Git Basics

What is Version Control?

Version control is simply keeping track of changes to a file or several files over time. The idea is to always be able to go back to a specific version of a file in case anything goes wrong. This can be done in many different ways. The most obvious way to do this would be to save a file as a new version each time a change is made. For example, when working on a document, this method would leave your filesystem looking a little like this:

```
GitBasics_V1.txt    - 2KB
GitBasics_V1.2.txt  - 2.4KB
GitBasics_V2.txt    - 2.5KB
GitBasics_V2.8.txt  - 3.3KB
GitBasics_V3.txt    - 4.9KB
```

This way, you can always go back to any version of the file you would like. If you need to completely revert to an older version, simply delete all the versions, or copy the older version to a new version file.

Can you think of any problems with this system however? Probably not since the example shown is just a bunch of tiny text files. Imagine a photographer saving each version of their high res images in this manner, those file sizes will definitely add up very quickly. Moreover, as programmers, the names of our files are often crucial. For example, in Java, the name of a source file must match the name of the class that the file contains. This is a huge problem because we would have to rename each class every time a new version is created. The last thing we want to do is to have to change our source code to handle our version control. So instead, we use version control software.

The idea of version control software (VCS) is to keep track of only the changes made to a file. The way a VCS would do this is to keep track of what is called a **repository** which is a collection of all the files that will be tracked. So, for every file that a repository **tracks**, the repository keeps the starting point of the file, and all the individual changes made to that file since it was added to the repository. Therefore, a file can be constructed by taking the original version and applying all the changes since. Using a VCS comes with many advantages including tools that allow teams of developers to collaborate on the same code base. The VCS that we will be going over is also one of the most widely used VCS's in the current industry, it is called Git.

Git

As stated above, Git is the most popular VCS in the industry by far. One of the reasons for this is because it uses a Distributed Model which you can learn more about on your own time.

When working with Git, the repository is saved in a folder called `.git` in the root folder of the code base. This folder is managed by Git and should not be touched by a developer. The code base itself is just the files in the folder that contains the `.git` folder. As you create files, they will be **untracked**, which means that the file has not yet been added to the repository and so version control will not act on that file. As soon as you add a file to the repository it will become **tracked**, meaning that the file now be included in version control.

You may have also heard of GitHub, please note that Git and GitHub are not the same thing.

Installing Git

To use Git, you must have it installed on your computer. Installing Git is a relatively simple process that differs slightly based on your operating system. On Windows, simply go to https://git-scm.com/downloads and download the windows installer.

For Linux, you can simply use `apt-get` or similar package managers to install Git. Here is the command to install using `apt-get`:

```
sudo apt install git
```

There are several ways to install on Mac, the easiest being to install the Xcode Command Line Tools.

If you would like to verify whether or not you already have Git installed on your system, simply run the following command in your command line.

```
git --version
```

The next thing we need to do is configure our user information for all our local repositories. What you will need to set up is your name and email since that information needs to be attached to every change you make to a file. Do that with the following commands:

```
git config --global user.name "[your name]"
git config --global user.email "[your email address]"
```

Now that we have Git installed and set up, let's jump into actually using it.

Creating a Git Repository

Creating a repository in Git is incredibly simple. Simply use the following command:

```
git init
```

This will create an empty Git repository in the current working directory of your terminal. You can verify this by checking for the `.git` folder in your directory. Now note that any files that are also in the directory you created the repository in are untracked by default.

If you would like to create a repository completely from scratch, meaning there will be no untracked files initially, you can use the following variant of the `init` command.

```
git init [repository-name]
```

This, instead of creating the Git repository in the current directory, will create a blank folder with the given name and then create the Git repository in that new folder. It is best practices to create Git repositories in folders with all lower case names and no spaces.

The Git Workflow

The processes of tracking changes in a repository is relatively simple, but might seem a little tricky to absolute beginners. Git keeps track of changes to a repository through commits, a **commit** is a snapshot of all the changes made to a repository at a certain time. Each commit has its own identifier, a unique number, and a message. The message is meant to describe what was done to the files in that commit. In order to commit changes to a repository, they must first be **staged**, which now leads us to taking a look at the 4 areas when working with Git.

- Untracked: These are the files that are untracked are not yet included in the repository at all.
- Unstaged: These are the changes to tracked files that have not yet been staged.
- Staged: These are the changes to the repository that are soon going to be committed.
- Repository: The collection of all commits.

