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. Collaborative access to shared code run on multiple machines
2. Editing code without fear of disrupting a working version
3. Easily update or revering versions of code after experimenting
4. Inherent record keeping and version recording as automatically carried out by the git software
5. Backup versions (current and historical) of code as saved on both GitHub and the Citadel server

While there is desktop software for GitHub (and, presumably, git more generally), the command line version of git is very simple and intuitive, gives pretty easy to read output, and has very helpful warnings and failsafes built in to prevent you from doing something foolish. Additionally, the command line offers full functionality, whereas the desktop apps are a restricted subset. For this reason, I recommend using the command line and have written this document for that. I do encourage experimentation with desktop applications if you'd like, especially for more graphical version edits - this document should still define the functionality in those applications.

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 contains some lacuna for your purposes, 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. The hope is that this document will highlight the key features for our purposes and expedite learning.

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 use a repository like a regular folder with code in it. Edit the code (or general files) with your preferred software, and save it like normal without any added thought.

To use git, you will occasionally run a "`commit`" command in the repository to tell git to save a version. The state of the folder when you committed it is then stored in a log history, and you can easily access any old versions.

Additionally, once you have a repository, you can use it in conjunction with a remote version of the repository. A remote version is a repository saved on GitHub or some other server, e.g. the citadel. The state of the local repository and remote repositories can diverge, say, from different people editing them, but git is set up to allow you to easily update the remote version using your local edits, or vice versa. This is done with the `push` command to put your edits on the remote repository, or `pull` to take the remote edits and put them on your local machine. Git will let you know if there are conflicting edits to both versions, and let you manually view them and edit into them into a desirable combined version, which you can then use to update the local repository, remote repository, or both.

3.1 Installing git

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.

3.2 Making and Using a git Repository on your Computer

First, change the working directory in the command line so that you are using the directory that you want to turn into a repository. Then run the command

```
git init
```

The directory is now a repository.

At this point, you can edit the files in the directory (which is now also a repository) as you normally would. 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 keep track of (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). When you later make commits,

only the files you have told it to track will be saved in the version history, so make sure you don't forget any files! Fortunately, git is quite good at make missing files hard to ignore, unless you have explicitly told it to ignore them with a `.gitignore` file.[1]

To tell git to track the file `<filename>`, use the command:

```
git add <filename>
```

You can go through and do this for each file in the repository. 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 convention. Resources exist online to learn them, such as https://geek-university.com/linux/wildcard/. I'll give one relevant 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.

To see that this process worked, you can again run

```
git status
```

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 briefly describe the version saved in this commit, with the intention of being able to understand easily from the message, what that version is, and whether or not you might want to revert to it.

An important note, 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 again stage the modified files before they will be saved in the new commit. However, you can conveniently skip this step with

```
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 and then make a commit (it will *not* stage any new files though).

3.3 Checking Status and Logs for a Repository

In general, you can attain the status and history of a repository with the `git status` and `git log` commands.

[1] I will not explain .gitignore in this document, but it is pretty easy to figure out from looking up.

We have already seen use of `git status` above. It will 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).[2]

`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 on branches in section 3.5).

I recommend

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

which makes an ascii art representation of the log history with all of the branches.

3.4 Looking at an Old Version

As a relevant example situation, 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 files as they were in that previous state.[3]

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 of the commit number). 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 separately 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).

[2] More information on remote repositories in section 4

[3] I should say that there are other methods for this than I give here, and I am frankly not sure if the `reset` command is more appropriate in most cases, but the example given here seems the least invasive method of what I've learned in my bit of prep. You can also probably make a branch with a name out of the old version with something like `git checkout -b ...` which might be useful.

3.5 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 on a test branch 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. 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 repository directory to the version that is on that branch.

Of note, depending on the type of file and editing software, you might need to close and reopen the files in your repository after switching branches or checking out an old version. Matlab seems to automatically change the displayed files for you without making you reopen them, but you might run into this with some simple text editors or something.

3.6 Merging Branches

Lastly, I will discuss merging branches, which is when you want to combine two branched versions of code. 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 frequently the case, you will want to update the "`master`" branch, and so you will want to replace `branch1` 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 into the final form that you would like the code to have after the merge. 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>
```

Then run

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

(`git commit -am "message"` probably works as a shortcut for the previous two lines as well).

