Git Gud At GitHub

Introduction

Although Git and GitHub are essential to developers, their utilities are not limited to people with programming experience. Whether you are writing an essay, making a powerpoint, or coding an application, Git and GitHub can help make your life easier, especially when working on group projects.

Git is version control system that helps keep track of all changes in your files. No matter what project you are working on, you will constantly be making edits, trying new ideas, or restarting on old ideas. How many times did you make changes to something you are working on only to realize it wasn't as good as what you did earlier, but you were unable to remember what you did earlier exactly? By keeping track of your changes, Git offers you the flexbility to explore new directions and also the security of keeping old work in case you wanted to go back to them.

GitHub is the largest hub or single point of open source projects. It is a platform for sharing work and allows groups and communities to work on projects together with ease. Think of it as a social network like Instagram, but instead of posting pictures, people can post all the cool stuff they have been working on for others to see or to work on together. GitHub runs on Git and is also capable of keeping track of all the changes you make to your projects.

This post offers an introduction to the utilities of Git and GitHub for people with or without programming exprience. It is by no means comprehensive but I hope it does help you get a basic understanding of what Git and Github is and how to use them so that further exploration will not be as daunting.

Some Definitions

These are some of the terminology that you must get familiar with when using Git and GitHub. Keep in mind that these are high-level definitions and do not actually go into the details of the data structures and technicals of how a repository is made.

- *Repository*: A git repository is a data structure that holds a set of commit objects and references to each of the sets. You can think of it as the master file that holds all the changes you ever made to your project. There are two types of repositories, local and remote. A local repository is one that exists on your own computer, it resides inside your project folder. A remote repository is a repository that is online, it resides in some GitHub account. This post will go through how to use both repositories.
- *Commit Object*: A commit object consists of a set of files that reflect the project state at a given point in time. Think of this as a 'version' of your project. Say your project had Files A and B. You modify File A and make a commit to your git repository. This creates a commit object that holds the modified File A and unmodified File B the git repository keeps track of as a version of your project. See more examples in the sections below.
- *Commit*: A commit is when you make modifications to your project files and you tell your git repository to keep track of the current state of the project. It is a **snapshot** of your project at a given point in time. Each commit will be assigned a unique id made of letters and numbers (technically called a SHA1 Name) which you can use to reference specific versions of your project.
- *Branch*: A branch is a 'pointer' to a commit. When your git repository is initialized, a *master* branch is already made for you to keep track of commits. A branch will always point to the latest commit you made while on this branch. Think of making a new branch as giving yourself a new copy of your commits which you can modify to your content without worrying about breaking your project. Branches are useful so that you can try out multiple ideas for a project in different states without getting them confused. It is also useful if multiple people want to work on the same project at the same time. This might be a confusing concept, so check out the references section for some helpful links!
- *HEAD*: The HEAD is a reference to the currently checked out commit. Usually, it is referenced at the same point as where your branch is. However, if you check out a commit by its commit id instead of by a branch name, you can end up in a detached HEAD state, explained later in this post.

Using Git Locally

Before we get to working with remote repositories and open source projects, lets first walk through some of the basics on how to use git locally.

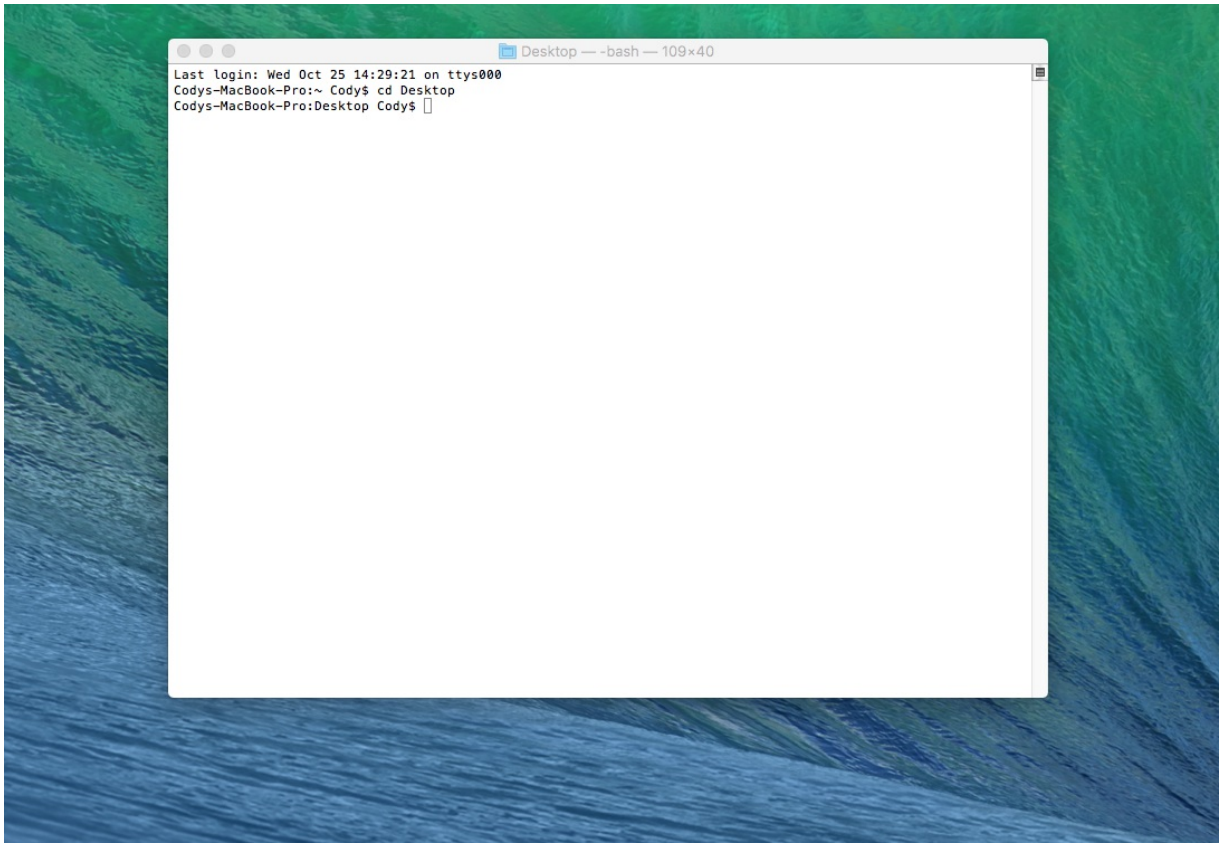
The Terminal

The majority of interactions with git will be done through the command line. This is a program that takes in text based commands instead of working on a GUI with your mouse. This is also called the Terminal on a Mac and can be downloaded for Windows users. It will be very very helpful for you to learn how to work with the command line in order to use git.

A quick guide of some important terminal commands

- `ls`: This lists the contents of the directory (folder) that you are in on the command line. Keep in mind, some files are hidden and need special flags to be seen with ls. For example `ls-a`.
- `cd`: This changes the directory that you are in to one that is specified.
- `mkdir`: This creates a new directory in the current directory with a specified name.
- `touch`: This creates a file in the current directory with a specified name.

(For those who are more interested, I listed some websites in the references section that can help you better understand what a command line is and how shells work together with command lines)



What the Terminal Looks Like

Initializing Git

Download git to your computer.

https://git-scm.com/

After having git on your computer set up some essential git configurations with the following commands in your terminal.

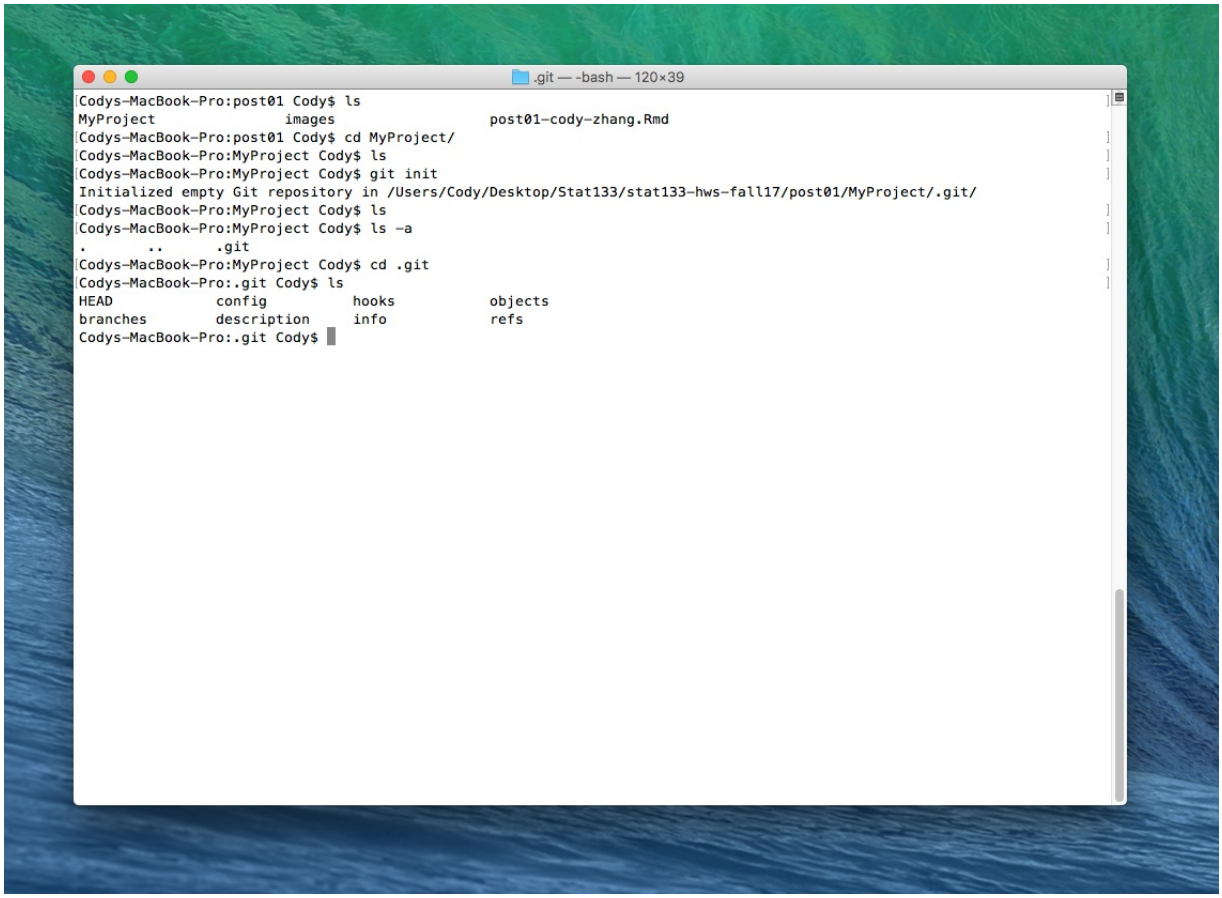
```
git config --global user.name "Your Name Here"
git config --global user.email "your_email@youremail.com"
```

