-secure/managing-your-personal-access-tokens#creating-a-personal-access-token-classic

Ensure that you click the box to have the **repo** scope selected (see below).



Note 2: If you are using an older version of Git, the default branch may be "master" instead of "main". If this is the case, for Labs 4 and 5, 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 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).

```
$ mkdir some-dir

$ cd some-dir
```

2.) In the new directory, initialize a new repo by running the 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 your-email-address
```

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-name*

$ echo *content* **>** *file2-name*

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 - **main**, 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-name*

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

$ git commit -am "*commit-message***"**

END OF LAB

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-name*

$ 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-name*)

$ 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-name*

$ 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-name*

$ 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 "*commit-message*"

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.

END OF LAB

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-name
$ git commit -am "commit-message"
```

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.

(Note: In some terminals your history may be longer than the screen and need to hit a key to continue. If you 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.) Note this command spans multiple lines - copy and paste may not work as expected.

```
$ 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 *some-existing-file*

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 (top) -> 1db49cf 2016-08-20 | add a line (HEAD -> main) [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 (bottom)-> 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.

$ git hist *earliest-SHA1..latest-SHA1*

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.

$ git diff *earliest-SHA1..latest-SHA1 file1-name*

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 sha1-value*

$ git tag first *earliest-SHA1*
$ git tag last *latest-SHA1*

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 to simply add that filename onto the command. Try out the example below.

$ git hist first..last --name-only *file1-name*

END OF LAB

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 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 "* main". This indicates that there is only one branch currently in your repository - main. 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 "main".

3.) Now, before we work with a new branch, let's update at least one file in the main branch to indicate that this is the version on main 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.

$ echo *main version* **>>** *file-name*

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 "main version"
```

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

```
$ git branch feature-branch-name
```

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

```
$ git branch
```

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

```
$ git checkout feature-branch-name
```

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

```
$ git branch
```

9.) Now create a new file in the feature branch. Then update the files in the feature branch to indicate that they are the "feature branch version".

```
$ echo some-text > new-file
```

Next, pick at least one previous file and update it with a way to indicate this is the version on the new branch.

$ echo *feature version* **>>** *file-name*

10.) 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.)

11.) Switch back to the main branch.

$ git checkout main

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

$ git branch

(Note: Should have a * by main.)

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

$ cat * (or "type *" on Windows)

Look for "main version" in the text.

END OF LAB

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 main version.

```
$ echo "some-text" > file5-name
```

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

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

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

```
$ git branch new-branch
```

5.) Change the same line in the new file (still on main)

```
$ echo "Update on main" > file5-name
```

6.) Stage and commit that change (still on main)

```
$ git add .
$ git commit -m "update on main"
```

7.) Switch to your new branch.

```
$ git checkout new-branch
```

8.) Now on your new branch, make a change to the same line of the same file.

```
$ echo "Update on new-branch" > file5-name
```

9.) Stage and commit it on your new branch.

```
$ git commit -am "update on new-branch"
```

10.) Switch back to the main branch.

```
$ git checkout main
```

11.) Merge your new branch back into main. (This will attempt to merge new branch into main.)

```
$ git merge new-branch
```

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

$ git status

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

$ cat *file5-name* (or "type *file5-name*" on Windows)

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

$ echo *"merged version"* **>** *file5-name*

15.) Stage and commit the fixed file.

$ git commit -am "Fixed conflicts"

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

$ git status

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

$ git branch -d *new-branch*

END OF LAB

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.

Note: If you are already familiar with ssh and have a GitHub ssh key setup, you can use that in place of the https: protocol in this lab. Only do this if you are sure you have ssh already setup and understand how to use ssh keys.

