Git: A Version Control System

By Adish Jain

Introduction

Throughout our class, we have been using tools such as git and Github for assignment submission, yet have only understood these systems on a very surface level. Thus, the purpose of this post is to help readers gain a better familiarity with these tools, and the full extent of their powerful applications. The first half of this post will give you an in-depth reading on what Git is and how it works. The reproducible portion of this post can be found in the “A Summarizing Example” section (okay’d by Professor Sanchez).

A Brief History

Before we can actually begin to dive into the different functions and usages of a tool like git, it would serve us well to take a quick look at its brief history.

As surprising as it may sound, given its ubiquitousness in today’s development community, git is actually quite a recent phenomenon, created only in 2005 by Linus Torvalds (the creator of Linux). Torvalds began development of this new version control system when he realized that none of the free and existing distributed control management systems would be sufficient in scaling his Linux kernel development. Because the development of the Linux kernel was completely collaborative and open-sourced, having a strong *distributed* control management system was instrumental to the huge project’s success.

Linus eventually decided to name this new version control system he had developed as “git,” a word which, just 12 years later, no software developer can get by without knowing.

What is Version Control?

In the previous section, we dicussed a little bit about the history of git in the context of version control systems (VCS), but what exactly does that mean? What is a VCS? Well, a VCS is essentially a program that you can use to keep track of changes made to other programs, files, or even entire directories. You can think of it as a sort of system which basically keeps track of changes you make to a file, where that file could be of any extension type (.txt, .py, .R, etc). Any changes you make to the file will be tracked if that file is *added* to the system and each change will essentially result in a new *commit* of that particular file. The system then allows you to revert back to any previous version at any time, and gives users a way to save their work frequently and easily.

While this might sound overly complicated at first, the idea of version control is very intuitive and something that we all do instinctively. For example, consider you’re playing a video game and that you’re very close to the winning level, but your parents want the TV so you have to stop playing. Well, you’re not just going to leave and lose all the hard work you put in getting so far. Instead, you’re going to save your progress so that you can pick up where you left off when your parents are done watching Game of Thrones.

If you can’t relate to that, consider the less fun scenario in which you are writing an essay. While writing an essay, a number of things could happen to your work: the computer could freeze, the text editor you are using could accidentally close, or you might accidentally delete all your work. So, to avoid having to start all over again if any of those things do happen, you are instinctively going to save your work periodically so that there’s always a backup, or a *version*, that you can resort to, should something go wrong.

This is all version-control is. A tool like git then allows one to take this idea and streamline it, so that it is a more automatic and secure process, and is scalable to hundreds and thousands of collaborators instead of just you on your own.

Hopefully this gives you a basic, high-level understanding of the function of a tool like git. And now that we know a little bit about the motivation behind and origin of git, we are ready to look at some of the more low-level functionality of the tool and the back-end processes that make it work.

How Git Works

There’s a lot going in behind the scenes with git, and one shouldn’t expect to be able to explain nor understand it in one sitting alone. Here, however, I’ll try my best to give you a quick look at the inner-workings of this specific VCS.

Local Areas of Operation

First, let’s understand the framework within which Git functions, limiting our scope to only local operations (don’t worry about remote repositories for now). On our local machine, we have three different “areas” which Git links together.

First, we have our working directory, which basically represents the directory the file with which we are working is currently in. So, for instance, if you were editing a word document you had saved under your ~/desktop directory, then that’s what your working directory would be.

Second, we have a place we can refer to as the “Staging Area”. This area serves as an intermediary between our working directory and our actual git directory (repository). When we *add* a file to our system to be tracked (saved), the file does not go directly into our repository. First it is stored in this intermediary staging area, and it only actually enters the repository after we decide to commit our work. Don’t worry if this doesn’t make complete sense yet; I promise it will come together in just a bit.

Finally, we have our actual git directory, or our repository. Briefly, a repository is essentially a directory in which all the files we actually want to save our stored. The repository contains things called commit objects (which are data structures that hold our file contents) and some other metadata about those commits.

In short, git functions across all three of these areas, and these areas interact with each other via different commands such as “add” (working directory to staging area), “commit” (staging area to repository), and “checkout” (repository to working directory). If this is not yet clear, hopefully the following picture will clarify:

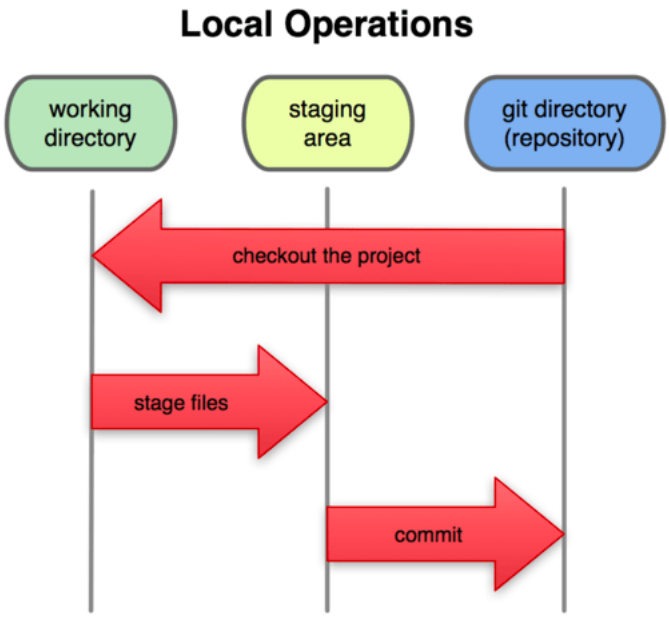


Illustration of the main three states your Git versioned file's lifecycle

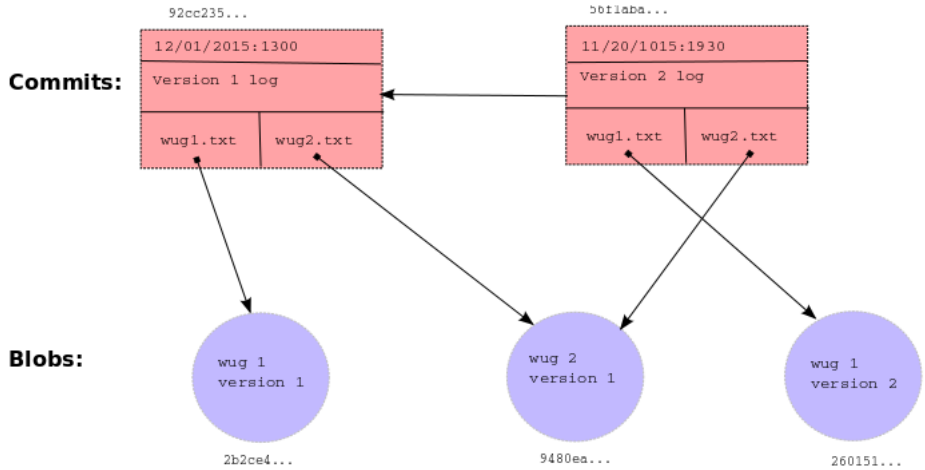
Objects in Git

Git, like most programs, is best understood in the context of an object-oriented paradigm, meaning that each piece of code represents a certain object, and the program as a whole describes how those objects interact with each other.

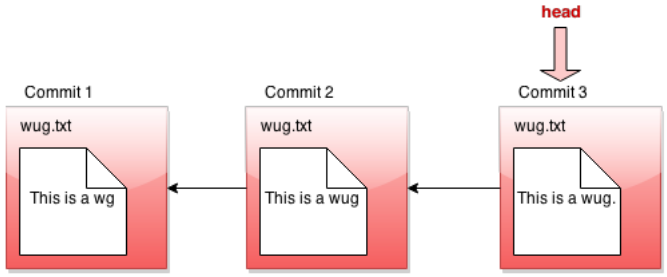
In Git, there are a couple basic objects we deal with: blobs, commits, branches, and trees.

A blob is basically a file that we want to start tracking in our repository (a file that we want to save a version of). All blobs contain are the name of the file we are working with, and the contents of the file. When a file is *added*, it is taken out of our current working directory and put into our staging area, as described above. When it is put into our staging area, it becomes a blob, and is marked by a unique identifier to keep track of it.

Commits are a little more complicated. A commit is basically an object that is created everytime we choose to move all the blobs from our staging area to our actual git repository, via the command commit (notice that the command's name is "commit," not to be confused by the actual object created, which is also called a "commit"). Each commit contains several crucial pieces of information: some metadata regarding when the commit was made, who the commit was authored by, a commit message written by the author, and a unique identifier which helps keep track of the commit. Additionally, commits contain a couple of references, which, in Computer Science, are basically addresses to other objects. Commits contain pointers to their "parent commit" (more about this when we discuss tree objects), and they contain references to all the blobs which they are currently tracking. The following diagram illustrates the relationship between blobs and commit objects:



Okay, so what are these branch things then? Well, our commits keep track of all our blobs (our files), but how do we keep track of our commits themselves? Well, keeping track of our latest commit is easily done. Whenever we create a new commit, we point a new branch object called "HEAD" to that most recent commit. This way, we know and can easily recover the most *recent* version of our files (our blobs). We would simply follow the branch object's pointer to the commit object, and then follow the commit object's references to all the most recent versions of our blobs.

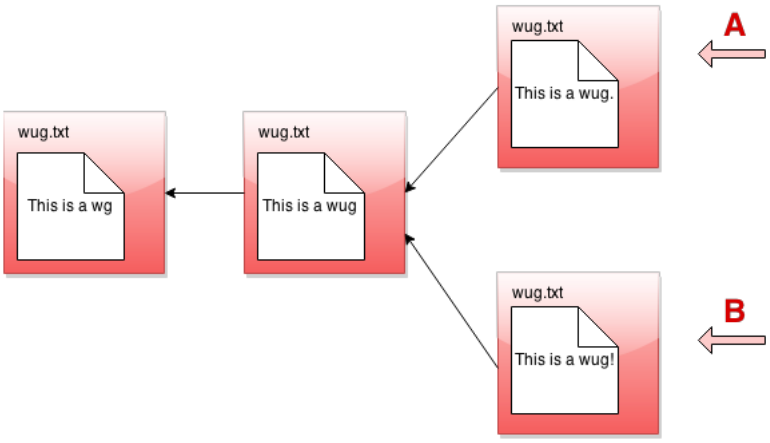


But wait, the whole point of version control systems is to be able to go back in time and recover *any* version of our files, not just the most recent one. Well, this problem is already solved for! Recall that one of the metadata that each commit object contains is a reference to its parent commit, meaning that if we our HEAD branch points to our latest commit, we can follow the latest commit's reference to its parent commit to go back to previous commit objects (and thus previous versions of our blobs). Hopefully this is starting to make some sense.