This information will help git know who to credit the commits of your project to.

The first step to start using git locally with your project is to initialize a local git repository inside your project's directory. A directory is just another name for folder, so your project's directory is the folder that contains all your project's files. To do so type the following command in your terminal (Note, words following a '#' are part of a comment, it is not part of the command and is just there to help clarify what the actual command does).

Make sure you are in your project directory before you call git init!

```
#Changes directory to your project directory
cd MyProject

#Initializes the git repository in a directory called .git
git init
```



Git Init

As you can see this initialized a directory called *.git* inside your project directory. The .git directory is where your local git repository is stored.

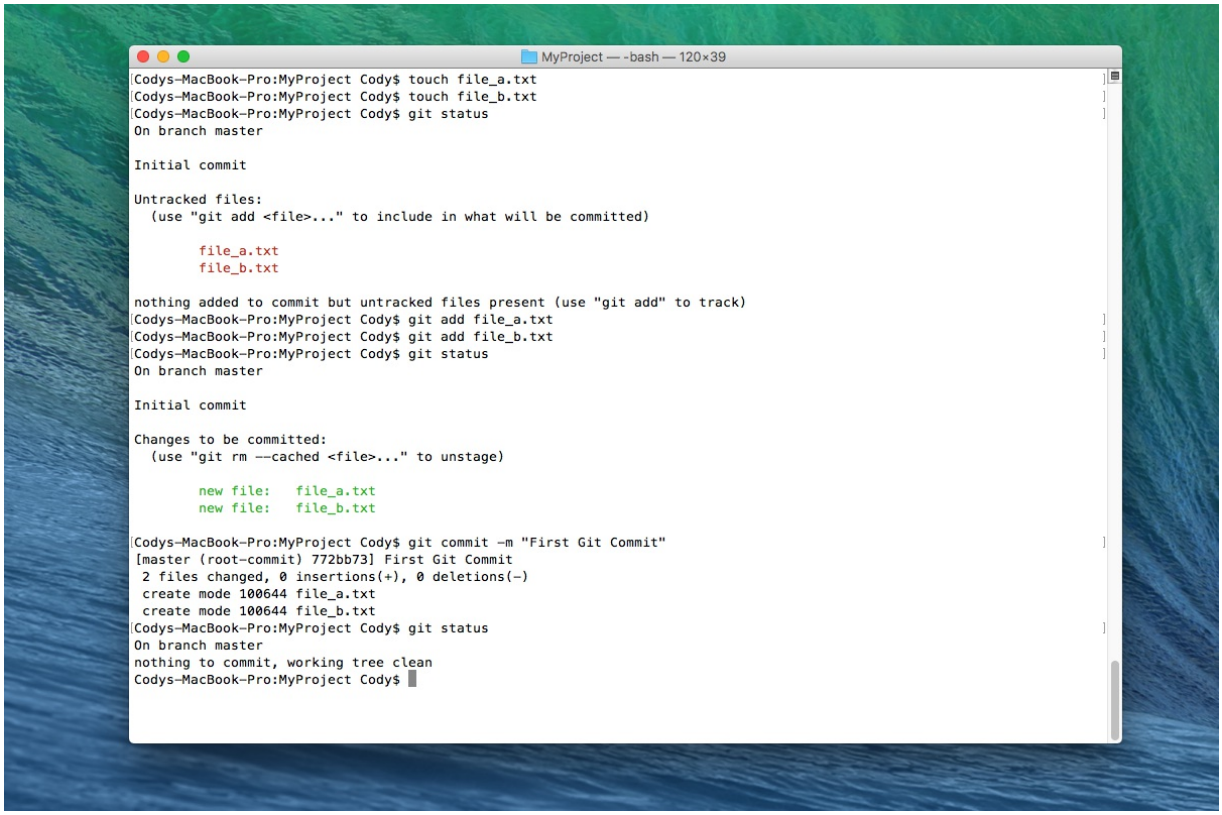
First Git Commits

Let's create some files for the project and start using git to keep track of your changes!

Change to your project directory then run the following commands in your terminal.

```
#Create some files
touch file_a.txt
touch file_b.txt

#Check what the staging status of your working directory is
git status

#Add the created files
git add file_a.txt
git add file_b.txt

#Check your staging status again! What changed after calling git add?
git status

#Commit them to your local git repository with a message
git commit -m "First git commit"

#Check your staging status again! What changed now after the commit?
git status
```

Here's a screenshot that shows you what each of the commands should output in your terminal



First Git Commit

As you can see, after creating our files, the `git status` is that we have untracked files. `git add` specifies which files we want your git repository to keep track of for this commit. After calling `git add`, the status returns the which files were newly added to your commit object and then you can actually commit it to your git repository with `git commit`. Also, note how the commit tells you how many files were changed and how many lines were inserted or deleted.

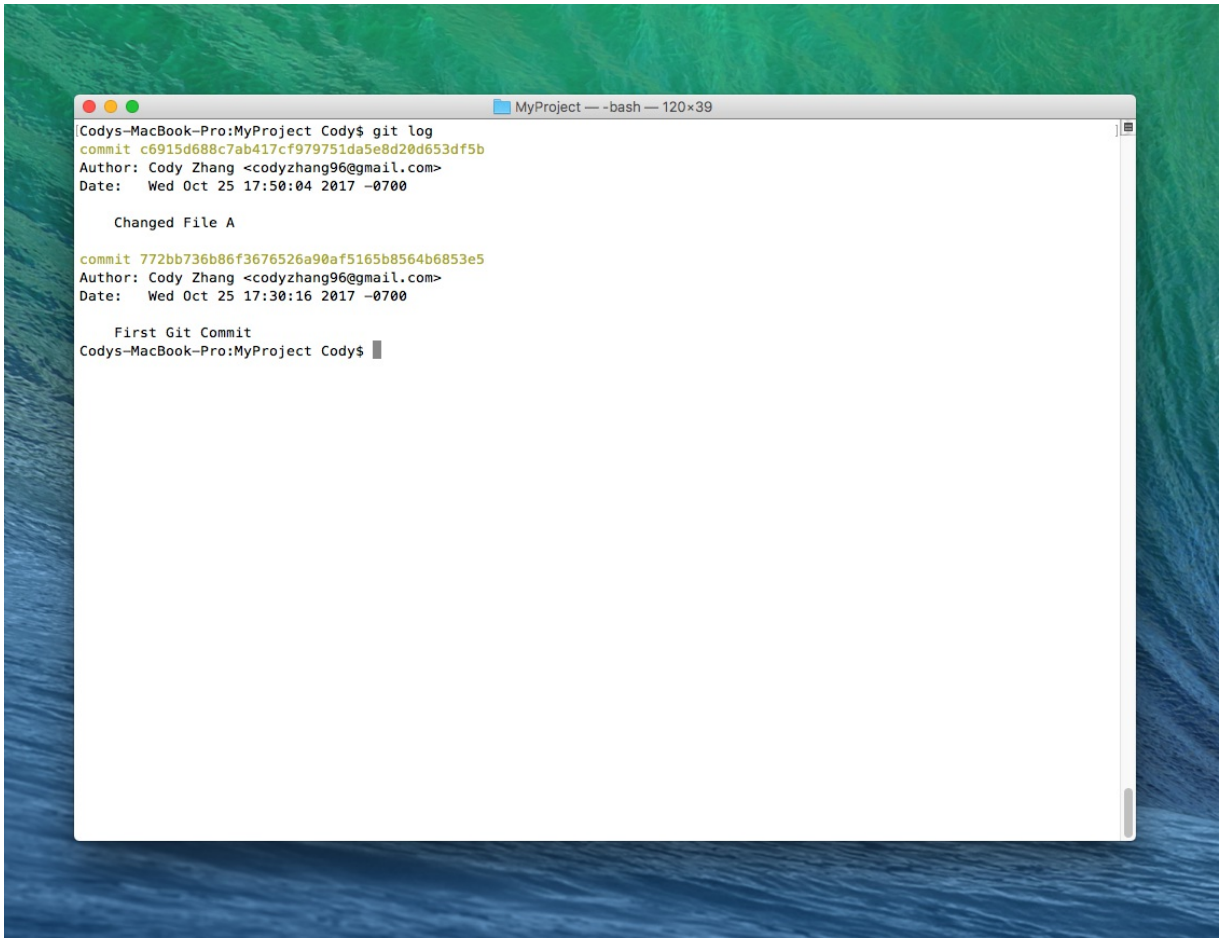
Now let's modify our files a bit and make a new commit of our modifications!