Creating a New File

Now let's suppose we had a blank repository in a folder called `basics`. Let's create a file called `a.txt` in the root of that folder with some random text inside. Right now, that file is **untracked** because it has never been committed to the repository before. We can see this using the following command:

```
git status
```

`git status` shows us the current state of the repository, specifically which files have been created, modified, and staged. In our example, we would get the following output:

```
On branch master

No commits yet

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

        a.txt

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

As you can see, `a.txt` is untracked. Before we can commit it to the repository, we must stage this file first. To do so, we will use the `git add` command:

```
git add a.txt
```

Using the `git add` command moves our untracked file into the staging area. We can verify this again with `git status`:

```
On branch master

No commits yet

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

        new file:   a.txt
```

Committing Changes

Now that our changes are staged, let's commit them to our repository. The command to commit is quite simple

```
git commit
```

This will open your shell's editor, there you must type a commit message. The commit message is supposed to be a brief (<70 characters) description of what changes were made in the commit. If you would like to use a different editor (such as `nano`, `vi`, `emacs`, etc.) You can use the following command:

```
git config --global core.editor [editor]
```

Alternatively, you can provide a commit message directly with the `commit` command as a parameter. This is my preferred way:

```
git commit -m "[message]"
```

Since this is my preferred way, I will commit `a.txt` to my repository with the following command:

```
git commit -m "Added a.txt"
```

And I will get the following response:

```
[master (root-commit) 0f841d0] Added a.txt
 1 file changed, 1 insertion(+)
 create mode 100644 a.txt
```

This commit is now permanently logged in my repository's history. I can see this by using the following command:

```
git log
```

This gives me this output:

```
commit 0f841d0b9daf9980473390c6c389d89456167d03 (HEAD -> master)
Author: Jose Rodriguez Rivas <example@example.com>
Date:   Sat Feb 8 13:35:16 2020 -0800

    Added a.txt
```

The long number there is the ID of the commit, if I ever need to reference the commit for whatever reason I would need that ID.

Modifying Existing Files

So great! We have successfully created a new file and added it to our repository. Now let's modify the file. Running `git status` would give us the following:

```
On branch master
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:   a.txt

no changes added to commit (use "git add" and/or "git commit -a")
```

Now, instead of being untracked, `a.txt` is now modified, or unstaged. More specifically, the *changes* made to `a.txt` are unstaged. Moving those changes to the staging area is actually the same command from earlier:

```
git add a.txt
```

Now our changes have been staged, verifiable by `git status`

```
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        modified:   a.txt
```

What do you think will happen if I modify `a.txt` again? You can assume that those changes would be staged since `a.txt` is in the staging area. However, remember earlier I said that the changes to `a.txt` are in the staging area, so any changes I make after staging will be unstaged. You can see this below:

```
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        modified:   a.txt

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:   a.txt
```

I can simply use `git add a.txt` again to include the new changes to the staging area before I commit the changes. I will now commit those changes to the repository:

```
git commit -m "Updated a.txt"
```

And get the following response:

```
[master 5d96418] Updated a.txt
 1 file changed, 3 insertions(+), 1 deletion(-)
```

I can see the two commits now using `git log` again.

```
commit 5d9641819afc4f68b2a65d4303053e3938e1937c (HEAD -> master)
Author: Jose Rodriguez Rivas <example@example.com>
Date:   Sat Feb 8 13:53:10 2020 -0800

    Updated a.txt

commit 0f841d0b9daf9980473390c6c389d89456167d03
Author: Jose Rodriguez Rivas <example@example.com>
Date:   Sat Feb 8 13:35:16 2020 -0800

    Added a.txt
```

Note that the `(HEAD -> master)` is on the most recent commit. The `HEAD` is simply a reference to a commit, so I can use `HEAD` instead of the long ID. By default, `HEAD` is assigned to the most recent commit made.

Viewing Changes

Now suppose we modified `a.txt` again. Before we stage the changes, we can check the changes using the following command:

```
git diff
```

Gives me:

```
diff --git a/a.txt b/a.txt
index 85722fd..c21df61 100644
--- a/a.txt
+++ b/a.txt
@@ -1,3 +1,3 @@
-Hello world, my name is Jose!
+Hello world! My name is Jose!
 Pleased to meet you!
```

Now if I stage those changes, I will get nothing from `git diff` this is because the command shows us the changes that are not yet staged. I can change this by using `git diff` with the following options:

```
git diff --staged
```

This will give me the same output as before.

Adding Multiple Files at Once

As programmers, chances are we are going to be working on several files at a time. Adding each file we edit to our staging area one at a time can be very tedious. We can use a simple trick to add all the files that we have modified to our staging area at once. We are going to be using the fact that when we give a directory to the `git add` command, all the files in that directory will be added. This is recursive, so all files in subdirectories will also be added. I am going to write some simple C code to show this. Here is the result of `git status` after writing the C code.

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

        sample.c
        sample.h

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

Now, I could just add each of these files individually, or together in the `git add` command, but I do not want to do it this way. Imagine I had 20 files that I updated, that would get incredibly tedious, especially if those files are not in the same folder. So what I am going to do is run the following command

```
git add .
```

Running this on the root of my repository will add all updated files, including untracked files. This is because `.` means the current working directory. So, if I pass in the root directory to the `git add` command, which is recursive, then all the files that were changed will be added to the staging area. This can be verified with `git status`

```
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        new file:   sample.c
        new file:   sample.h
```

Easy as that! Now I can just commit my files as always and be good to go.

Ignoring Files

An important part of maintain a Git repository is knowing that not all files belong in the repository. These are files that we want to be ignored. To do so, we create a special file called `.gitignore`. What this file does is defines which files, or types of files, should be completely ignored by Git. These are files such as binary executables, OS files, etc (More Info). `.gitignore` files are applied recursively to the folder they are in, and to every subfolder. Therefore, you can have several in your repository, or just a single root one, or both!

So what I am going to do is compile my code. Now here is the `git status`.

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

        a.out

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

However, I do not want `a.out` in my repository so I am going create a `.gitignore` file with the following content:

```
a.out
```

As you can see, `a.out` is no longer unstaged, and `.gitignore` is untracked. Now, if I execute `git add .`, `a.out` will not be included.

Resetting Commits

Let's say that someone comes in and writes some nasty code, or for whatever reason, we want to reset our repository back to a certain point in time. Here is a `git log` after adding some nonsense code that I would like to undo

```
commit c8ab479194e7e37a93ffabe83db38991baaf9121 (HEAD -> master)
Author: Jose Rodriguez Rivas <example@example.com>
Date:   Sat Feb 8 18:46:02 2020 -0800

    Bad code

commit ea54946d6bbcb8352f1b3207d6bed700a288e0d3
Author: Jose Rodriguez Rivas <example@example.com>
Date:   Sat Feb 8 18:40:58 2020 -0800

    Created gitignore

commit 825f6fcd2914efb6ec82da86c55fd94e15f66365
Author: Jose Rodriguez Rivas <example@example.com>
Date:   Sat Feb 8 17:03:39 2020 -0800

    Created sample program

commit ad7cad639a1f912eec47cf78d12813c75ca616e7
Author: Jose Rodriguez Rivas <example@example.com>
Date:   Sat Feb 8 14:03:43 2020 -0800

    Updated a again

commit 5d9641819afc4f68b2a65d4303053e3938e1937c
Author: Jose Rodriguez Rivas <example@example.com>
Date:   Sat Feb 8 13:53:10 2020 -0800

    Updated a.txt

commit 0f841d0b9daf9980473390c6c389d89456167d03
Author: Jose Rodriguez Rivas <example@example.com>
Date:   Sat Feb 8 13:35:16 2020 -0800

    Added a.txt
```

What I am going to do is copy the ID of the commit that I want to revert my repository to by using the following command:

```
git reset ea54946d6bbcb8352f1b3207d6bed700a288e0d3
```

Doing this gave me this output:

```
Unstaged changes after reset:
M       sample.c
```

So what happened here, is that the commits after the commit I put in the reset command were removed, but the files themselves were not changed. This can be what you want, but if you want to revert the repository and actually undo the changes, use the following command:

```
git reset --hard ea54946d6bbcb8352f1b3207d6bed700a288e0d3
```

This gives me this output:

```
HEAD is now at ea54946 Created gitignore
```

And after running `git status`:

```
On branch master
nothing to commit, working tree clean
```

As you can see, files were completely reverted back to the commit I provided in the `reset` command.

Branching

So far, we have only worked on one branch, `master`, which you can probably see in all of the calls to `git status`. A branch is basically a string of commits. Commits can be added to commits in parallel. You can list the branches with the following command:

```
git branch --list
```

This gives us this output:

```
* master
```

So right now, the only branch is `master`, and the `*` shows that we are currently on the `master` branch. Now, I want to make a new branch called `some-branch` with the following command:

```
git branch some-branch
```

Now, here is the list of branches:

```
* master
  some-branch
```

The new branch, `some-branch`, has all the commits at the point we ran the `git branch` command. You can also see that we are still on the `master` branch. So any new commits will be created on the `master` branch. If we want to change to the new branch we just need to run the following command:

```
git checkout some-branch
```

This gives us the following output:

```
Switched to branch 'some-branch'
```

Now, if we list the branches we can see the following:

```
  master
* some-branch
```

You can now see that `some-branch` is now the selected branch. So if I made a commit, it will be made on the `some-branch` branch. Here's the `git log`

```
commit 531864e58d346529763729880ebeff7476bfce86 (HEAD -> some-branch)
Author: Jose Rodriguez Rivas <example@example.com>
Date:   Sat Feb 8 19:20:56 2020 -0800

    Added a nice message

commit ea54946d6bbcb8352f1b3207d6bed700a288e0d3 (master)
Author: Jose Rodriguez Rivas <example@example.com>
Date:   Sat Feb 8 18:40:58 2020 -0800

    Created gitignore

commit 825f6fcd2914efb6ec82da86c55fd94e15f66365
Author: Jose Rodriguez Rivas <example@example.com>
Date:   Sat Feb 8 17:03:39 2020 -0800

    Created sample program

commit ad7cad639a1f912eec47cf78d12813c75ca616e7
Author: Jose Rodriguez Rivas <example@example.com>
Date:   Sat Feb 8 14:03:43 2020 -0800

    Updated a again

commit 5d9641819afc4f68b2a65d4303053e3938e1937c
Author: Jose Rodriguez Rivas <example@example.com>
Date:   Sat Feb 8 13:53:10 2020 -0800

    Updated a.txt

commit 0f841d0b9daf9980473390c6c389d89456167d03
Author: Jose Rodriguez Rivas <example@example.com>
Date:   Sat Feb 8 13:35:16 2020 -0800

    Added a.txt
```

If I now checkout the master branch, the changes I made in the `some-branch` branch will no longer be applied to the repository. And the `git log` will not have the commits that were made on `some-branch`. Now I am going to make changes to a different file on `master`. And the `git log` will show only the commit I just made on `master`.

```
commit acdf74e5e0d0601946bb5499a28df536144a809c (HEAD -> master)
Author: Jose Rodriguez Rivas <example@example.com>
Date:   Sat Feb 8 19:24:58 2020 -0800

    Added some comments

commit ea54946d6bbcb8352f1b3207d6bed700a288e0d3
Author: Jose Rodriguez Rivas <example@example.com>
Date:   Sat Feb 8 18:40:58 2020 -0800

    Created gitignore

commit 825f6fcd2914efb6ec82da86c55fd94e15f66365
Author: Jose Rodriguez Rivas <example@example.com>
Date:   Sat Feb 8 17:03:39 2020 -0800

    Created sample program

commit ad7cad639a1f912eec47cf78d12813c75ca616e7
Author: Jose Rodriguez Rivas <example@example.com>
Date:   Sat Feb 8 14:03:43 2020 -0800

    Updated a again

commit 5d9641819afc4f68b2a65d4303053e3938e1937c
Author: Jose Rodriguez Rivas <example@example.com>
Date:   Sat Feb 8 13:53:10 2020 -0800

    Updated a.txt

commit 0f841d0b9daf9980473390c6c389d89456167d03
Author: Jose Rodriguez Rivas <example@example.com>
Date:   Sat Feb 8 13:35:16 2020 -0800

    Added a.txt
```

Now what I want is for the commits I made on `some-branch` to also appear on the `master` branch. This is called **merging**. To merge a branch, I need to have the destination branch checked out, so staying on `master`, I will run the following command:

```
git merge some-branch
```

Doing so may open a shell editor to enter a commit message, or you can add `-m` just like the `git commit` command. The command gave the following output:

```
Merge made by the 'recursive' strategy.
 sample.c | 1 +
 1 file changed, 1 insertion(+)
```

Now the `git log` gives me the following (note the `:` at the bottom was just because there were too many commits to show at once)

```
commit cb35435dcd01677d2fd3fa995b256dd5e46bb504 (HEAD -> master)
Merge: acdf74e 531864e
Author: Jose Rodriguez Rivas <example@example.com>
Date:   Sat Feb 8 19:28:05 2020 -0800

    Merge branch 'some-branch'

commit acdf74e5e0d0601946bb5499a28df536144a809c
Author: Jose Rodriguez Rivas <example@example.com>
Date:   Sat Feb 8 19:24:58 2020 -0800

    Added some comments

commit 531864e58d346529763729880ebeff7476bfce86 (some-branch)
Author: Jose Rodriguez Rivas <example@example.com>
Date:   Sat Feb 8 19:20:56 2020 -0800

    Added a nice message

commit ea54946d6bbcb8352f1b3207d6bed700a288e0d3
Author: Jose Rodriguez Rivas <example@example.com>
Date:   Sat Feb 8 18:40:58 2020 -0800