Okay so we are starting to see some sort of structure emerge, where each commit sort of represents a *node*, and each commit contains multiple references to different blob objects. And all the commits are linked together by references to their parent commits, and, of course, we have a reference to the head commit via our HEAD branch object. Putting all these different objects together, we start to see a tree-like structure form, more formally known as the commit-tree.

In both the previous diagrams, our tree structure had been flat and looked very little like a tree. This is because we had simply been going from one version of our file to another, and only had one branch called the HEAD pointing to the most recent node. But the true structural value of a commit-tree is really only seen when we explore the idea of going from one file to two separate versions of that file in the same time-step, meaning that we would no longer have a single "most recent version" of that file, but would rather have two! In this case, we would require another branch object to keep track of the other "most recent" commit node, which is where the idea of a tree structure really begins to become clear.

For instance, say we had a text file which contains the sentence "This is a wug", and we want to create two separate versions of that file, one with a period ("This is a wug.") and one with an exclamation mark ("This is a wug!"). It wouldn't make sense for us to put any one of those versions after another; rather they should be on the same "level". And we would need require another branch (other than our HEAD branch) to keep track of both these nodes. The commit tree would then look something like the following:



In the above tree diagram, A and B represent our two separate branch objects, which point to our two different commit objects (which are on the same level), and who both point to the same commit object as their parent commit (because they are both differently updated versions of the same file). Hopefully, this makes more clear the idea of what a commit tree is and how it can be used to represent version history.

Now that we have a clear understanding of the different types of git objects and their general interactions with each other, let's quickly explore how these objects are actually stored in a git repository.

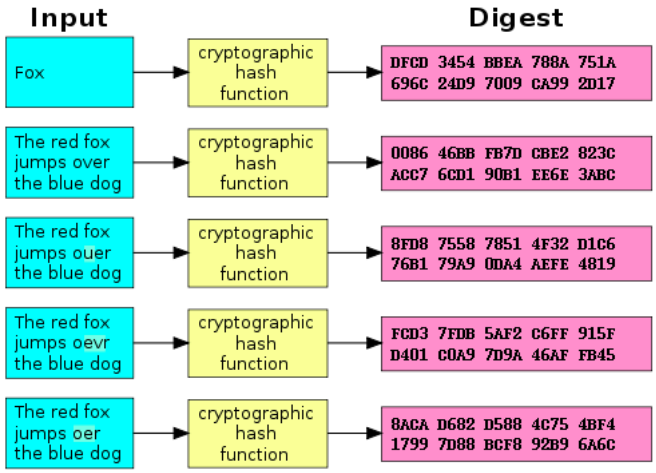
SHA-1 Encryption

Imagine a project where you have several thousand different commits, and an overall large revision history. Many of these files might have similar content, and might share names, so how can we go about uniquely identifying each of these blobs as their own entities?

Instead of using file names or their content as unique identifiers, Git instead resolves to use a handy encryption tool called SHA-1 to serialize and store each blob. SHA, which stands for Secure Hash Algorithm, is a tool developed by the United States' National Security Agency for data encryption. The algorithm takes an input (a file in this case) and produces a unique 40 digit long code known as the object's SHA id, which can then be used to identify the file as its own.

The tool extends beyond just taking files as inputs, and can actually take any "object" as its input parameter. Thus, actually *all* git objects - trees, branches, commits, and blobs - are represented in the git repository with their SHA codes.

If this is still unclear, the following diagram shows how even the slightest change in file content can cause the algorithm to output a completely different SHA-1 identifier. This is what makes the SHA such a valuable tool in a VCS, as it provides us a way of hashing our different file versions.



Applications

Now that we have a solid understanding of what git does and how it actually works, we are ready to look at some of the more specific applications and functions that make git the powerful and applicable tool it is.