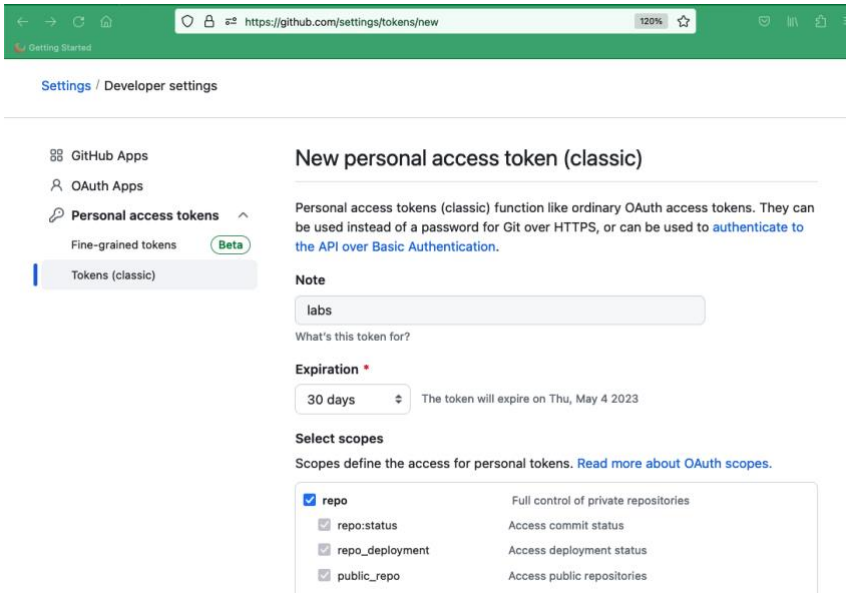
Prerequisites

You will need to have a GitHub free account and a GitHub Personal Access Token (PAT). Follow the instructions at the link below to generate the token. Keep a copy of it somewhere so you can copy and paste it to use in lab 6.

(You can just select the classic token, not the fine-grained one)

https://docs.github.com/en/authentication/keeping-your-account-and-data-secure/managing-your-personal-access-tokens#creating-a-personal-access-token-classic

Ensure that you click the box to have the **repo** scope selected (see below).