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 them back together. It will also tell you what commit (version) `branch2` is on.

3.7 Rebasing a Branch

I will not explain how to rebase a branch here. I consider it outside of our scope. You may want to be aware of it though.

To give a vague understanding, 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 the log history ignores the time when `branch1` and `branch2` diverged, and were perhaps edited.

Rebasing 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.

In principle, the `rebase` command 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 fuller 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 edits from multiple programmers. The general procedure is as follows: You set up one (or multiple) of these remote repositories, and you can then (1) copy updated commits from the remote repository to your local repository or (2) make updates to the remote repository based on commits you have made on your local repository.

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 standard practices for the situation in the Weld Lab. I believe a good option for us is to have duplicate remote repositories on the Citadel and GitHub. The duplicates are easy to maintain and the there are benefits in terms of organization and backups against data loss.

4.1 Cloning from an Existing Remote Repository

4.2 Clone from GitHub

First, a simple example. Suppose a repository exists on GitHub at
https://github.com/weldlabucsb/Weld-Lab-Git-Primer. 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/weldlabucsb/Weld-Lab-Git-Primer.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/weldlabucsb/Weld-Lab-Git-Primer.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/weldlabucsb/Weld-Lab-Git-Primer.git` on GitHub. How to do this is explained in section 4.8.

4.3 Clone from Citadel

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/00_GitTutorials/Weld-Lab-Git-Primer.git`. You can clone this repository in much the same way as from GitHub with the command

```
git clone /Volumes/WeldLab2/GitRemotes/00_GitTutorials/Weld-Lab-Git-Primer.git
```

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). The above git repository has such a subdirectory. It is `00_GitTutorials`.

4.4 A Note on Remote Repository Use

You *cannot* use these remote repositories the same way as you use a normal repository. That is to say, you are not able to open the files in them from the finder / file explorer and edit them. The reason is that they are "bare" repositories, which is what git requires of remote repositories. 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. Then you can use those commits to update the remote repository.

There might be work arounds, but this 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.

4.5 Seeing What Remotes You Have Setup

At any point, you can see what remote repositories you have setup. This is done with the command:

```
git remote
```

Or, if you want added information about the exact URL or file path of the remote repositories

```
git remote -vv
```

Information about what branches are "tracking" which remote branches can be attained with

```
git branch -vv
```

Additionally, the remote repositories are shown when you use the `log` commands, but of course, they could need updating if you haven't told git to check the server / citadel. (See section 4.8 for updating your local repository with the `pull` command)

4.6 Making a Remote Repository on GitHub

Instead, suppose you have an existing local 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:[4]

```
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 want to use, the default seems confusing.

`<myLocalBranch>` will almost always be `master` at setup, unless you change the defaults. 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 different branch name.

4.7 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 to making one on GitHub.

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
```

Unlike most git commands, 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 a file path to the local 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.

[4] This is very similar to the " ...or push an existing repository from the command line" setup instructions that the GitHub website gives

Now, go back to the original local repository that you used to make the bare repository in the command line. To tell your original repository where the new remote repository is and make a first update to it, 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`.

As with making a remote repository on GitHub, `<myLocalBranch>` will almost certainly be `master` unless you are adding a new branch.

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

Now that you have a remote repository, you will want to either update the remote repository from your edits, or get versions from the remote repository that others have made.

4.8.1 Push

To update the remote repository, you use the `push` command:

```
git push <remoteName>
```

The newest commit on your machine will be used to update the history on the remote repository.[5]

This is simplest when you only ever push the `master` branch. You might make different branch versions of the code that you want also available and stored on the remote repository. You will need to tell the remote repository to keep track of those branches as well. Otherwise, it will only keep the `master` branch and the commit version history of this branch.[6]

If you want to add one of your branches to the remote repository, you just use the

```
git push -u <remoteName> <myLocalBranch>
```

command again, but for `<myLocalBranch>`, put the appropriate branch name. (NOTE: you will need to be on the branch that you want to put on the remote branch by using the

```
git checkout <myLocalBranch>
```

command. I think git will let you know if you mess this up though and tell you how to do it correctly.)