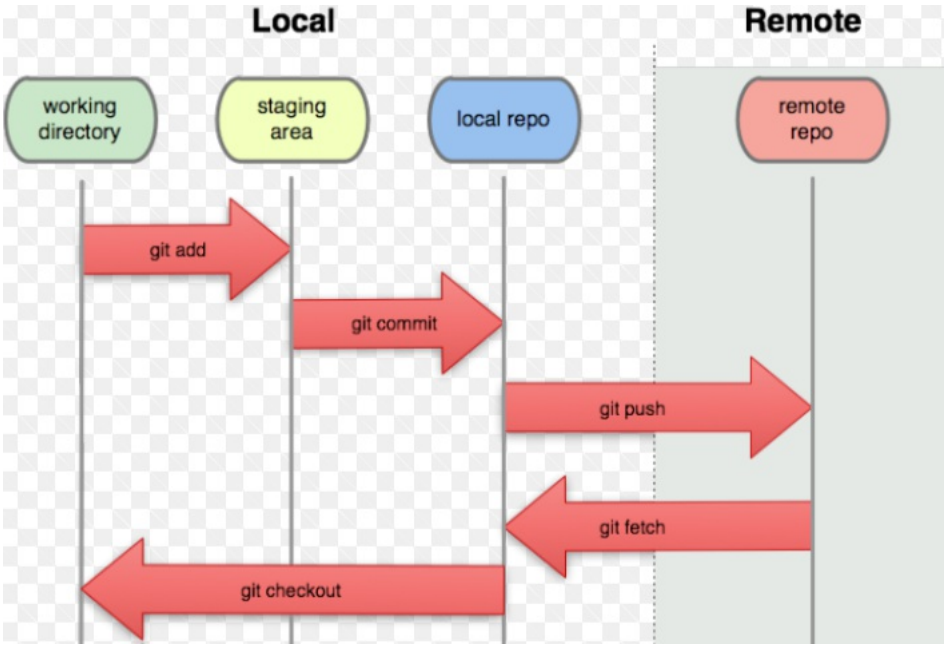
Remote Repositories

While we've learned about how local repositories work and Git's local areas of operations, the true application of Git lies not in its local function, but rather its power of remote operation.

Think about it like this: so say we have 100 versions of our essay stored on our local machine in a git repository, giving us the ability to revert back to any old version in an instant. Now nothing can defeat us! If our text editor freezes or our machine accidentally restarts, we can always just access the most recent version of our file (stored in the most recent commit) and pick up right where we left off. But imagine what were to happen if your machine got stolen or physically broke. Yeah, you had a lot of versions of your files in your git repository, but that entire repository was on your machine. Without your actual machine, you've pretty much lost all your work, regardless of how many times you saved it in the VCS. Well this is where the power of remote repositories comes into play.

Just how we can set up a Git repository in our current working directories on our local machines, we can also do the same thing on remote machines or servers. What's more is that we can then create local copies of those repositories on our machine, which means that we can work on our own machine, and then save that work somewhere else where its secure and safe from any external factors.

The idea of a remote repository simply adds an extra layer to the areas across which git can function. Now, in addition to our working directory, staging area, and local git repository, we have an area which we can refer to as our remote git repository, as illustrated in the revised diagram of before.



Whenever we want to save our work from our local repository to this new remote one, we can simply *push* our work to it. And conversely, whenever we want to collect our work from our remote repository and have it on our local machine, we can simply *fetch* it from the remote.

Indeed, it is the power of remote repositories which make git such an applicable tool, from allowing for much more secure backups to things like mass-collaborative projects.

Collaboration

Let's go back to the simple scenario we looked at in the Version Control section above, where you are working on an essay and are periodically saving your work so as not to lose all your progress. Now in this situation, it is simple enough to just press the "save file" button every now and then. After all, you're the only collaborator, and can hold yourself accountable for making sure to save your work frequently. And even if you forgot to save your work every now and then, you'll probably be able to keep track of what changes you made in the past 15-20 minutes, and can always just edit the document directly should something go wrong.

But imagine what would happen if instead of just one person, it was a crowd of 500 people working on a single essay using a tool like Google Docs. Simply expecting each person to remember to save all their contributions would no longer be feasible. And if one of those 500 people were to accidentally delete all the content of the essay on their end, that would impact everyone's versions as well. Version-control in this scenario becomes a much more messy problem, and that is why a tool like git is critical in logging everyone's contributions and keeping an account of all versions of that essay. This way, even if one person was to delete their version of the essay, no one else's work would be affected and the person could always get their work back, assuming they had pushed all their changes to some central, remote repository.

GitHub

Okay, so we've learned about the power of remote repositories and their application in collaborative development, but how do we actually setup this remote repository without having another entire machine besides our local one? This is where tools like GitHub come into play.

Aside from the menial task of setting up our repositories, we have mostly been dealing with Github in this course for logging the different versions of our assignment submissions. What is often not clear to beginners is that Github is simply a framework upon which git functions.

GitHub is really just a company which provides Git repository hosting, meaning that it is basically a framework which allows you to host remote Git repositories on. It allows you to easily setup remotes, which can then be synced up with your local machine.

Basic Commands

Okay, so now we pretty much know everything we need to know about git. We know about the motivation behind it, we know about how version control works, we know about the different areas upon which git acts, about the different git objects and how they interact with each other, and even how to set up our own remote repository on GitHub. But we have yet to learn how we can actually work with Git. Here's a list of some of the basic git commands, and the purpose of each:

1. git init: initializes a hidden git repository (.git directory) in your current working directory
2. git status: checks the status of your git repository
3. git add [file name]: adds the given file to the staging area
4. git commit -m [commit message]: creates a new commit object and clears the staging area by moving all the blobs from the staging area to the new commit
5. git rm [file name]: removes a file from being tracked in the next commit object
6. git branch [branch name]: creates a new branch object with the given name and points it to the head commit
7. git checkout [branch name]: allows us to switch between different branches
8. git log: view all the changes that have been made (lists all the versions)
9. git push: pushes changes to your remote repository
10. git pull: update local repository by pulling most recent commit from the remote repository
11. git remote add origin ssh://git@github.com/[username]/[repository-name].git: adds a remote repository on github with the given link

We'll make use of some of these commands in the following example, so you get to see firsthand how they work!

A Summarizing Example

Though this example, we'll bring together everything we've learned so far and apply it to create our own remote repository, push and pull files to and from it, and look at some of the more advanced Git functionalities that we haven't gotten a chance to explore in class. Because most of the post has been dedicated to teaching you how Git really works, this example will serve as the reproducible portion of this assignment.