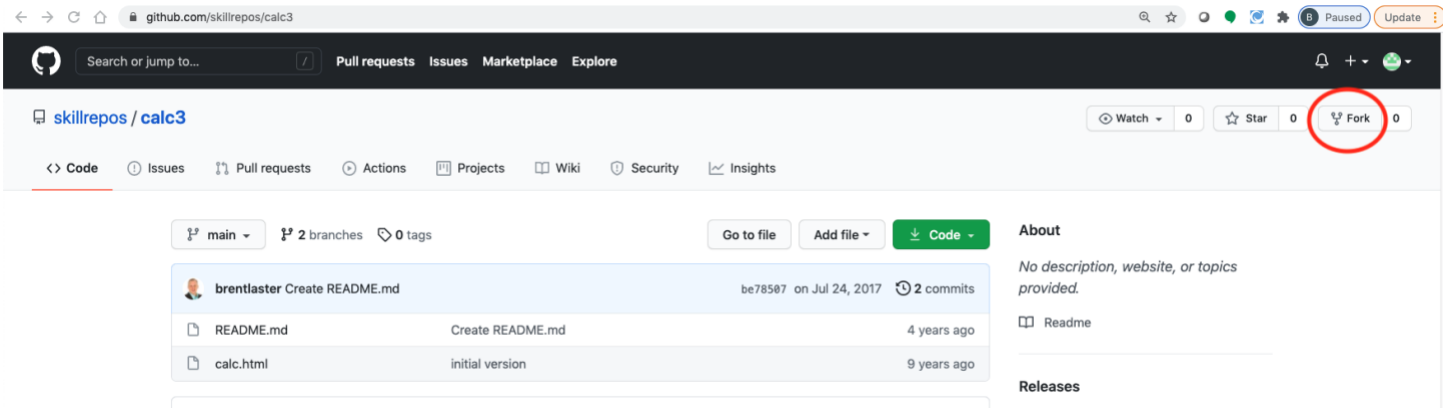


Steps

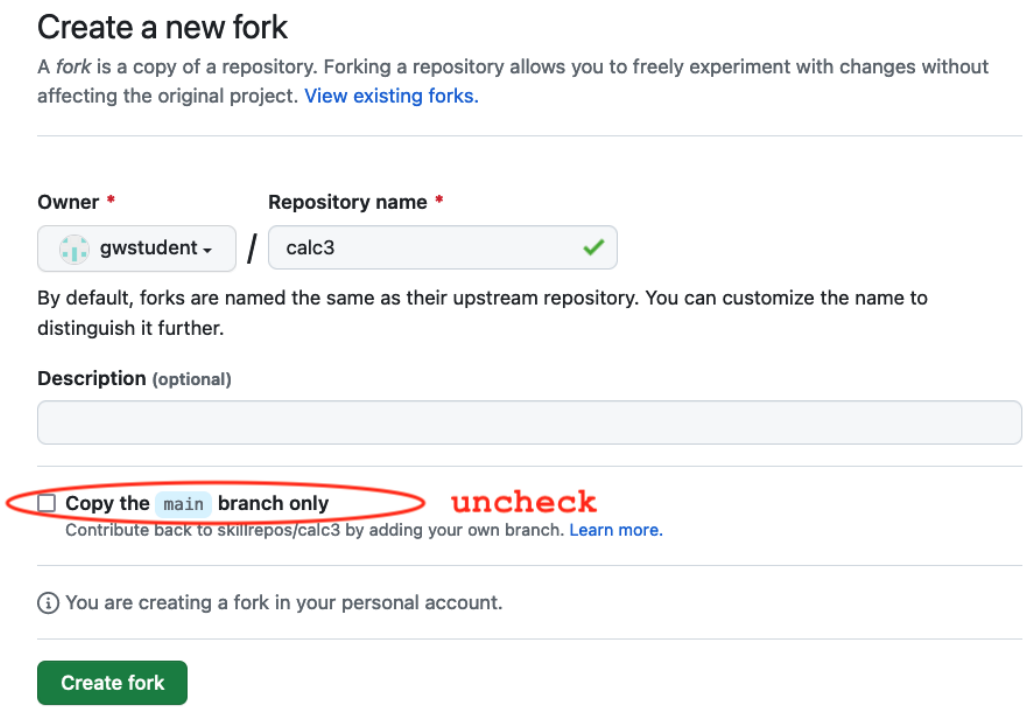
1.) Go to **https://github.com** and sign in to your Github account.

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

3.) Click on the **Fork** button (top right). **Uncheck** the checkbox next to "Copy the main branch only" and click on the green "Create fork" button. The repository will be forked to your userid. (Your URL should change to https://github.com/*your-github-userid*/calc3.)

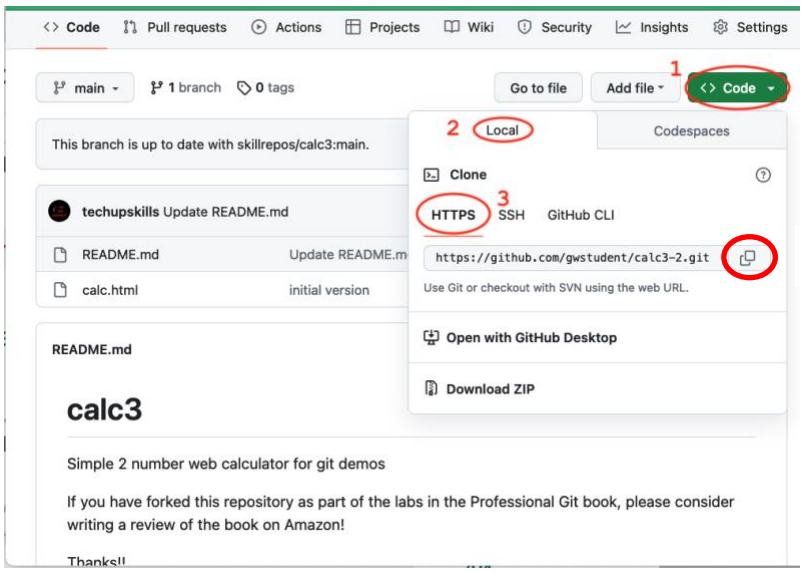


On the next screen with the fork options, **make sure to uncheck the "Copy the main branch only"** box so we will get the other remote branches. Then click on the "Create fork" button at the bottom.



