Weld Lab git and GitHub Primer

Peter Dotti

1 Introduction

This document is intended to establish basic git and GitHub practices in the Weld Lab and to be a simplified introduction to using git and GitHub. The hope is that implementing these practices will enable the following:

1. Editing code without fear of disrupting a working version
2. Easily update versions of code once a testing version of the code is established
3. Inherent record keeping and version recording as automatically carried out by the git software
4. Backup versions (current and historical) of code as saved on both GitHub and the Citadel server
5. A unified procedure so that accessing and using other lab member code is easy

While there is desktop software for GitHub (and, presumably, git more generally), I found it not to be sufficiently intuitive, robust, or easy to justify using it over command line inputs, so this document will focus on use of command line git.

Hopefully it will be easy and painless to do these things. Please email / slack message me (Peter) if you have suggestions to improve this document in the future. Additionally, I hope to use git on this document itself for easy editing.

2 An Important Reference

If you find that this primer is lacking some piece of information that you would like, I strongly recommend using the online "git Book" from `https://git-scm.com/book/en/v2`. I found it to be very well written and well organized, although it will certainly have more details than you need (although, you will likely not need most details past the fourth chapter, unless you wish to do more advanced things).

3 Using git on your Computer

Git revolves around the use of "repositories." A repository is just some directory (folder) on your computer with a hidden git file in it that designates it as a repository and allows you to set and keep track of files.

You are able to change the files in the repository however you'd like, using whatever code editor or program you'd like, but you will regularly want to make a "`commit`" with git so that the version

of the repository is saved. Additionally, you will want to occasional use the "`push`" command to use the current version of the repository on your computer to update some remotely saved version of the repository (i.e., you will use `push` to update a stored version of the code on GitHub and the Citadel)

From here on, I will assume that the reader has some command line tool with git installed on it (instructions on how to do that here: https://git-scm.com/book/en/v2/Getting-Started-Installing-Git). I will also assume that the reader knows how to change directories and get directory path names in the command line.

3.1 Making and Using a git Repository on your Computer

Once you've navigated to the directory that you want to initialize as a repository, you run the command

```
git init
```

At this point, you can fill the directory as usual (assuming it didn't already have code in it). If you want something to test basic git functionality, I would recommend saving a text file in the directory and editing it.

At this point, you can use the command

```
git status
```

And it will show something like

```
On branch master

No commits yet

Untracked files:

  <A list of the files in the directory>
```

The general meaning of this is that git knows that you have files in the directory that you might want to use as a repository, but they are currently "untracked" and git is waiting for you to indicate which files you want it to keep track of edits for (and moreover, which files it should copy over to an external server such as GitHub or the Citadel when you ask it to do so).

To indicate which of the files in the list you want it to track, use the command:

```
git add <filename>
```

for each of the files you'd like. If you want it to track all of the files, you can conveniently use

```
git add --all
```

There are fancier patterns to use. I believe they generally follow the UNIX wildcard characters. Resources exist online to learn them, such as https://geek-university.com/linux/wildcard/. I'll give one example:

```
git add *.m
```

Git will look for any file of the form <SOMETHING>.m, i.e., it will add any .m matlab file to the repository.

You can now run

```
git status
```

again, and you should now see an indication that some files have been added. The jargon is that these files are "staged", and they are ready to be "commited."

Now it is time to use the `commit` command to tell git that you want it to mark this as a point in history that you would like remembered. In other words, the `commit` command saves version of the current repository state.

Use the command

```
git commit -m "<Message describing this commit>"
```

The message you enter should describe this version or why it is different from the last commit.

And that's about it for a repository on your computer! You can now go along and make edits to the files in your repository. Then you make commits whenever you see fit.

The one last note I should make is that whenever a file is edited and you want to make a new commit, you have to once again use the `git add <filename>` command to once again stage the modified files (as if you were adding a new file again), before using the `commit` command. However, it is generally more convenient to skip the `add` step and use

```
git commit -am "<Message describing this commit>"
```

If you missed it, I added an "a" between the - and m. That "a" tells git to stage all of the modified files (but *not* new or untracked files) and make a commit.