Open up file_a.txt with any text editor you want and type in whatever you want.

Run the following commands in your terminal.

```
#Check the stage status after modifying your file.
git status

#Add the changed file to the stage to be committed. Note how you don't have to add file_b since nothing was change
d there.
git add file_a.txt

#Commit the change to your git repository
git commit -m "Changed File A"
```

You can run the command `git log` to see all your recent commits. It will show the commit id, message, date, and author information.



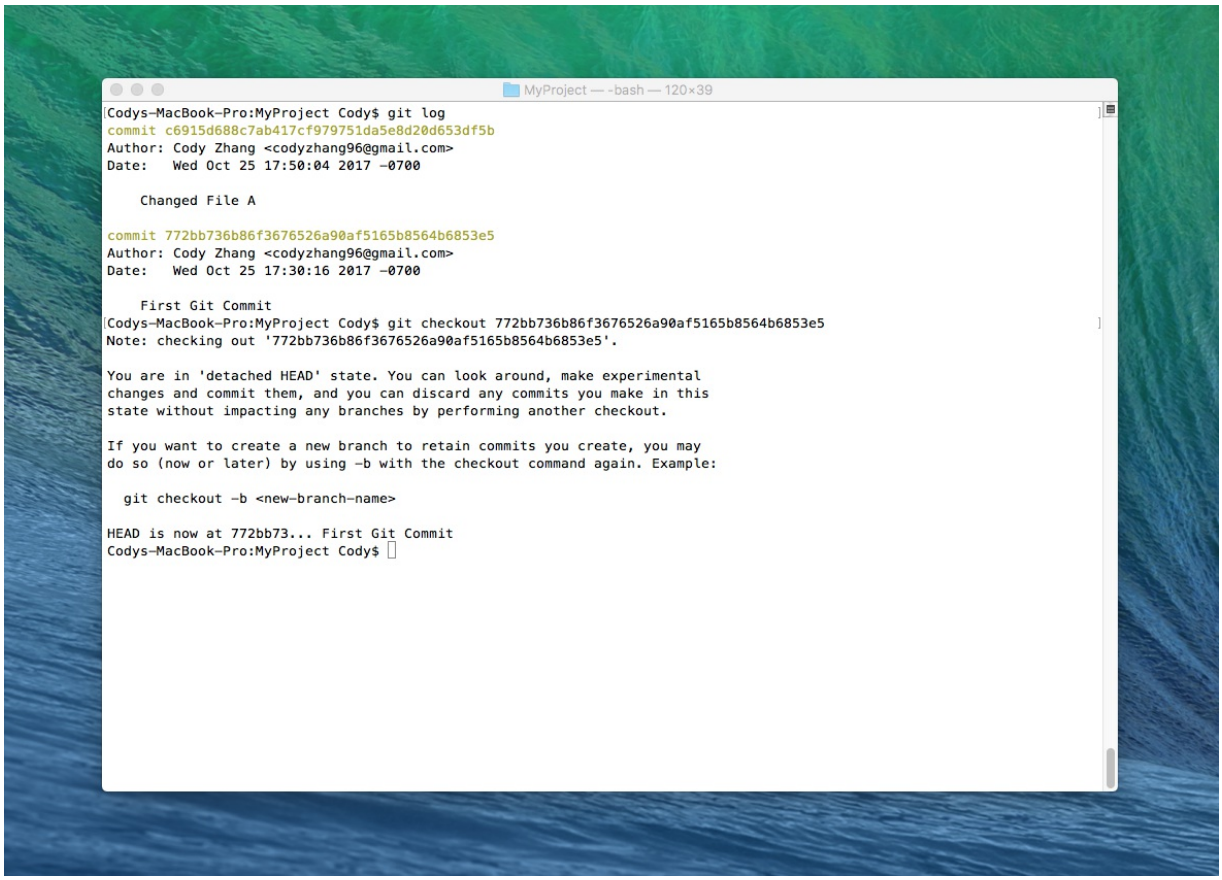
Git Log

Git Checkout and Branches

Every time you commit something to your git repository, git automatically moves your branch's pointer to the latest commit. But if you wanted to work on an old commit for whatever reason, you can use the checkout command.

However, there are some complications that can come from using the checkout command.

One way to use checkout is to just checkout an old commit by its commit id directly.



Checking out old commit

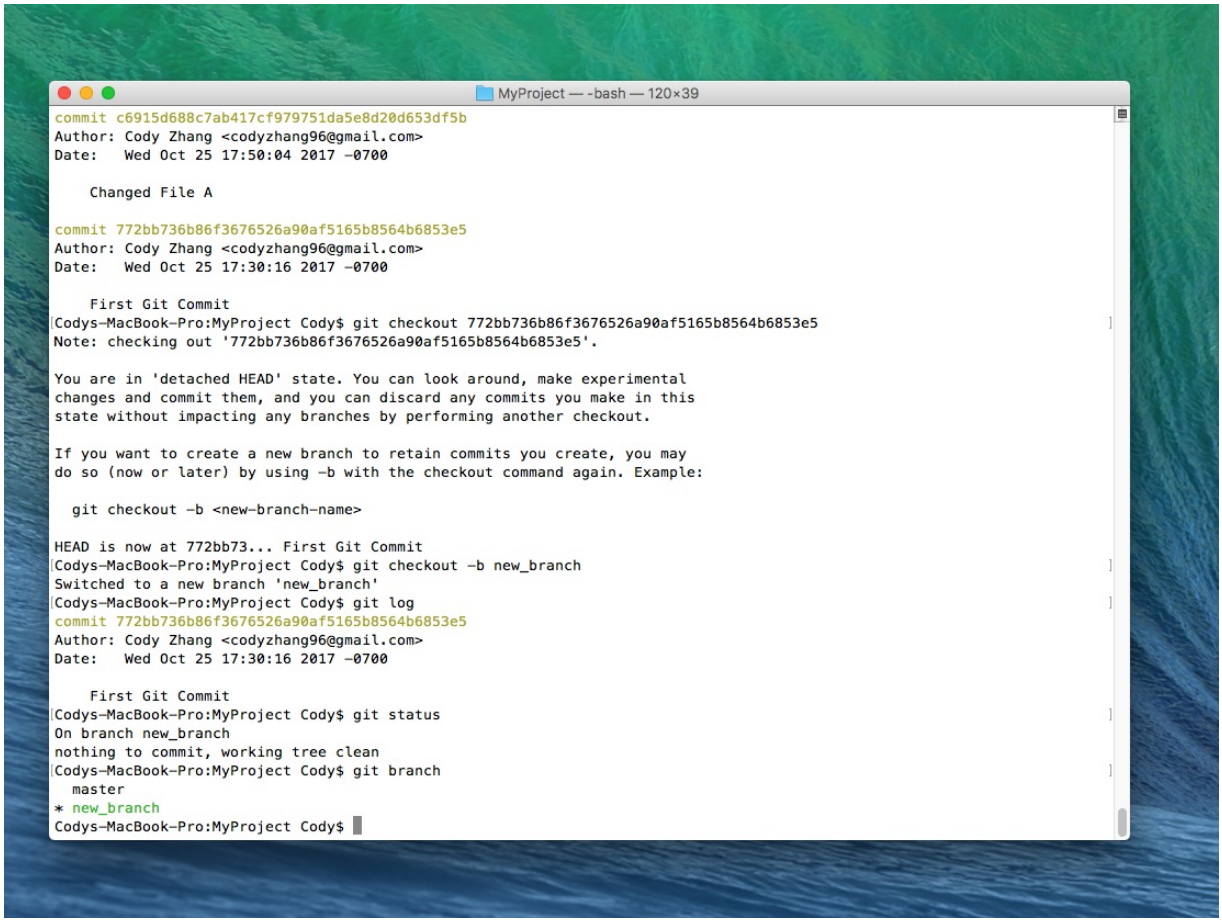
However, this is usually not suggested because you will lose your branch pointer by doing this. Git will give you a warning that you are in a 'detached HEAD' state. What this essentially means is that although you can make changes to the state of this commit, you will not be able to get back these changes if you ever move to another state unless you remember its commit id exactly. But, remembering a gibberish commit id made of lots of numbers and letters is not realistic. This is where branches come in very useful.

Recall, that a branch is a pointer to a commit, it points to a specific state of your projec. Multiple branches are helpful because they allow you to easily keep track of different stages of your project and refer back to them. Branches can take whatever name you want, but git always creates the first branch for you, the *master* branch. The master branch is typically a state of your project that you showcase, so testing new ideas usually occurs on other branches. **Branches always point to a single commit and reattach themselves to the newest commits automatically**. Whenever you create a new branch, it maintains all the commits of the branch where you called the create. For example, if you created a new branch from your master branch, the new branch will still have all the commit history of the master branch even though it is its own pointer now, it is 'branching off'.

To resolve the deatched HEAD state, you can create a new branch which will add a pointer to this current state so that in the future you can just checkout this branch to return to this state.