4.) On the GitHub screen, near the middle right, should be a green button, labelled "<> Code". Click on that. A new window pops up with two tabs. Select the one labelled "Local". **Then, unless you already have an ssh key setup, select the "HTTPS" protocol.**

You'll then see a box 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 "copy" icon. Click on that to copy the path to your clipboard. This saves you from having to construct the path yourself.



5.) 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.

$ cd .. (if needed)
$ git clone https://github.com/*your-github-userid*/calc3.git

6.) 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.

```
$ cd calc3
```

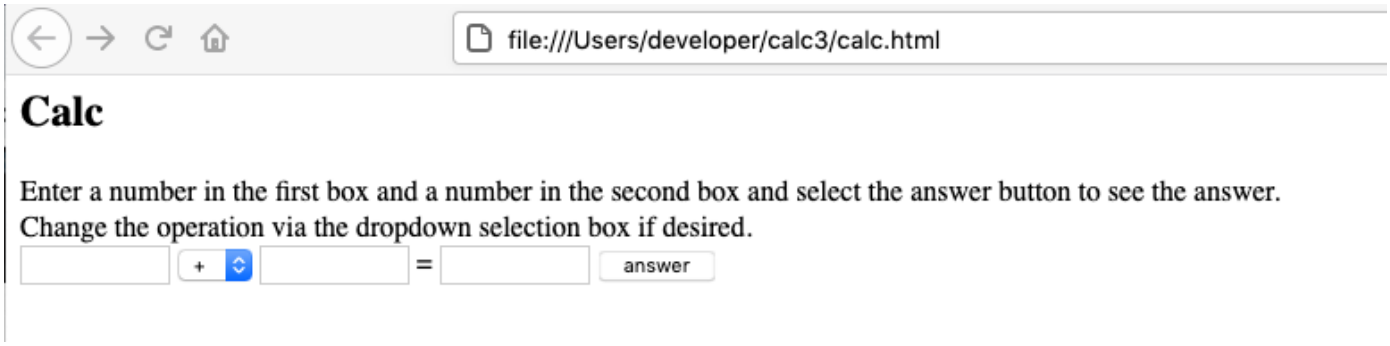
7.) You can now browse around the calc3 directory. There are only two files in there, but if you look at the hidden files, you'll see the **.git** repository that was cloned down from the remote. You can also run commands like branch against them to see the set of branches. Also try the commands below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

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

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

```
$ git remote -v
```

9.) (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.



10.) We want to incorporate some other features into our calculator program from the features branch. First, we'll setup a local features branch to track where the remote branch is in the repo we just cloned. Create a local branch tracking the features remote branch.

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

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

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

12.) 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 main branch, run the rebase command as follows.

```
$ git rebase features
```

13.) Once that finishes, you can do a quick log of your current branch (main) 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.

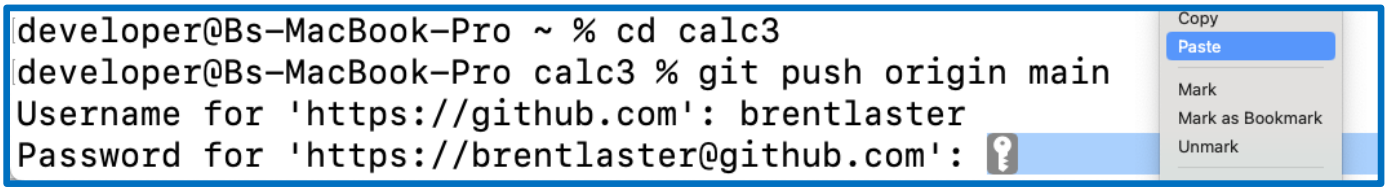
14.) Assuming the rebase was successful, push the updates out to the remote.

```
$ git push -u origin main
```