1. Creating a Local Repository We'll start by creating a directory on our local machine where we want to create our local Git repository. Navigate to your Terminal and type in the following command

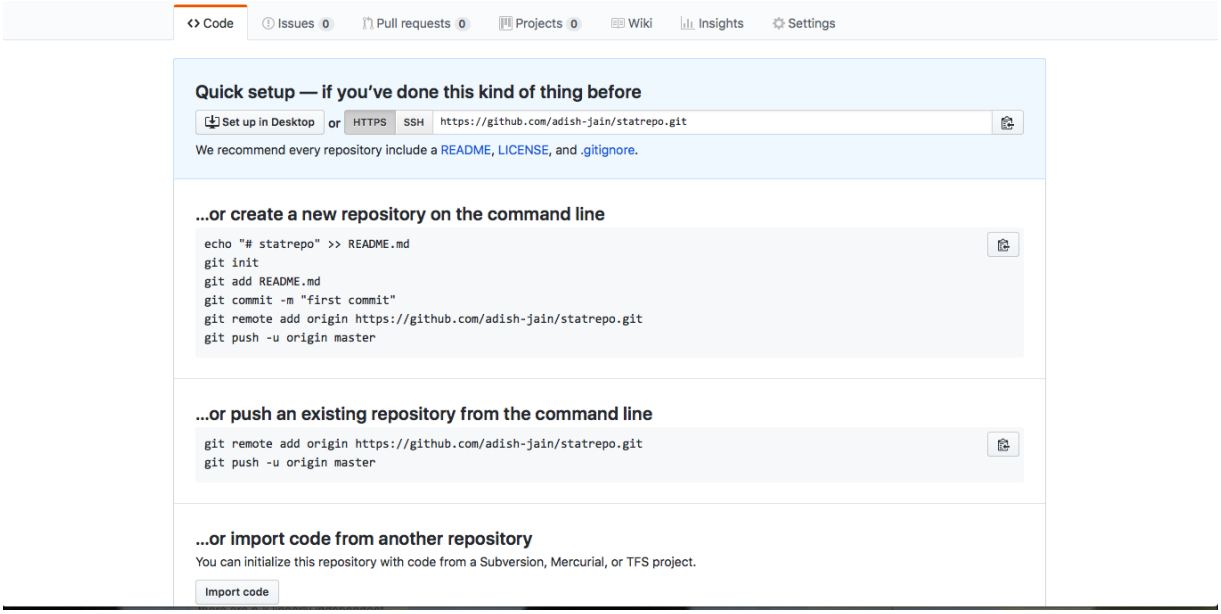
   ```
   mkdir statrepo
   ```

   This will create a new directory called statrepo in your current working directory. This will eventually become our Git repository once we initialize it. But if we just initialize it right now, it will just be a local repository. Remember that our goal is to sync it with a remote repository on GitHub to make use of Git's full functionality.

2. Creating a Repository on GitHub Open up a browser and navigate to github.com. If you don't have an account already, please follow the instructions to sign up for an account. After doing so, click on the plus sign on the top right, and click on "New Repository". Follow the instructions to create a new repository by choosing a name for your repo. Let's name it "statrepo" again, to conform with the local repository we have already setup on our machine. You don't have to put anything in the description right now, and can decide to make the repo public or private. Let's not worry about a README either for now. Go ahead and click "Create Repository" to create the Repo on GitHub.

3. Sync Both Your Repositories

You should now see a page that looks like the following



This is the page we'll need to sync up our local repository with our remote one! Let's navigate back to our *local* statrepo repository on Terminal again. Once you are in the statrepo directory, type the following command:

```
git init
```

which will initialize a Git repository in statrepo (making statrepo a local Git Repository). You should see something like the following message appear:

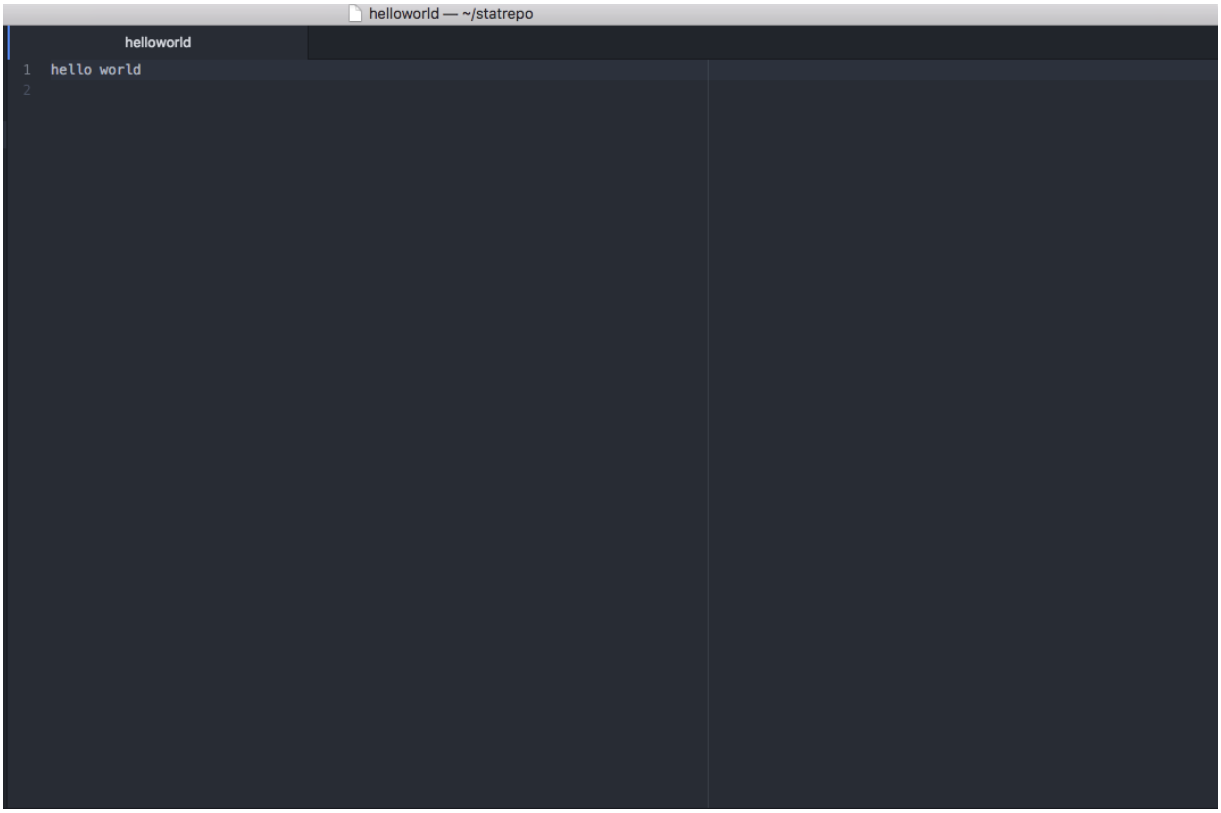
```
Initialized empty Git repository in /Users/[your name]/statrepo/.git/
```

After that, use command number 11 from the list above, and type that into your Terminal

```
git remote add origin https://github.com/[your GitHub username]/statrepo.git
```

which will hook up your local repository to your remote one! And that's it! You've set up a remote repository and synced it to a local repository of your own. Now let's see how we can actually manage files in these environments.

4. Creating a file Let's create a new file that we want to work with. Open up your favorite text editor and write something down.



I've written "hello world" in this case. Make sure to save this file in your local statrepo directory!

Now navigate back to your statrepo on Terminal, and let's see if anything changed. Type in the command

```
ls
```

and you should see the file that you just made in your directory.

Now let's work with some more Git commands. Type in the command

```
git status
```

and you should see something similar to the following

```
Adishs-MacBook-Air:statrepo Adish$ git status
On branch master

No commits yet

Untracked files:
  (use "git add <file>..." to include in what will be committed)

        helloworld

nothing added to commit but untracked files present (use "git add" to track)
```

Notice how Git tells us that our file is currently *untracked*. Going back to our areas of operation, this means that our file is in our current working directory, but is *not* in our staging area, meaning that it's not actually being tracked by Git. To change this, we need to use

```
git add .
```

which will add all the files (just the one in this case). This puts the files in the staging area and readies it to be a blob in our next commit. Typing in "git status" again should now show us the following

```
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)

        new file:   helloworld
```

5. Committing our File The "new file" indicates that we have a new blob in our staging area, but we have yet to commit this blob out of our staging area and actually move it in to our local repository via the commit command. So let's do that by typing in

   ```
   git commit -m "First Commit"
   ```

   You should now see the following in your Terminal

```
Adishs-MacBook-Air:statrepo Adish$ git commit -m "First Commit"
[master (root-commit) ee5bd6a] First Commit
 1 file changed, 1 insertion(+)
 create mode 100644 helloworld
```

which means that a new Commit object has been made. You'll also notice the first 7 digits of the SHA-1 id for this specific commit object, "ee5bd6a", which is how our commit object will be identified in the internal git structure. If we type in

```
git log
```

we'll be able to see more information about our Commit object, including the full SHA-id of our commit, the author, and the date and time it was committed. You'll also be able to see that our HEAD branch points to this commit.

```
Adishs-MacBook-Air:statrepo Adish$ git log
commit ee5bd6a37a842bb68644215c011b6225fb21329f (HEAD -> master)
Author: Adish Jain <adish.jain@berkeley.edu>
Date:   Sun Dec 3 13:26:00 2017 -0800

    First Commit
```

6. Modifying our File Let's modify our hello world file one more time, by adding an exclamation mark at the end of our sentence. Now, if we type in "git status" again, you'll see that Git can identify that our file has been modified, and tells us the following:

```
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        modified:   helloworld
```

We can add and commit our file again by typing in the following commands

```
git add .
git commit -m "Second Commit"
git log
```

and we'll see the following:

```
Adishs-MacBook-Air:statrepo Adish$ git log
commit 6ca768203bb5e6221ca1dbb6cadeff4c615ed912 (HEAD -> master)
Author: Adish Jain <adish.jain@berkeley.edu>
Date:   Sun Dec 3 13:43:31 2017 -0800

    Second Commit

commit ee5bd6a37a842bb68644215c011b6225fb21329f
Author: Adish Jain <adish.jain@berkeley.edu>
Date:   Sun Dec 3 13:26:00 2017 -0800

    First Commit
```

which shows us that we now have a second, more recent, commit object to which our HEAD branch points to. Okay perfect, everything seems to be working fine locally.