    Created gitignore

commit 825f6fcd2914efb6ec82da86c55fd94e15f66365
Author: Jose Rodriguez Rivas <example@example.com>
Date:   Sat Feb 8 17:03:39 2020 -0800

    Created sample program

commit ad7cad639a1f912eec47cf78d12813c75ca616e7
Author: Jose Rodriguez Rivas <example@example.com>
Date:   Sat Feb 8 14:03:43 2020 -0800

    Updated a again

commit 5d9641819afc4f68b2a65d4303053e3938e1937c
Author: Jose Rodriguez Rivas <example@example.com>
Date:   Sat Feb 8 13:53:10 2020 -0800

    Updated a.txt

commit 0f841d0b9daf9980473390c6c389d89456167d03
Author: Jose Rodriguez Rivas <example@example.com>
Date:   Sat Feb 8 13:35:16 2020 -0800
:
```

Now, the `some-branch` branch is out of date, so I will checkout that branch and merge master with the following commands:

```
git checkout some-branch
git merge master
```

Since no files were actually changed, no new commit is necessary. Here is the output:

```
Updating 531864e..cb35435
Fast-forward
 sample.h | 1 +
 1 file changed, 1 insertion(+)
```

And here is the `git log`:

```
commit cb35435dcd01677d2fd3fa995b256dd5e46bb504 (HEAD -> some-branch, master)
Merge: acdf74e 531864e
Author: Jose Rodriguez Rivas <example@example.com>
Date:   Sat Feb 8 19:28:05 2020 -0800

    Merge branch 'some-branch'

commit acdf74e5e0d0601946bb5499a28df536144a809c
Author: Jose Rodriguez Rivas <example@example.com>
Date:   Sat Feb 8 19:24:58 2020 -0800

    Added some comments

commit 531864e58d346529763729880ebeff7476bfce86
Author: Jose Rodriguez Rivas <example@example.com>
Date:   Sat Feb 8 19:20:56 2020 -0800

    Added a nice message

commit ea54946d6bbcb8352f1b3207d6bed700a288e0d3
Author: Jose Rodriguez Rivas <example@example.com>
Date:   Sat Feb 8 18:40:58 2020 -0800

    Created gitignore

commit 825f6fcd2914efb6ec82da86c55fd94e15f66365
Author: Jose Rodriguez Rivas <example@example.com>
Date:   Sat Feb 8 17:03:39 2020 -0800

    Created sample program

commit ad7cad639a1f912eec47cf78d12813c75ca616e7
Author: Jose Rodriguez Rivas <example@example.com>
Date:   Sat Feb 8 14:03:43 2020 -0800

    Updated a again

commit 5d9641819afc4f68b2a65d4303053e3938e1937c
Author: Jose Rodriguez Rivas <example@example.com>
Date:   Sat Feb 8 13:53:10 2020 -0800

    Updated a.txt

commit 0f841d0b9daf9980473390c6c389d89456167d03
Author: Jose Rodriguez Rivas <example@example.com>
Date:   Sat Feb 8 13:35:16 2020 -0800
:
```

Merge Conflicts

Working with different branches, and especially with different people will inevitably lead to merge conflicts. A merge conflict happens trying to merge when the same file was updated in the two branches. So now I am going to make some commits to the same file on the two branches. Now, after trying to merge the two branches I get the following output:

```
Auto-merging sample.c
CONFLICT (content): Merge conflict in sample.c
Automatic merge failed; fix conflicts and then commit the result.
```

Now, if I edit my file, the following will appear:

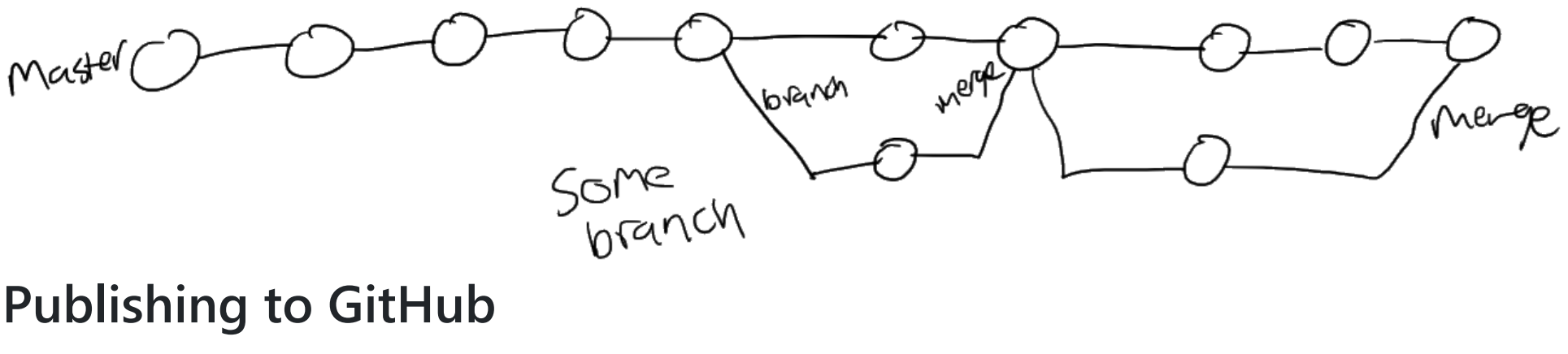
```
<<<<<<< HEAD