Lets make a branch to help fix our detached HEAD state! Run this in your terminal.

```
#This creates a new branch to point at the current commit and automatically switches you onto this branch.
git checkout -b new_branch

#Git status will return a clean work tree. Why? There was nothing new that was added to our project files, we only made a new pointer to this commit so that we can easily reference it later.
git status

#We should now have 2 branches in our git repository. The master branch which was created upon initiation and the new_branch we just created. Check by using the following command. The branch you are currently on will be highligh ted.
git branch

#Try running git log on this new branch, what is different from when we ran git log before? Depending on the branc h you are on, git log will only show you commits that have happened on this branch. Note upon a branch creation, g it will add all previous commits that happened from the commit you created the new branch from into the logs as we ll.
git log
```



Fixing the detached HEAD state

Branches are extremely useful because they allow you to easily move to different stages of your project. Especially if you are working on a large project with a lot of edits over time, having branches will make your life a lot easier when you wanted to build on what you did a week ago, a month ago, or even a year ago in different ways. Imagine if you were adding new things to your project for an entire month, but horribly, you realize everything you added was wrong. Now you made a lot of commits since a month ago so going through the logs is very very painful. Also, you didn't add very helpful commit messages so you can't really find exactly which commit you want to go back to for starting over. Wouldn't it be nice if you had a branch that pointed to the exact state of the project from a month ago? All you have to do to start working on that state is to call `git checkout branch_name`.

Branches are also extremely useful for collaboration. Imagine if you had an idea you wanted to try out for a group project but your teammate had another idea. If you were both working on the same branch, your commits are very likely going to mixed together and result in files that do not

make sense, since you are implementing very different things. Furthermore, it will be very hard for you to go back to a clean state of your project if your team member keeps committing changes that don't make sense to you or simply just don't work. Instead, you and your partner can each create separate branches to work on. This way, you can keep track of your own changes to the project and test your idea while your partner is able to freely implement his idea without conflicts.

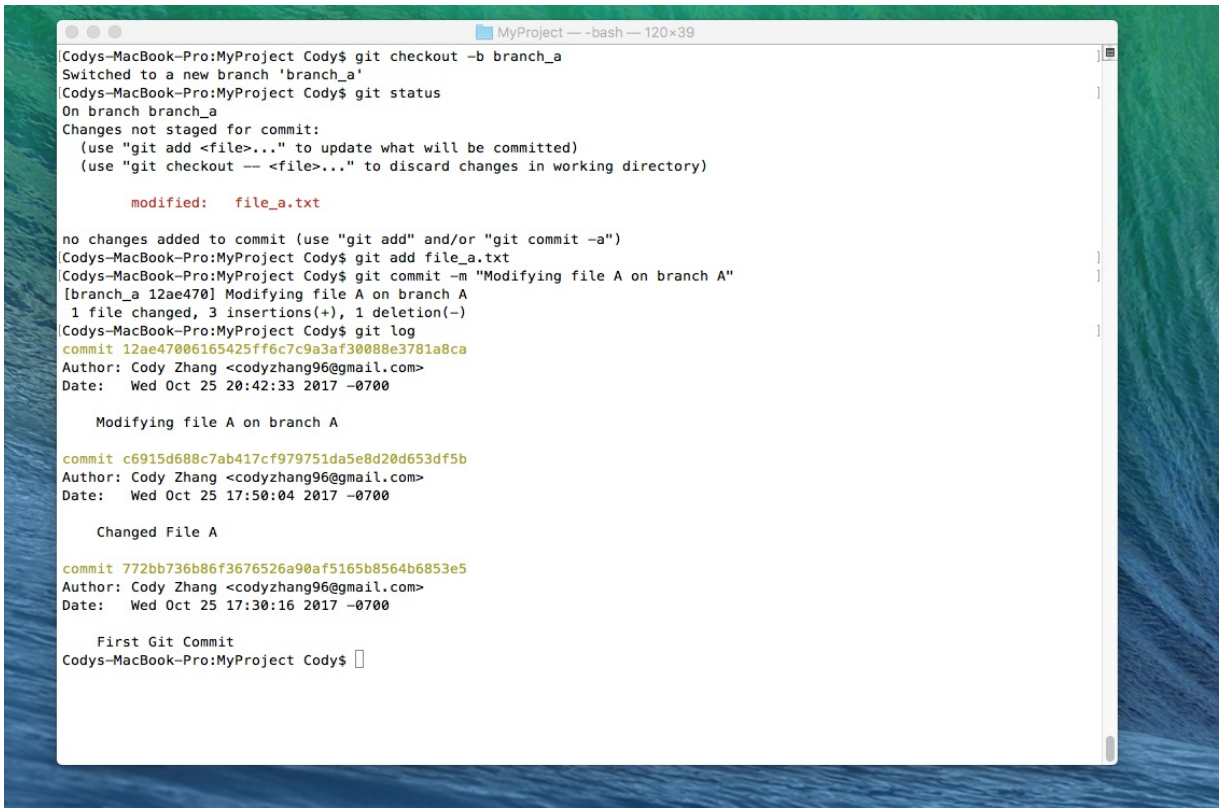
(It doesn't actually make a lot of sense that you and your teammate are both working on the same local repository since it is local to your computer so keep the above discussion of branching for collaboration in mind, but it is usually applied to a context where remote repositories are involved, explained later in this post.)

Summary of when to make new branches:

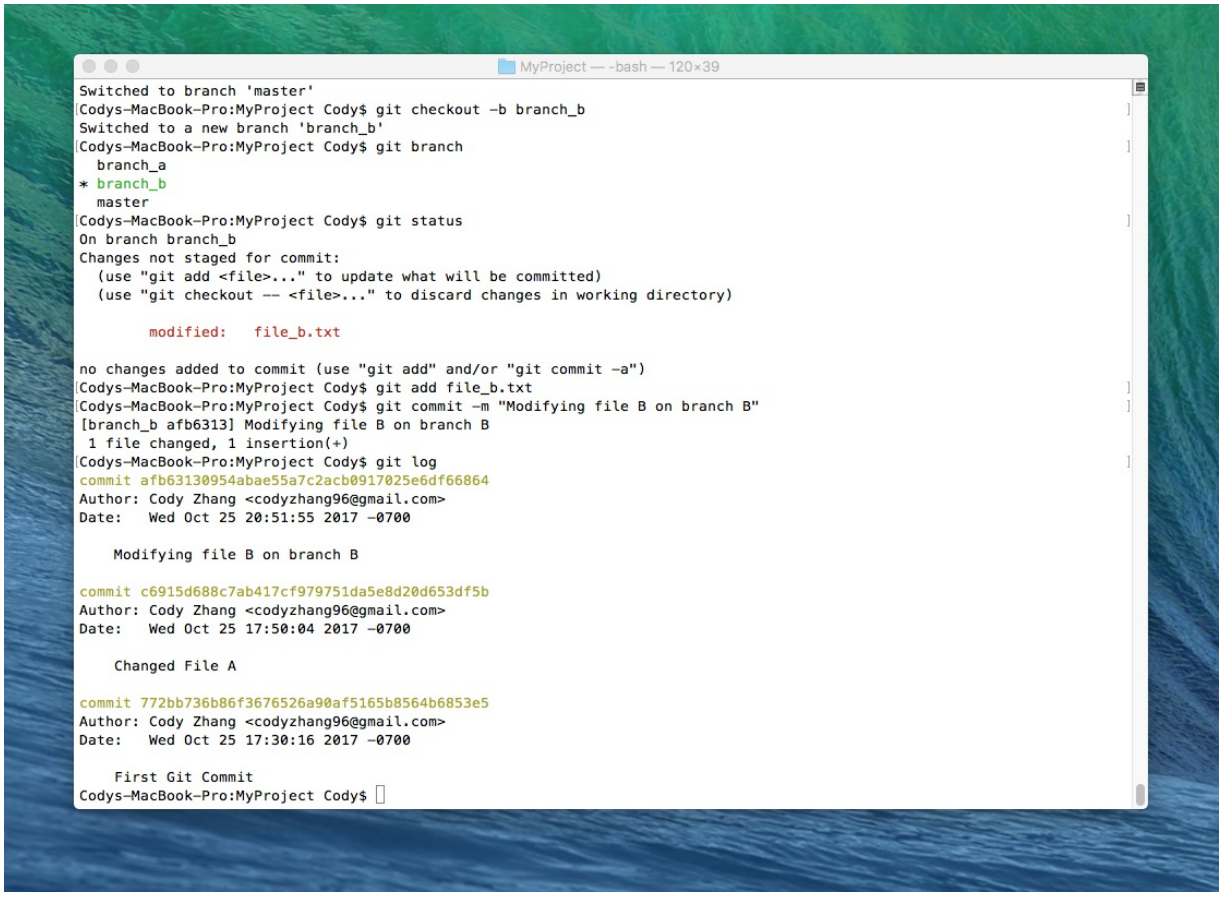
1. Whenever you want to try out a new idea from the current state of the project.
2. Whenever you think this is a state of the project that you will return to in the future, regardless of what you are going to add to the project until that point in the future.
3. Whenever you and a teammate want to try out different approaches, it is a good idea for each of you to have a separate branch to work on.