7. Pushing to our Remote

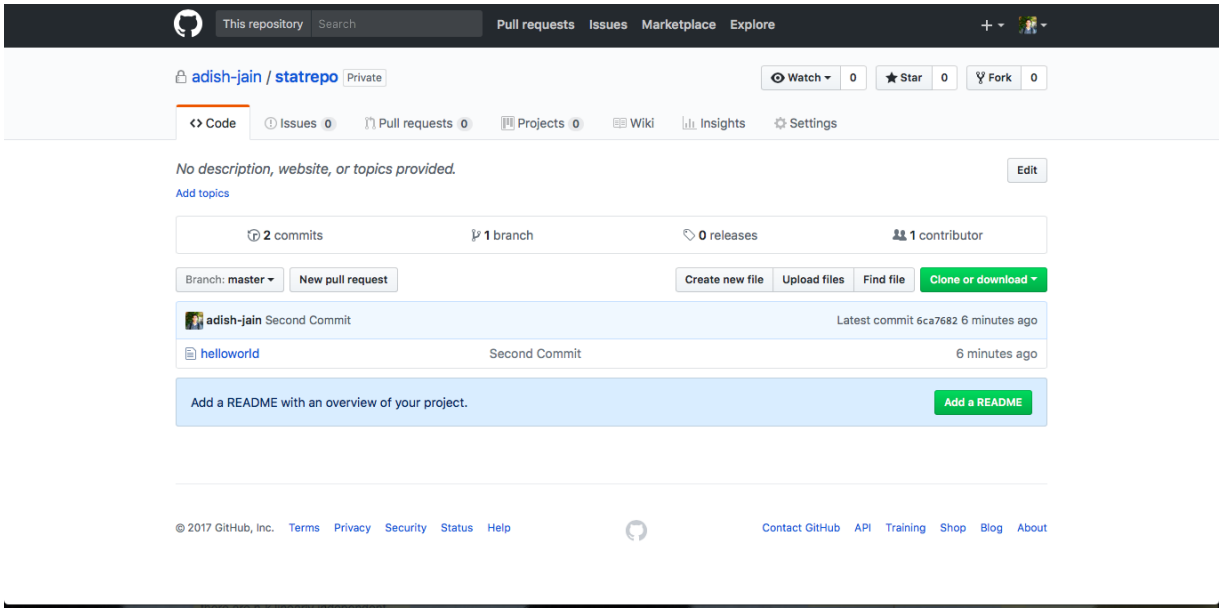
Let's finally push our file to our remote repository, by typing in the following command

```
git push origin master
```

you should see the following appear on your Terminal

```
Adishs-MacBook-Air:statrepo Adish$ git push origin master
Counting objects: 6, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (6/6), 463 bytes | 463.00 KiB/s, done.
Total 6 (delta 0), reused 0 (delta 0)
To https://github.com/adish-jain/statrepo.git
 * [new branch]      master -> master
```

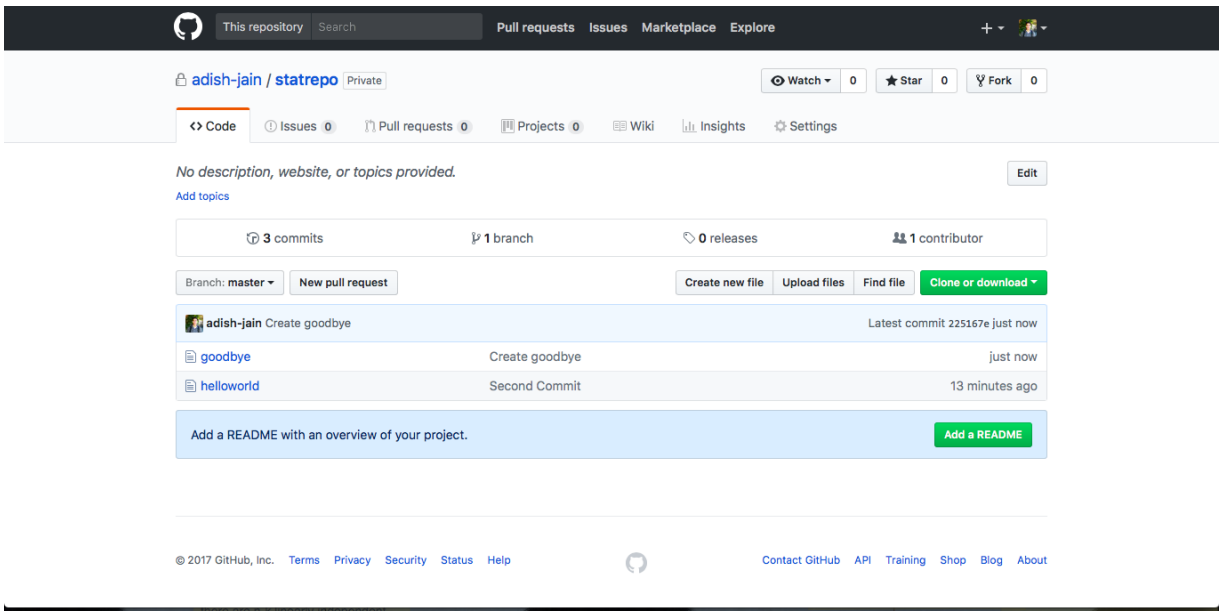
meaning that our file has been pushed to our remote. Let's check our GitHub repository that we previously setup and we'll now see the following



Isn't that cool? The file that we had on our local repository is now in our remote repository! But can we go the other way around? Can we create something on our remote repository and pull into our local? Let's try it.