3.2 Checking Status and Logs for a Repository

In general, you can get helpful information about the status and history of a repository with the `git status` and `git log` commands.

We have already seen use of `git status` above. It will typically tell you what files have been modified or what new files have been added to the repository since the last commit. It will also tell you whether or not the current commit has been used to update the remote repositories (e.g., GitHub or the Citadel).

`git log` will tell you about the version history of the repository. Specifically, it will list out the previous commits in reverse chronological order. Moreover, it will tell you if you "branched off" other versions and merged them later on in the history (more info later).

I recommend `git log --graph --decorate` which makes a sort of ascii art representation of the log history.

3.3 Looking at an Old Version

Now, suppose you think there was some code in a previous version that was lost or abandoned and needs resurrection. We will now give a simple way to temporarily restore to that old version and view the file. There are other methods, and I am frankly not sure if the `reset` command is more appropriate in most cases, but this seems the least invasive method of what I've learned in my bit of prep.

First, you will want to get the identifying number for the old commit that you want to see. To get it, run `git log`. Each commit will have a number (in hexadecimal) after it. It will look like `commit 9a52cbc45aa6f5c6064c654ef7951081930274fb` for example.

You can now use the command

```
git checkout 9a52cbc
```

(you only need the first 7 digits). It will give some statement about going into 'detached HEAD' state (which sounds worse than it is) along with some helpful advice.

Your files will now be in the state of the old commit corresponding to the number you gave it. You can open the code from the repository directory in the usual way and it will be like you've gone back in time to that old commit. You can even make a new branch with edits from it to save that state and use it as something to work on separate from the newest version.

To go back to the newest version, run

```
git checkout <branchName>
```

`<branchName>` will almost certainly be `master`, unless you have changed it (or want to checkout a different branch). To see what named branches you have, you can run

```
git branch
```

and it will return a list of branches (more on branches in the next section).

3.4 Branches and How to Make Them

I would like to keep a discussion of branches as brief as possible here. A fuller introduction is at https://git-scm.com/book/en/v2/Git-Branching-Branches-in-a-Nutshell. However, I believe it is necessary to know what a branch is to use remote repositories on GitHub and the Citadel, and this is one of our main goals.

As we saw before, git allows you to restore a repository to the state at a previous commit. A natural question is "Instead of just working with previous and current versions of a code set, is there a way to have multiple different working versions of code that sort of branched off from an earlier version?" The answer is yes, and git calls these different working versions "branches."

These branches can be especially useful if you want to keep a working version of the code as the master branch (git's preferred term for it) while editing in added functionality, that you want to be using and testing without corrupting the master version.

To make a new branch with the name `<branchName>`, enter

```
git branch <branchName>
```

Now to see what branch you are currently using, as well as all available branches, enter

```
git branch
```

You should see a list of branch names, one of which will have a * next to it, indicating that branch as the one you are working with. You will probably see `* master`, since this is the default branch name for the first branch (it is also possible that "master" will be some different name). Also on the list will be the new branch you just made. To switch to the new branch, enter

```
git checkout <branchName>
```

Now you can make commits as usual, and these changes will be tracked on this branch. In essence, this branch represents one version of the code that you can alter and play with.

If you want to switch between branches, you need only type

```
git checkout <someBranchName>
```

This will entirely change the state of your directory.

One note on switching branches - depending on the type of file and editing software, you might need to close and reopen the file you are looking at for it to represent the new branch. I don't believe this will be a problem with any code editing software, but it might be necessary if you are saving text files or something in your git folder.

3.5 Merging Branches

Lastly, I will discuss merging branches. Consider the following situation: I made some edits on this new branch I was working on. I think it's good now and I'd like to combine it with the master branch (or some other branch). However, the master branch has been updated by me or someone else since I started working on this new branch. I would like to combine them somehow.

git has you covered with the `merge` command. If you have two branches, branch1 and branch2, and you want to combine the two branches, you run

```
git checkout branch1

git merge branch2
```

Important note: The previous is not perfectly symmetric under exchange of branch1 and branch2. Specifically, branch1 is updated to the merged file, whereas branch2 remains as it was. This is important to remember because, as is usually the case, you will want to update the "master" branch, and so you will want to have branch1 replaced with master in the previous code.