Let's see an example of working with branches. Run these commands in your terminal.

```
#Create a branch where you will only work on file a
git checkout -b branch_a

#Modify file_a.txt with any text editor you want and then add and commit your changes to branch a
git add file_a.txt
git commit -m "Modifying File A on Branch A"

#Check your logs for branch A
git log

#Go back to our master branch
git checkout master

#Create a branch where you will only work on file b
git checkout -b branch_b

#See how many branches you have
git branch

#Modify file_b.txt with any text editor and then add and commit the changes to branch b
git add file_b.txt
git commit -m "Modifying File B on Branch B"

#Check your logs for branch B
git log
```



Working with Branch A

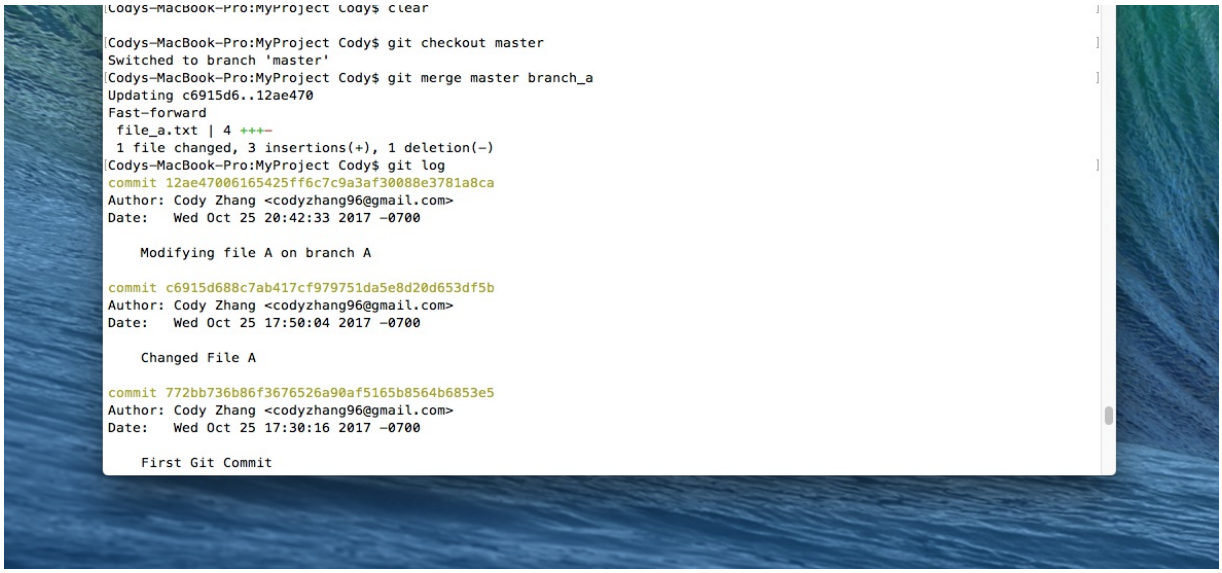


Working with Branch B

If you check your File A while on Branch B, you should see that the changes you made to File A on Branch A are not found. Similarly, if you check your File B while on Branch A, you should see that the changes you made to File B on Branch B are not found. Now if you check your files on the master branch, neither file changes you made can be seen. As you can see, each branch will start keeping track of its own unique changes.

How do you put all your changes together into one branch? Git allows you to do this really easily with the merge command. Let's try to merge our changes to file a and file b on their respecitve branches into the master branch.

```
#Checkout your master branch.
git checkout master

#Merge in the changes of branch A into master.
git merge master branch_a

#Check your git logs. What was added? You should see that your commit on branch A is now logged for master too aft
er merging.
git log
```



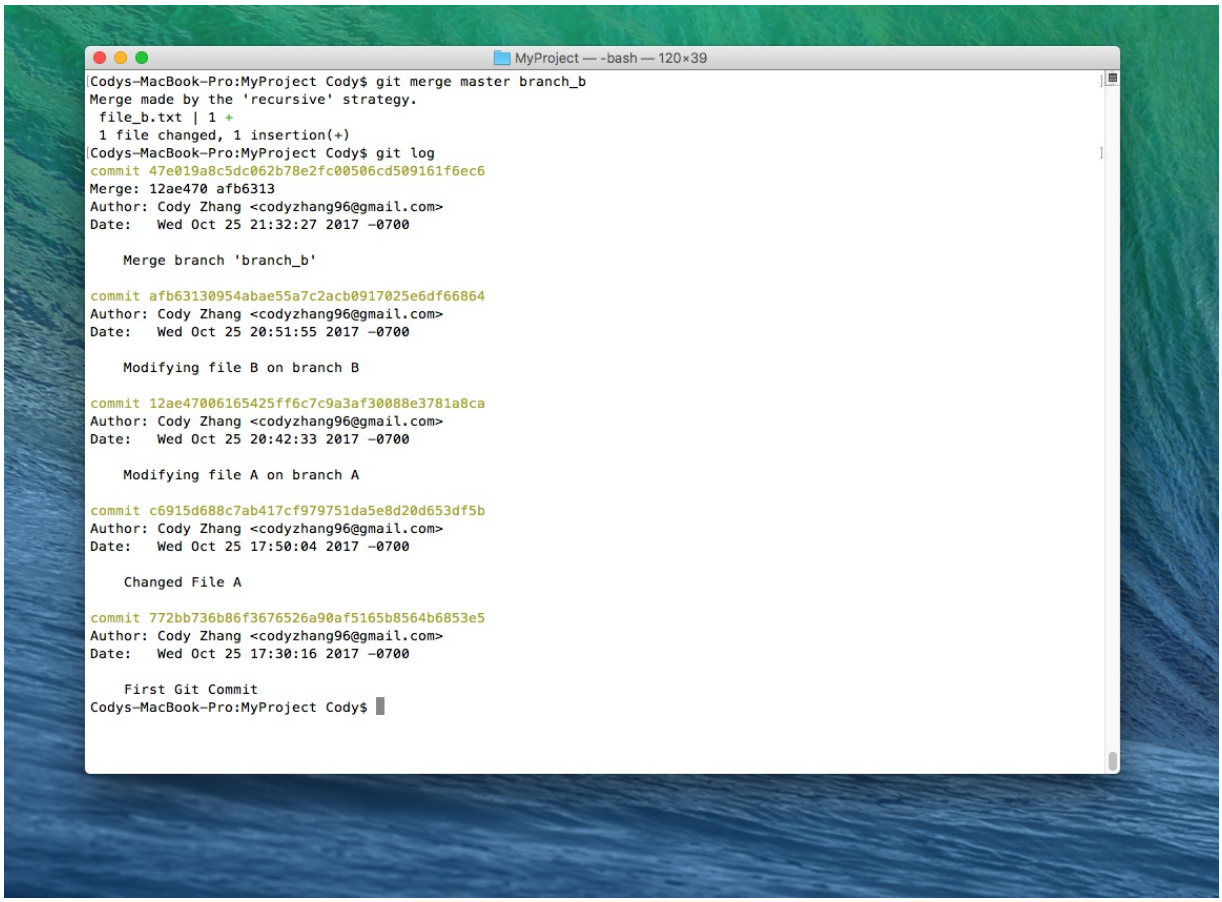
Merging in Branch A to Master

After merging branch a into master, we proceed to merge `branch_b` into `master`. However, something interesting will be noted here. Git will prompt you to write a commit message to explain why the merge is necessary. Merging `branch_a` did not prompt this message because that was a *fast-forward-merge*. A fast fowrard merge means the branches did not 'diverge'. `branch_a`'s state can be reached from `master`'s initial state with only additions. However, after merging in `branch_a`, `branch_b`'s File A state can no longer be reached from the master branch with only additions, and thus an additional merge message is needed when we try to merge `branch_b` into `master`.

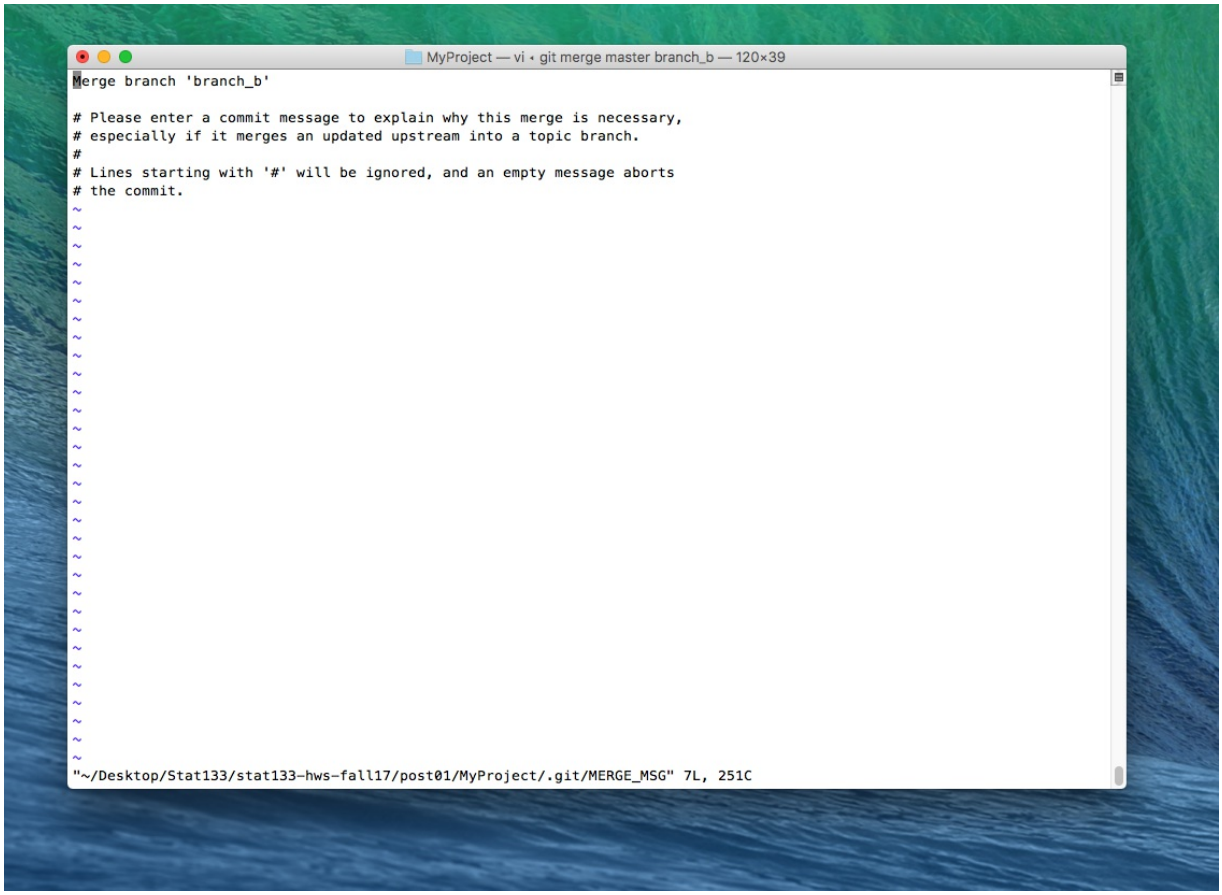
Git will prompt vim, a command line editor for you to write the message in. You can read about vim more online to see how to use it or just type `:q` to exit out of the vim editor and let git put in a default merge message.