// This is the main function. Gonna conflict!
=======
// This is the main function
>>>>>>> some-branch
```

I can now update `sample.c` to keep the version of the file I want to keep. Also, if I run `git status` you will see the following:

```
On branch master
You have unmerged paths.
  (fix conflicts and run "git commit")
  (use "git merge --abort" to abort the merge)

Unmerged paths:
  (use "git add <file>..." to mark resolution)

        both modified:   sample.c

no changes added to commit (use "git add" and/or "git commit -a")
```

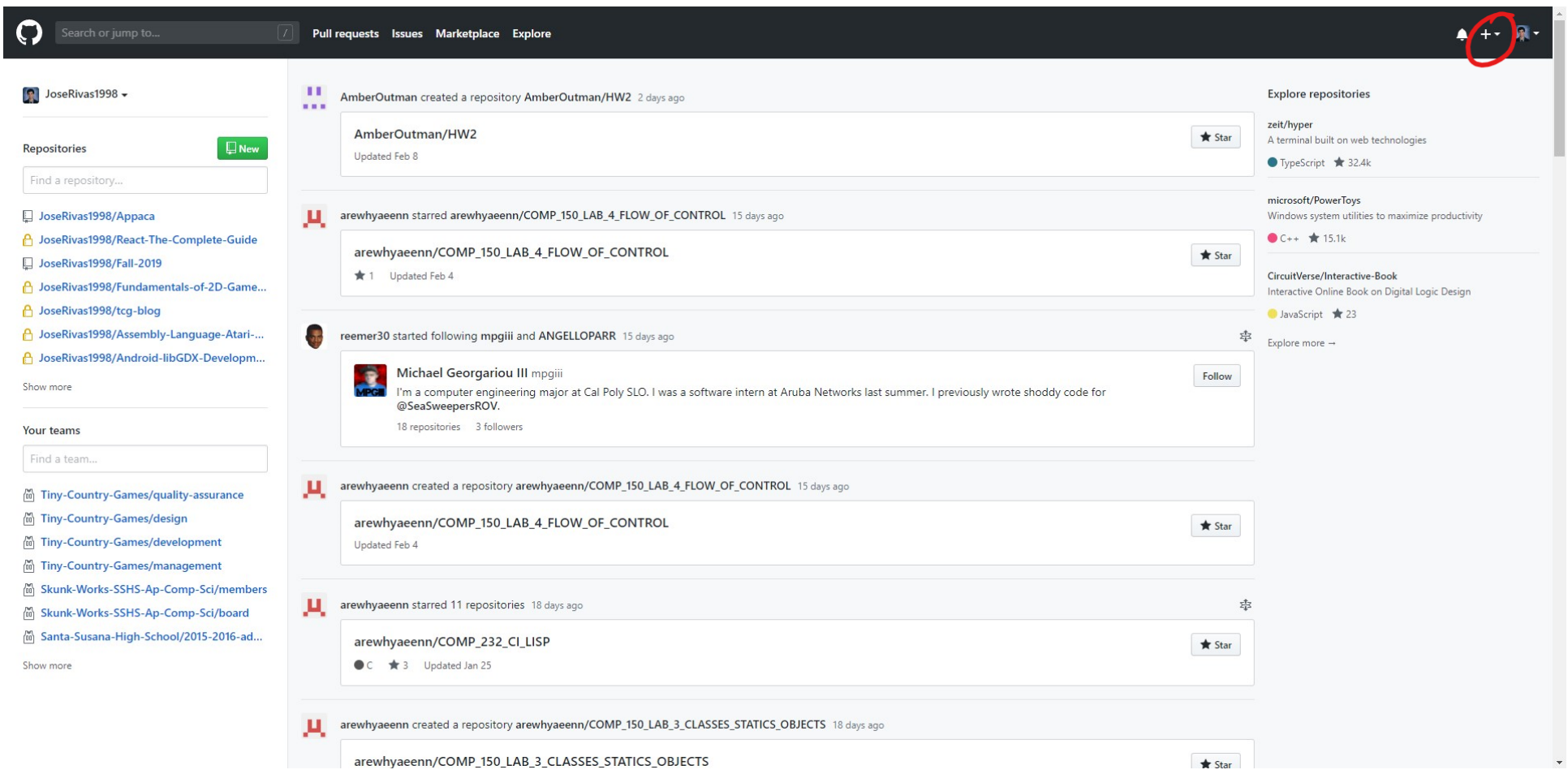
I can now complete the merge by running `git add` and `git commit`, which will create a new commit with the merged file. And voila! The branches are now merge successfully.

A Git repository can be described as a tree, so below will be a diagram where each bubble is a commit.

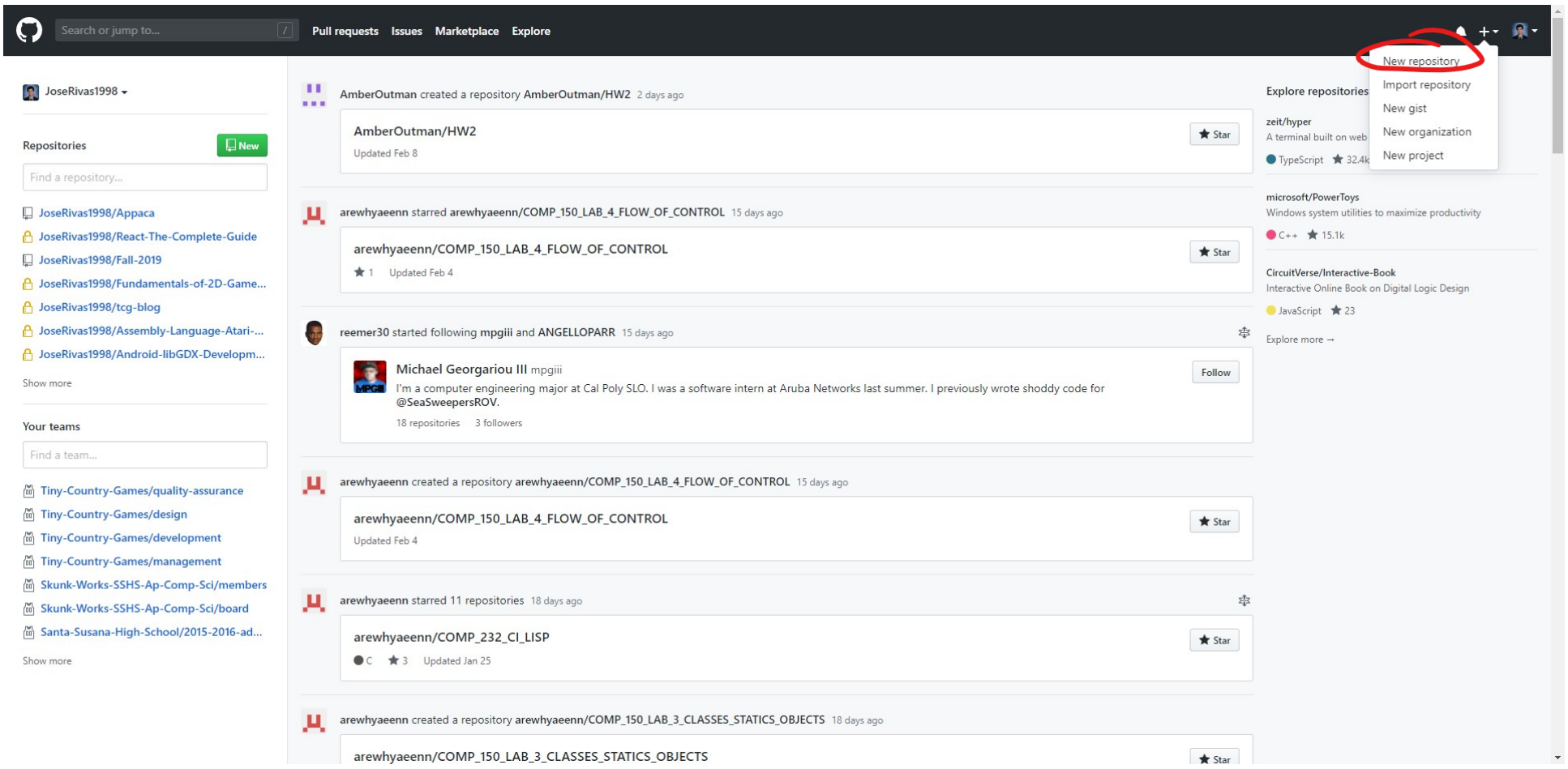