When you try this merge, it is likely that there will be some conflicts between the files (some differences that git did not feel it could resolve automatically). Git will inform you of this, and ask you to fix them. To do this, you simply open the file(s) it says there are conflicts in, and you look for the part where it says something like

```
<<<<<<<<<<<<<<<<<HEAD
<some things from branch1>
======================
<some things from branch2>
>>>>>>>>>>>>>>branch2
```

Then you edit this section how you'd like. Once finished, you need to tell git to commit these changes. I recommend checking the status with `git status` at this point to see if there are any files that need to be restaged or added using `git add <fileName>` (maybe `git commit -am "message"` works as well). Then run

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

You can confirm that git seems to have done what you wanted by running

```
git log --graph --decorate
```

It should show how the code branches diverged and were brought back together. It will also tell you what commit (version) `branch2` is on.

3.6 Rebasing a Branch

I will not explain how to rebase a branch here. I consider it outside of the scope. It can be useful, but it is *strongly* encouraged that you *do not* use rebase when updating a remote repository that is stored on GitHub or the Citadel. Otherwise, it can make a mess of the merge history if other people are making code edits on their own computer.

So that you have a vague idea what it is though, the `rebase` command allows you to basically take some edited `branch2` and "stack it on" to the end of some other `branch1`, as if you just made edits on `branch1` to get to the final combined version of the two branches. This will sort of make it so that history ignores the time when `branch1` and `branch2` diverged, and were perhaps edited.

I believe that the way we will use git will not benefit much from this feature really, but in principle, it can make for a sort of simplified log history that makes it easier to find and go back to the old commits that you might be interested in.

For the interested reader, a much better explanation is at https://git-scm.com/book/en/v2/Git-Branching-Rebasing.

4 Using git with Remote Repositories: An Application to GitHub and the Citadel)

One of the primary benefits of using git is that you can have remote repositories that can be updated with updated version from multiple programmers. The general procedure is that you set up one (or multiple) of these remote repositories, and you can (1) copy updated commits from the remote repository that other people have edited or (2) make your own edits to the remote repository from your commits.

GitHub is, at its core, just a collection of such remote repositories. However, you are free to designate practically any directory as a remote repository, whether it be on your personal computer, some shared computer, or a group server.

In this section, I hope to give the basics on using and setting up remote repositories. More importantly, I would like to establish some best practices for the situation in the Weld Lab, which I believe is to have duplicate remote repositories on the Citadel and GitHub, since I believe it will be easy to maintain, and has some benefits in terms of organization and backups against data loss.

4.1 Cloning from an Existing Remote Repository

First, a simple example. Suppose a repository exists on GitHub at `https://github.com/pdotti/aGitRepo`. Opening this page, you will see that GitHub has a bright green "Clone or download" button. Clicking this will tell you the url for this repository, namely `https://github.com/pdotti/aGitRepo.git`. To clone a working copy onto your computer, you need only (1) open a command line in the directory where you want the repository to be put, and (2) enter the command

```
git clone https://github.com/pdotti/aGitRepo.git
```

And then you have a working repository for you to make commits on, branch into new versions, and use to update the original `https://github.com/pdotti/aGitRepo.git` on GitHub, but we will get to that in a later subsection.

Now instead suppose that the remote repository is on the Citadel. Specifically, as I hope will be come common practice, it is in the `GitRemotes` directory that has been made on the WeldLab2 volume of the Citadel (aka the citadel expansion). This will have some directory path. When I mount it with afp on my Mac, the directory path is `/Volumes/WeldLab2/GitRemotes`. An example remote repository in this location might be `/Volumes/WeldLab2/GitRemotes/aGitRepo.git`. You can clone this repository in much the same way as from GitHub with the command

```
git clone /Volumes/WeldLab2/GitRemotes/aGitRepo.git
```