Run the following commands to see for yourself.

```
#Go back to master branch
git checkout master

#Merge in Branch B. This will prompt the vim editor for a merge message, just type `:q` for now.
git merge master branch_b

#Check the logs after merging complete.
git log
```



Merging in Branch B to Master



Vim editor prompted by git when merging in Branch B to Master

After finishing our merges, our master branch should have both the changes we made in `branch_a` and `branch_b`. Check File A and File B to see for youself!

Now, there are times when both you and your teammate modified the same things, in which case, git will tell you there are merge conflicts which you have to manually go into the files and fix when you try to merge the branches. Basically, git will put both of your work into the files and git will helpfully add tags of which commit each copy is from. You can manually erase the one you don't want or pick the best things from both to put together.

You can delete a branch with `git branch -d <branch_name>`, but you can't run this command if you are on the branch that you want to delete, run it from another branch.

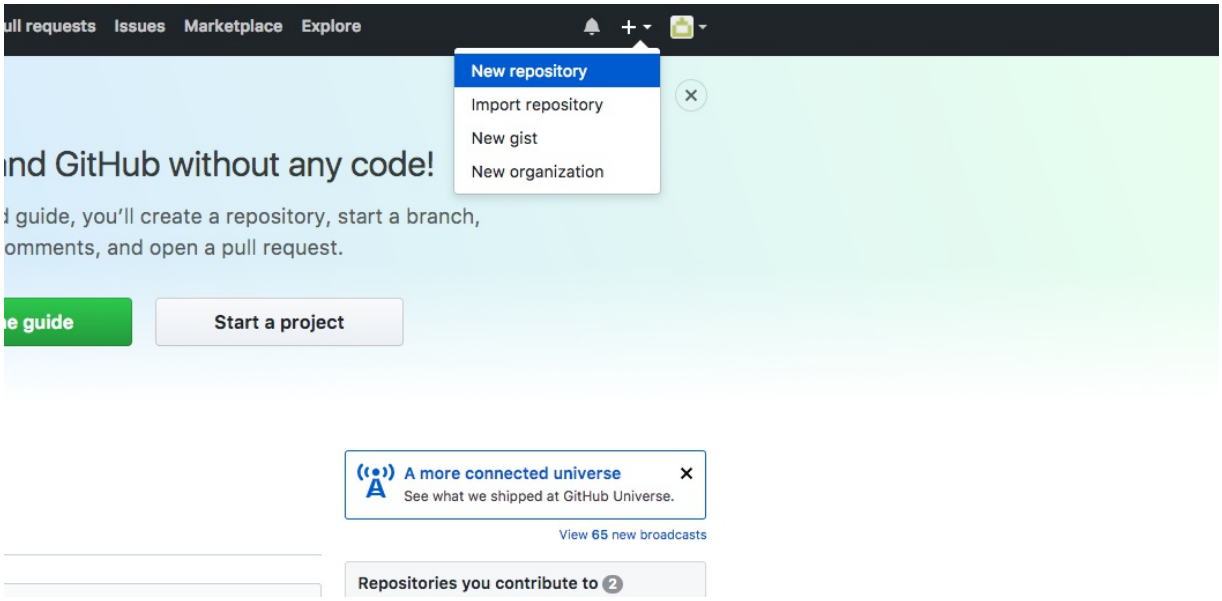
Using Git with GitHub

Now that you know the basics of how to use Git. Let's learn how to start using GitHub. As mentioned before, GitHub is a platform for sharing your work and collaborating with others. Everything that you learned about using Git locally still applies now we introduce the notion of remote repositories. Remote repositories are exactly like your local git repository, they still keep track of changes to your projects, but can be accessed by many people at once since it is online.

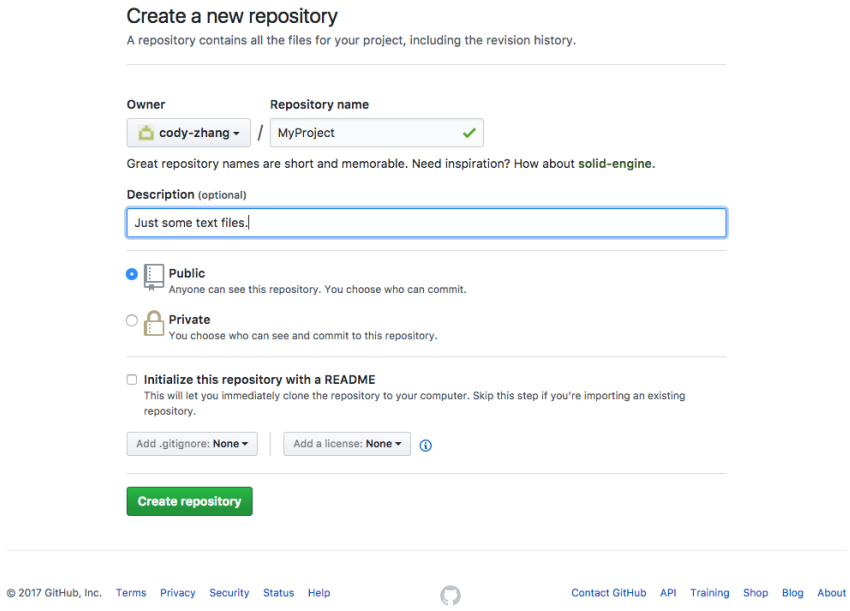
Creating a Remote Repository

First, make sure you have a GitHub account created. Register at https://github.com/

Log in to your account on GitHub then click new pository to create a new GitHub Repository for your project. Give your repository a name and an optional description. You have the option to make a private or public repository. A public one is open to the world to see while a private one is only seen by people who you give permission to.

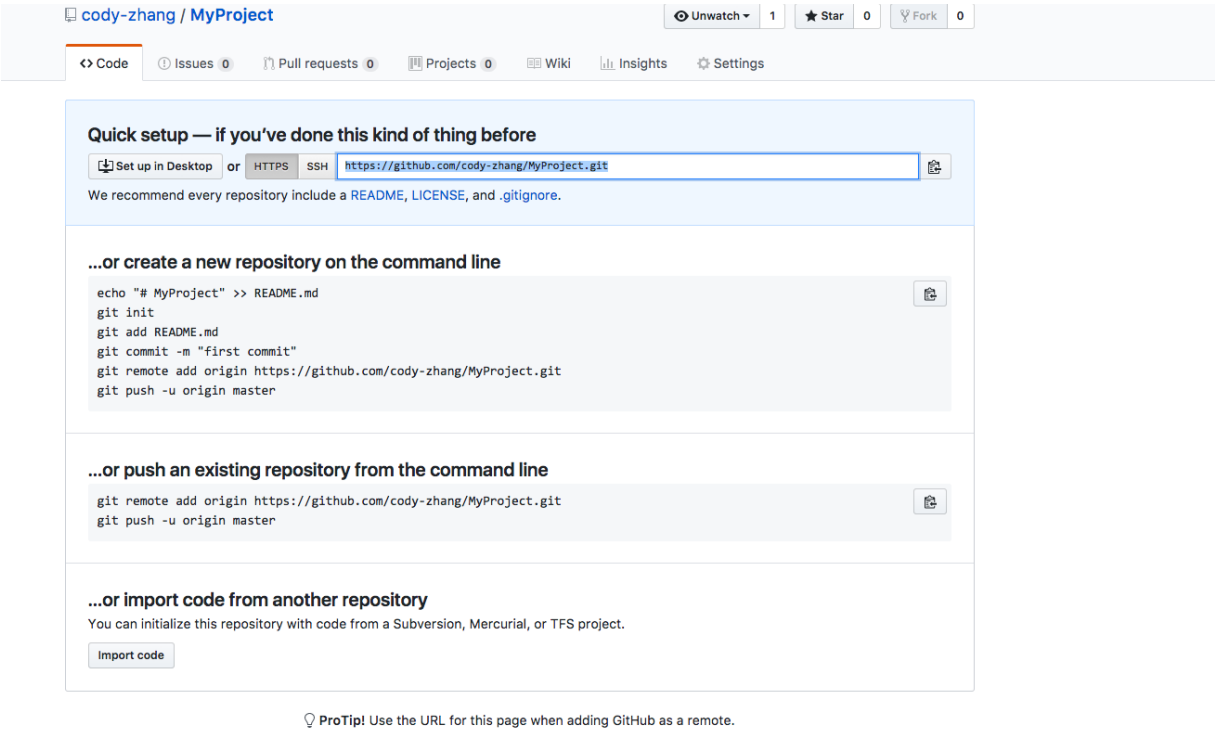


Making A Remote Repository1



Making A Remote Repository2

After creating the repository, you will see a url available for you. This is your Git repository url which you will use to sync up your local work with GitHub. Every remote repository on GitHub has its url.