8. Pulling to Our Local Repository

Click the "Create new file" button on the top right of your GitHub repo, and create a new file called "goodbye" with the word "goodbye" in it, like the following.



You should see the goodbye file appear in your remote repo. Now let's go back to our local repo on terminal, and type in the following commands

```
git pull origin master
ls
```

We will see the following

```
Adishs-MacBook-Air:statrepo Adish$ git pull origin master
From https://github.com/adish-jain/statrepo
 * branch            master     -> FETCH_HEAD
Updating 6ca7682..225167e
Fast-forward
 goodbye | 1 +
 1 file changed, 1 insertion(+)
 create mode 100644 goodbye
Adishs-MacBook-Air:statrepo Adish$ ls
goodbye         helloworld
```

which indicate to us that we now have the file "goodbye" in our local repository!

9. Creating a New Branch

Cool, so we have explored how to add files to our local repository, remote repository, and push and pull between the two. What else can we do with Git though? Let's try creating a new branch using the command

```
git branch second
```

which will create a new branch object called "second" which points to our most recent commit object, as described before. Now let's try changing the text in our helloworld file again, adding and committing it. Let's change the text by adding a second exclamation mark, making it "hello world!!", and use the following commands

```
git add .
git commit -m "Third Commit"
git log
```

Checkout what happened:

```
Adishs-MacBook-Air:statrepo Adish$ git log
commit 0e3a4476d432539fa4d91a68e7a68d9d5122cb59 (HEAD -> master)
Author: Adish Jain <adish.jain@berkeley.edu>
Date:   Sun Dec 3 14:25:02 2017 -0800

    Third Commit

commit 225167ebdf4c24dd1b35347a189061e61f2a7c12 (origin/master, second)
Author: Adish Jain <adish.jain@berkeley.edu>
Date:   Sun Dec 3 13:56:13 2017 -0800

    Create goodbye

commit 6ca768203bb5e6221ca1dbb6cadeff4c615ed912
Author: Adish Jain <adish.jain@berkeley.edu>
Date:   Sun Dec 3 13:43:31 2017 -0800

    Second Commit

commit ee5bd6a37a842bb68644215c011b6225fb21329f
Author: Adish Jain <adish.jain@berkeley.edu>
```

So our HEAD branch from before now points to our most recent commit, but the other branch that we'd just created, "second", still points to our second most recent commit. So now we have two separate branch objects, each pointing to two *different* commit objects.

10. Checking Out

Okay, so recall that our helloworld file reads "hello world!!". But with the following commands

```
git checkout second
git log
```

we'll see this displayed on our Terminal

```
Adishs-MacBook-Air:statrepo Adish$ git checkout second
Switched to branch 'second'
Adishs-MacBook-Air:statrepo Adish$ git log
commit 225167ebdf4c24dd1b35347a189061e61f2a7c12 (HEAD -> second, origin/master)
Author: Adish Jain <adish.jain@berkeley.edu>
Date:   Sun Dec 3 13:56:13 2017 -0800

    Create goodbye

commit 6ca768203bb5e6221ca1dbb6cadeff4c615ed912
Author: Adish Jain <adish.jain@berkeley.edu>
Date:   Sun Dec 3 13:43:31 2017 -0800

    Second Commit

commit ee5bd6a37a842bb68644215c011b6225fb21329f
Author: Adish Jain <adish.jain@berkeley.edu>
Date:   Sun Dec 3 13:26:00 2017 -0800

    First Commit
```

which indicates that the tree structure we are now looking at is the one pointed to by the SECOND branch instead of the HEAD branch we had before. In other words, our new HEAD is SECOND, and our previous branch - which is by default called MASTER - is just another branch. If we now open up and read our "helloworld" file, you'll notice that it reads "hello world!". So by checking out, we actually change the branch we are looking at and our blobs (our files) take on the versions of that specific commit.

Alright! That's it. If you were able to follow the steps in this example, you have mastered the basics of Git and its internals. You are now ready to step out into the real world, build your own projects, and save them using Git as your underlying version control system. Good luck!

References

1. https://www.git-tower.com/learn/git/ebook/en/desktop-gui/basics/why-use-version-control
2. https://www.visualstudio.com/learn/understand-git-history/
3. https://confluence.atlassian.com/bitbucketserver/basic-git-commands-776639767.html
4. https://www.lifewire.com/what-is-sha-1-2626011
5. https://www.sbf5.com/~cduan/technical/git/git-1.shtml
6. https://juristr.com/blog/2013/04/git-explained/
7. https://www.ibm.com/developerworks/library/d-learn-workings-git/index.html
8. https://git-scm.com/book/en/v1/Git-Internals-Git-References
9. https://softwareengineering.stackexchange.com/questions/173321/conceptual-difference-between-git-and-github