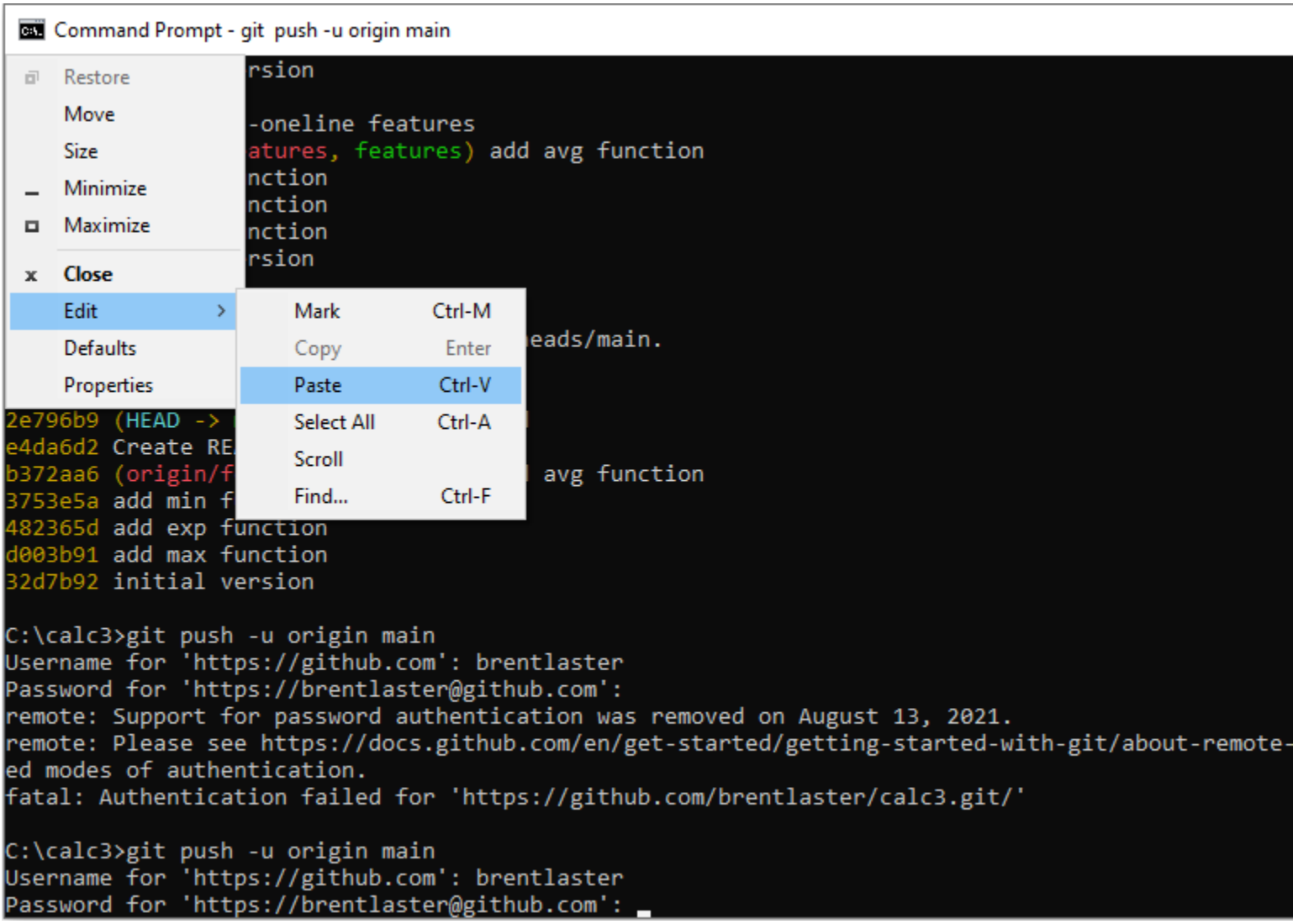
15.) After this, you'll be prompted for username (your GitHub username) and then a sign-in/Private Access Token or password. Wherever it asks for a token or a password, you can just copy and paste in the token you generated in GitHub prior to this lab. An example dialog that may come up is shown below.

If instead, you are on the command line and prompted for a password, just paste the token in at the prompt. Note that it will not show up on the line, but you can just hit enter afterwards.