GitHub Repository URL

To do this run the following commands in your local terminal from your project directory:

```
#Add the remote repository to your local workspace. The remote repository will be called 'origin' in your local re
pository.
git remote add origin https://github.com/cody-zhang/MyProject.git

#Check what remote repositories you have now.
git remote -v
```

You can add many remote repositories to your local git repository, but make sure you give them separate names. A use for having many remote repositories is that you might sometimes want to use some other people's projects for whatever reason, having their remote repostories linked to your local allows you to pull their work into your own local repository. The `origin` remote repository is a replica of your work for others to do the same with!

Just like how you can have branches in your local git repository, your remote repository can have branches as well. Your remote git repository replicates your local branches, there is a one on one mapping. The remote branches do not get generated until you push to your remote repository from a local branch however.

Pushing your work to GitHub

After having a remote repository linked to your local one, you can start sharing your work on GitHub! A simple workflow goes in the following way:

1. Pull from the remote repository. This will merge the most recent changes to the project someone has made and shared on the specified branch of your remote repository.
2. Make changes on your local repository.
3. Push your changes to the remote repository so your changes can be seen by your other collaborators.

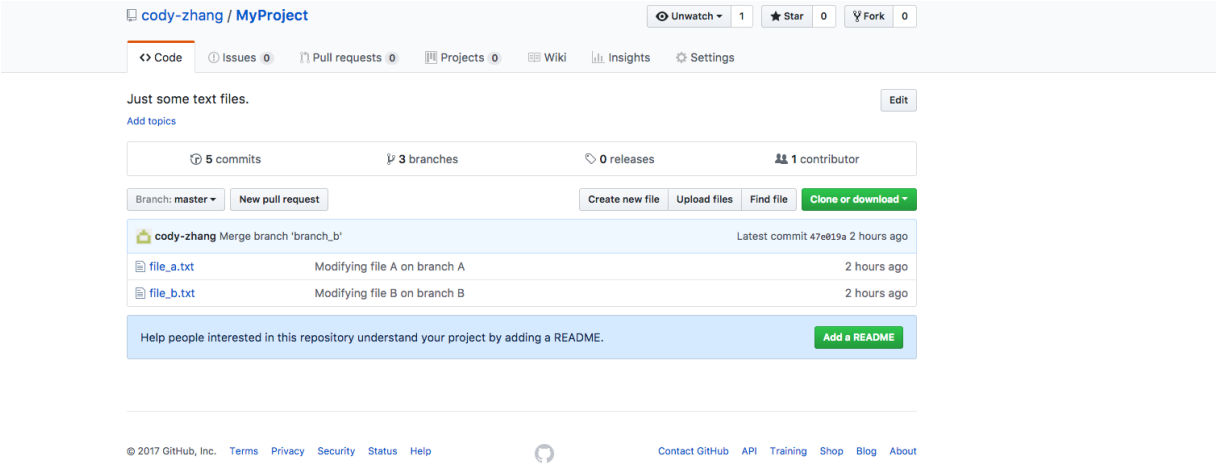
Let's Try it Out. The commands have the following syntax: `git <pull/push> <remote_name> <remote_branch_name>`

```
#Try pulling from your new remote repository. Since this is a brand new remote repository, there aren't any branch
es initiated yet on your remote, so you will get a ref error from this command.
git pull origin master

#Try pushing your previous project changes to your remote. This will push all the data from your local repository
about this branch including logs, commit messages, and etc. into your remote branch.
git push origin master

#Now try pulling again. It should say everything is up to date because your remote now holds the exact same thing
as your local.
git pull origin master

#Now lets initialize the remote mappings of our other local git branches.
git checkout branch_a
git push origin branch_a
git checkout branch_b
git push origin branch_b
```

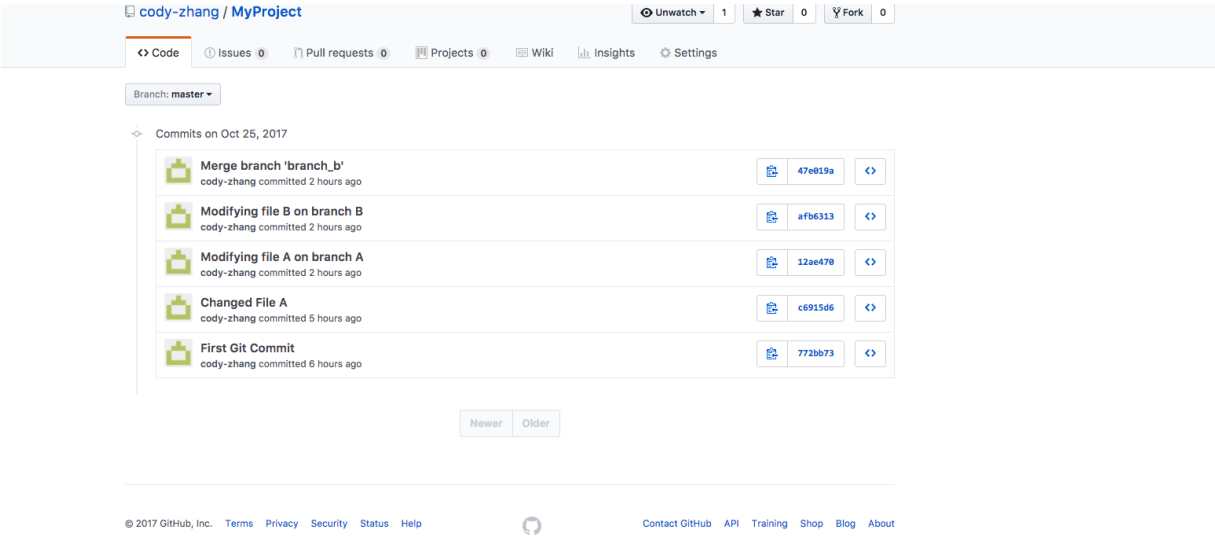
```
MyProject — -bash — 120×39
Codys-MacBook-Pro:MyProject Cody$ git remote add origin https://github.com/cody-zhang/MyProject.git
Codys-MacBook-Pro:MyProject Cody$ git remote -v
origin  https://github.com/cody-zhang/MyProject.git (fetch)
origin  https://github.com/cody-zhang/MyProject.git (push)
Codys-MacBook-Pro:MyProject Cody$ git pull origin master
fatal: Couldn't find remote ref master
Codys-MacBook-Pro:MyProject Cody$ git push origin master
Counting objects: 14, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (12/12), done.
Writing objects: 100% (14/14), 1.34 KiB | 0 bytes/s, done.
Total 14 (delta 1), reused 0 (delta 0)
remote: Resolving deltas: 100% (1/1), done.
To https://github.com/cody-zhang/MyProject.git
 * [new branch]      master -> master
Codys-MacBook-Pro:MyProject Cody$ git pull origin master
From https://github.com/cody-zhang/MyProject
 * branch            master     -> FETCH_HEAD
Already up-to-date.
Codys-MacBook-Pro:MyProject Cody$ git pull origin branch_a
fatal: Couldn't find remote ref branch_a
Codys-MacBook-Pro:MyProject Cody$ git checkout branch_a
Switched to branch 'branch_a'
Codys-MacBook-Pro:MyProject Cody$ git push origin branch_a
Total 0 (delta 0), reused 0 (delta 0)
To https://github.com/cody-zhang/MyProject.git
 * [new branch]      branch_a -> branch_a
Codys-MacBook-Pro:MyProject Cody$ git checkout branch_b
Switched to branch 'branch_b'
Codys-MacBook-Pro:MyProject Cody$ git push origin branch_b
Total 0 (delta 0), reused 0 (delta 0)
To https://github.com/cody-zhang/MyProject.git
 * [new branch]      branch_b -> branch_b
Codys-MacBook-Pro:MyProject Cody$ g
```

Pushing local chagnes to the Remote

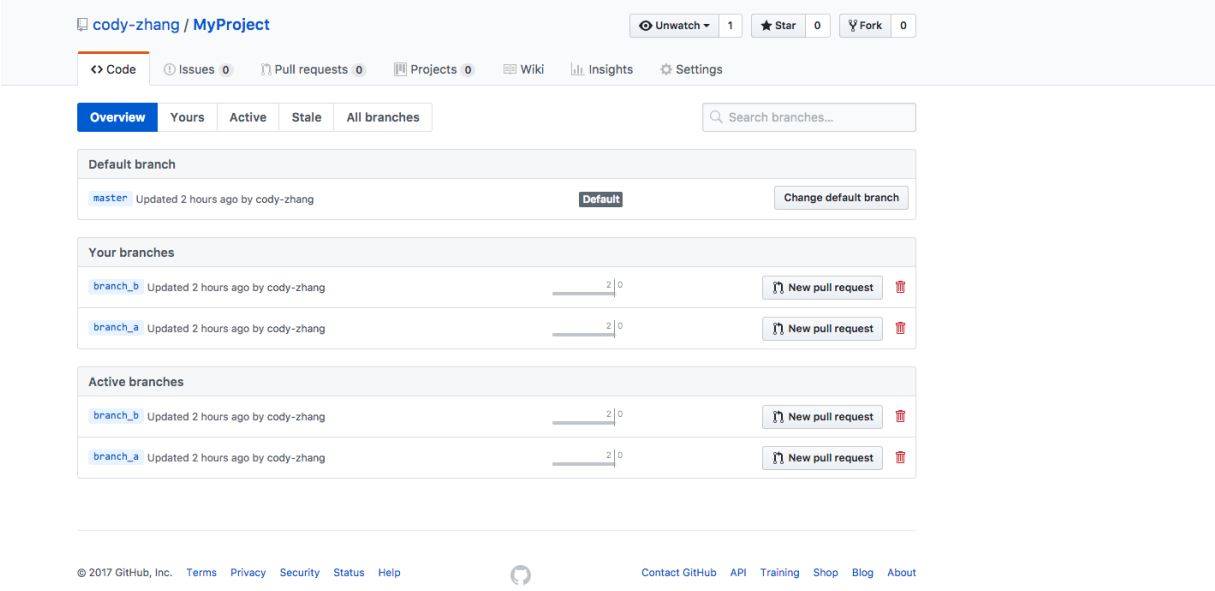
You can now see all your local work on your github account if you goto the remote url. You can also see things like commit histories, different branches, the changes made for each commit and other useful stuff. Explore for yourself on your GitHub!