Moreover, for organizational purposes, you can add sub-directories (folders) inside the `GitRemotes` directory to help organize the repositories (a feature GitHub lacks for some reason). For example, you could put the folder `testRepos` in the `GitRemotes` folder to hold all of the repos we use for testing. Then the path to clone from would just be `/Volumes/WeldLab2/GitRemotes/testRepos/aGitRepo.git`

NOTE: You *cannot* use these remote repositories the same way as you use a normal repository. The reason is that they are "bare" repositories, so you are not able to open the files in them from the finder / file explorer and edit them. Instead, if you want to work with them, you are supposed to clone them to some other location on your computer and use that cloned repository to make edits and commits, and then use those commits to update the remote repository. I believe otherwise it risks muddying up the commit history somehow... In any case, git will complain at you if you try to use a non-bare repository as a remote repository, and I presume it has a good reason for this. This is not to say that you can't have regular repositories on the citadel. Whether a repository is remote or not doesn't depend on where the data is physically stored, just whether or not it is designated as such.

4.2 Making a Remote Repository on GitHub

Instead, suppose you have an existing repository and you want to put it onto GitHub. First, log into GitHub and figure out where you want to put the repository — For example, you might go to `https://github.com/weldlabucsb`. Then, on the webpage, use the prominent, green "New" button to make a new (bare) repository on GitHub. Follow the instructions to give it a name. GitHub will then prompt you to set up this repository. To do this, use the following code from the existing repository on your computer (which is very similar to the " ...or push an existing repository from the command line" setup instructions that the GitHub website gives):

```
git remote add <remoteName> <URL>
git push -u <remoteName> <myLocalBranch>
```

The `<URL>` will be exactly what GitHub says it should be (e.g., `https://github.com/weldlabucsb/aGitRepo.git`) The `<remoteName>` is up to you, but I recommend `github`. This will basically be your nickname for this repository. By default, git likes to call it `origin`, but since there is more than one server we are likely to update to, the default seems confusing.

`<myLocalBranch>` will almost always be `master` at setup, unless you change the defaults. However, in general, you might have multiple branches that you want tracked by the remote repository (in essence, the remote repository gets multiple branches.) In that case, `<myLocalBranch>` would be a branch with a different name (such as `branch1` or `branch2` if we go back to the example branch names used in the branches section above.

4.3 Making a Remote Repository on the Citadel

Putting a remote repository on the Citadel requires a few extra initial steps, but is otherwise quite similar.

First, you need to create a "bare" repository from the existing repository. To do this from the command line run

```
git clone --bare <existingRepoDirectoryPath> <newBareRepoFile>.git
```

Running this initially confused me slightly because you do not necessarily run it from within the existing repository. If you happen to be in the existing repository, you should replace `<existingRepoDirectoryPath>` with a single period `.` which is the UNIX shorthand for "the current directory". Otherwise, it is just a path to the repository.

The resulting bare repository will be in the file `<newBareRepoFile>.git`.

Next, you will want to move (drag and drop) this new `<newBareRepoFile>.git` repository into its location on the Citadel. I recommend it be somewhere in the `GitRemotes` directory on the Citadel (in WeldLab2 aka the expansion).

Then, navigate to this `<newBareRepoFile>.git` repository in the command line (it acts as a normal directory), and run the command

```
git init --bare --shared
```

Frankly, I'm not sure this is necessary, but the reference I used said that this step should be done to configure it properly.

Now, open the existing repository that you used to make the bare repository in the command line. To tell your original repository where the new remote repository is, you run

```
git remote add <remoteName> <PATH/newBareRepoFile>.git
git push -u <remoteName> <myLocalBranch>
```

This is basically the same as the GitHub case, except the `URL` in that case has been replaced with a file path to the Citadel repository that you just made ¡PATH/newBareRepoFile¿.git. For example, the path might be `/Volumes/WeldLab2/GitRemotes/aGitRepo.git`.

In this case, I recommend that `<remoteName>` be `citadel`.

Again, `<myLocalBranch>` will almost certainly be `master` unless you are adding a new branch.

4.4 Pushing ("uploading") and Pulling ("downloading") from a Remote Repository

5 Quick Overview of Commands from this Primer