[5] `<remoteName>` will either be `github` or `citadel` if you use the suggested name convention.

[6] A subtlety here is that if the version history of the master branch includes multiple branches, the versions on both of those branches will also be available.

4.8.2 Pull

When you want your local repository to be updated with the edits that other collaborators have made to the remote branch, you can use the `pull` command:

```
git pull <remoteName>
```

This command checks all of the branches you have on the remote `<remoteName>` and updates all of the local branches on your local repository.[7]

If you only want one of the branches updated, you can use

```
git pull <remoteName> <branch>
```

There is a possibility that you and the person who edited the remote repository have made conflicting edits since you last `pull`'d. When you try the pull request, git will tell you this, and ask you to resolve the conflict in the same fashion as when using the `merge` command (see section 3.6)

4.8.3 Fetch and Merge - Alternative to Pull

Now, `pull` is simple, fast, and what you need most of the time. However, maybe you want to check what new versions are in the remote repository before you use them to update your own.

To check the remote repository, you use the `fetch` command.

```
git fetch <remoteName>
```

It will give you the status of the branches in the remote repositories (i.e. whether or not they are up to date or have been edited since you last pulled them). If a change has been made, it will say something like

```
<branch>   -->   <remoteName>/<branch>
```

This means that there is an edit that has been made to the branch `<branch>` on the remote repository, and it has saved this version to `<remoteName>/<branch>`.

You can then save this new version to a new branch and checkout that new branch all in once with

```
git checkout -b <remoteBranchCopy> <remoteName>/<branch>
```

Now, you are free to open the files that have been freshly updated with the edits from the remote repository. Your branch has not been updated with these edits yet. To return to that version of the folder, just checkout the branch again, using

```
git checkout <branch>
```

Once you have looked at the two versions and decided that you would like to make the edits to your version, you can merge the two branches with the same commands as shown in section 3.6.

```
git checkout <branch>
```

```
git merge <remoteBranchCopy>
```

[7] Of course, it only updates the branches that you have told git to "mirror" on the remote repository with the `git push -u ...` command.

5 Quick Overview of Commands from this Primer

Inputs in square brackets are options that change how much information git shows you about the action, but not the effect of the command.

Checking Status

```
git status [-vv]
git log [--graph] [--decorate]
git diff
```

(shows changes between current version and last commit)

Create a repository out of the current directory (not for creating a remote repositories)

```
git init
```

Staging files so that they are updated in the next commit

```
git add <filename>
git add --all
git add *.<fileSuffix>
```

Commits

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

(Stages all modified files and commits in one step. Saves using `add` command.)

Branches

`git branch` (See what branches you have.)
`git branch <branchName>` (Make a new branch called `<branchName>`)

```
git checkout <branchNameORcommitNumber>
```

(Switch to `<branchName>` OR the old version with commit number)

Merging branches

The following two commands changes `branch1` into a merge of the two branches.

```
git checkout branch1
git merge branch2
```

Cloning a remote repository onto your computer

```
git clone <githubURL>
git clone <directoryPathForRemoteRepositoryOnCitadel>.git
```

Checking remote repositories that your local repository is linked to

```
git remote [-vv]
```

New GitHub repository Setup

After initializing a repository on GitHub (by just making one on the website)

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

New Citadel Repository Setup

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

Move `<newBareRepoFile>.git` to the appropriate folder on the citadel.
Run the following command in that citadel folder.

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

Then, from the original working repository that you are linking to the remote repository, run

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

Using Remotes - Updating them and copying updates from them

`git push <remoteName>`
Updates the remote, `<remoteName>`, with the linked branch in your local repository.

`git push --all`
Update all remotes

`git pull <remoteName> <branch>`
Update your local branch `<branch>` based on the remote `<remoteName>`

`git pull <remoteName>`
Update all branches based on the remote repository `<remoteName>`

`git fetch <remoteName>`
Check remote for edits (but do not update)

`git checkout -b <newBranch> <remoteName>/<branch>`
Make a new branch `<newBranch>` that is a copy of `<remoteName>/<branch>` and check it out