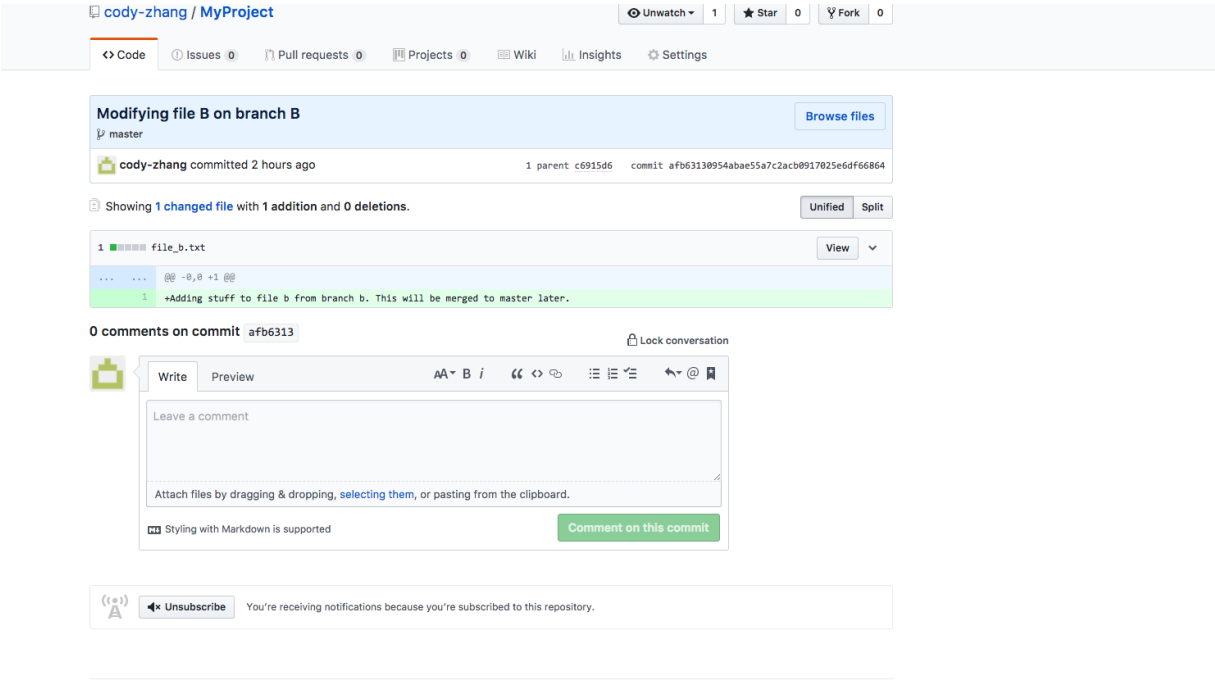
Your GitHub Repository After Pushing Local Changes



The Commit Histories Seen On GitHub



The Branches Seen on GitHub

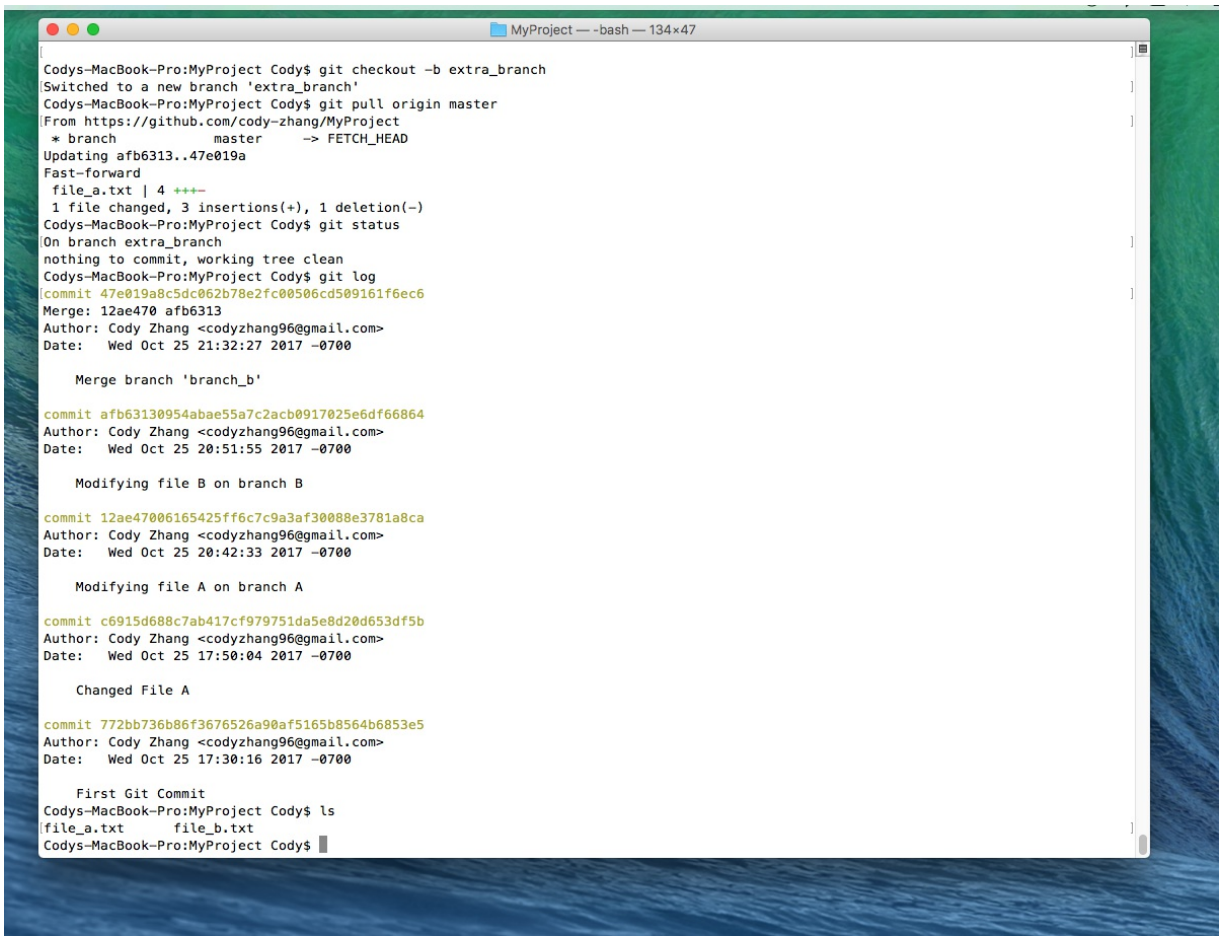


A Commit's Details Shown on GitHub

Pulling Work From Remote Branch

We can now create a new branch locally and pull from the remote master branch to replicate all the project files into our new local branch.

```
#Creat the new branch
git checkout -b extra_branch

#Pull from the remote master branch
git pull origin master
```



Creating new branch and pulling from origin master

If you now check the git logs of the new branch or the project files, we were able to successfully merge in file_a and file_b from the remote master branch into our new local branch. Pulling from remote branches is essentially the same as merging two local branches. There are merge conflicts that you must resolve manually sometimes. This new branch will not exist on GitHub until you push it to the your remote repository.

Last but not least, another very useful command is `git clone <remote url>`. This will clone a remote repository's files into your local repository and automatically set up a the remote as your origin. This is very useful if you just got brought onto someone else's project as a new team member and need to start building on what they have already locally.

References

Hopefully now you have an idea of how to use Git and GitHub. This post only goes over the basics however, and there are a lot of resources for you to learn more about these tools. Here are some very helpful ones:

A Git Command Cheatsheet

A Very Very Good Walkthrough of Working With Git and GitHub

A Good Stackoverflow Answer Clarifying Fast-Forward Merges

A Good Website Clarifying A Detached Head State

A Very Detailed Complete Documentation on Git

A GitHub Tutorial For Beginners

A Good Forum Explaining Shell, Terminal, And Console

Take Home

Computers are inseparable from our work in this day and age. Regardless of what you are working on, Git and GitHub offers everyone great utilities. These tools provide great benefits to collaborating on projects, and gives you the flexibility to explore new ideas without losing track of your edits. Git may be daunting at first, but with time, it will become a tool that will make your life much easier and more efficient when working on projects.