If you are using a Mac, a Linux system, or the Git Bash shell, it should look something like this:



If you are using a Windows command prompt, you may have to use the command prompt's menu to do this:



16.) At this point, you'll see an error message about Git "rejecting" your push due to a "non-fast-forward" situation. **This is expected.** The rebase created a situation where there is content in our local branch that can't just be fast-forwarded.

NOTE: This isn't exactly what would occur if we ran into changes by another user who got their changes in first, but you get the same experience – rejected because it can't fast-forward.

To get past this, we need to merge in the content from the remote. To try this, we can just do a pull operation from the remote.

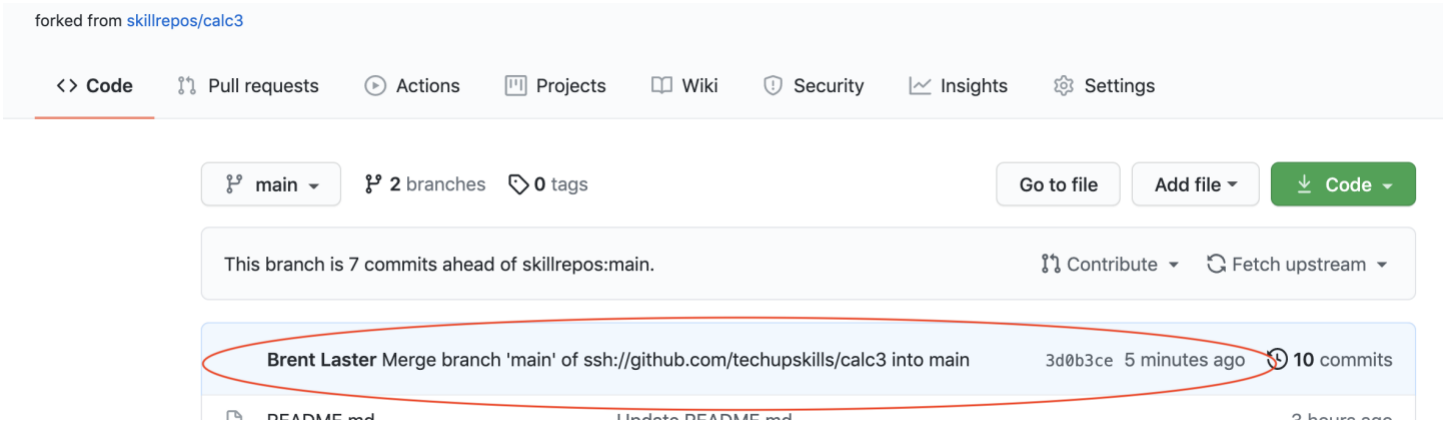
```
$ git pull --no-rebase
```

Git will want to create a new "merge commit" for this and prompt you for a commit message (via the editor). If so, you can change it if you want, but when you are done, just 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. (You'll need to supply your username and token as before.)

```
$ git push origin main
```

18.) If you want, you can look back in GitHub and see the update from the merge.



END OF LAB

That's all - THANKS!

Appendix: Other options for making changes in repo vs https (if the https approach doesn't work for you) – choose one of A or Bif and only if the https push did not seem to work…

A. Reseting credential helpers: Especially on Windows, if you are pasting in your token for the password, but still getting an error message referencing password authentication, you may be running into issues because you have previous credentials stored in the *credential helper*.

One of the things you can try in this case is resetting the stored credentials via:

```
$ git config --global credential.helper store
```

Then you do your push as per the lab. It will probably pop up a text entry box for you to add your username in and another to paste in your password (PAT) and then will replace your credentials with those and complete the push.

(Note: If you prefer to disable the global credentials helper entirely, you can try

```
$ git config --unset --system credentials.helper
```

This may or may not work depending on if you have access to do this.)

B. SSH keys: If you are familiar with using ssh and have keys, you can add them into GitHub and use those. Ref https://docs.github.com/en/authentication/connecting-to-github-with-ssh/adding-a-new-ssh-key-to-your-github-account for more details.

If you go this route, when you get the remote URL from the browser, select the SSH tab.