Publishing to GitHub

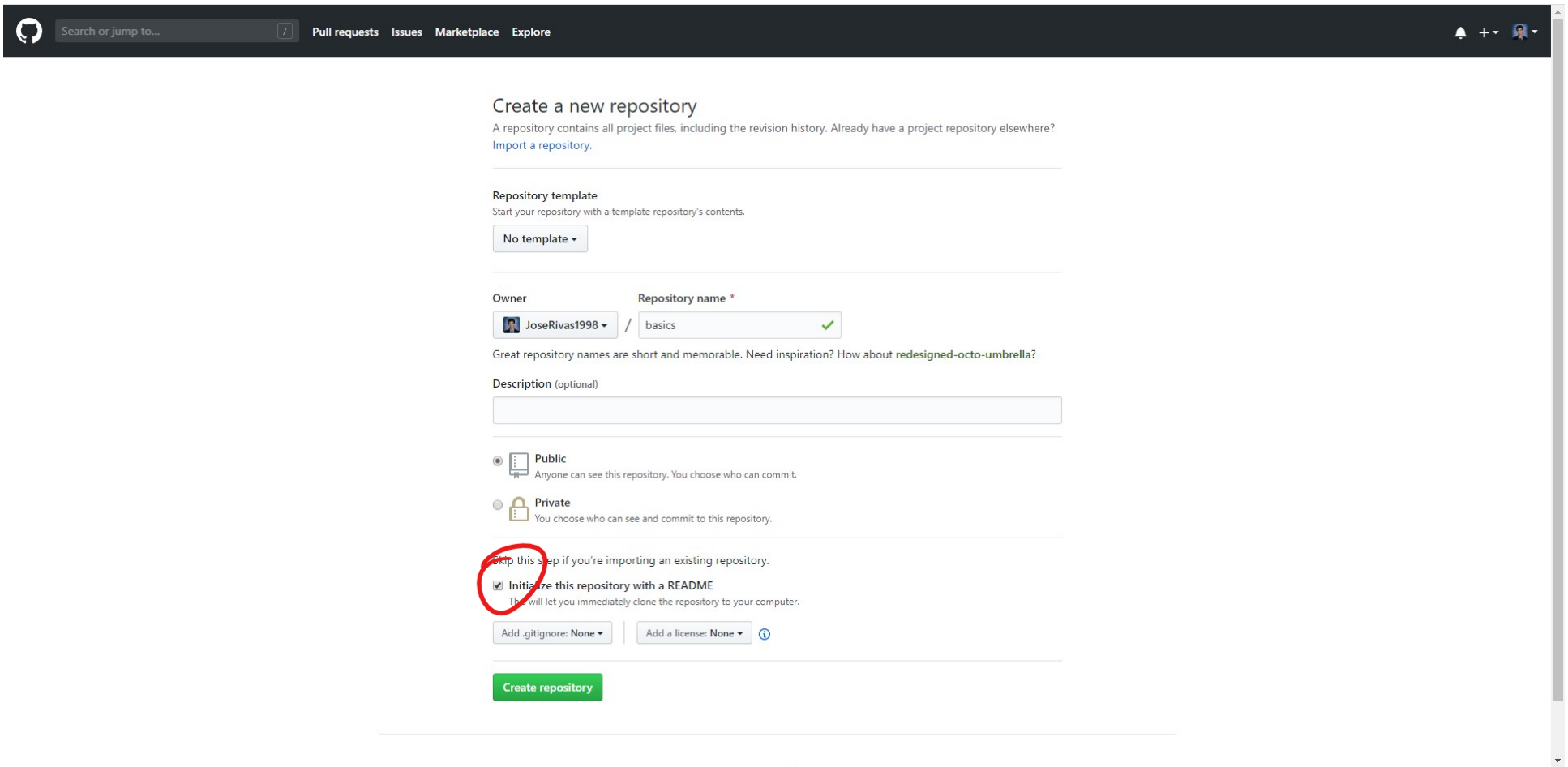
Now, we are going to talk about GitHub, as I said, Git and GitHub are not the same thing, though they are often confused. GitHub is a service that can host your Git repositories for you on the cloud. Creating an account is free, or you can pay to use private repositories. So, having an account I will make a repository by clicking the circled button in the image below:



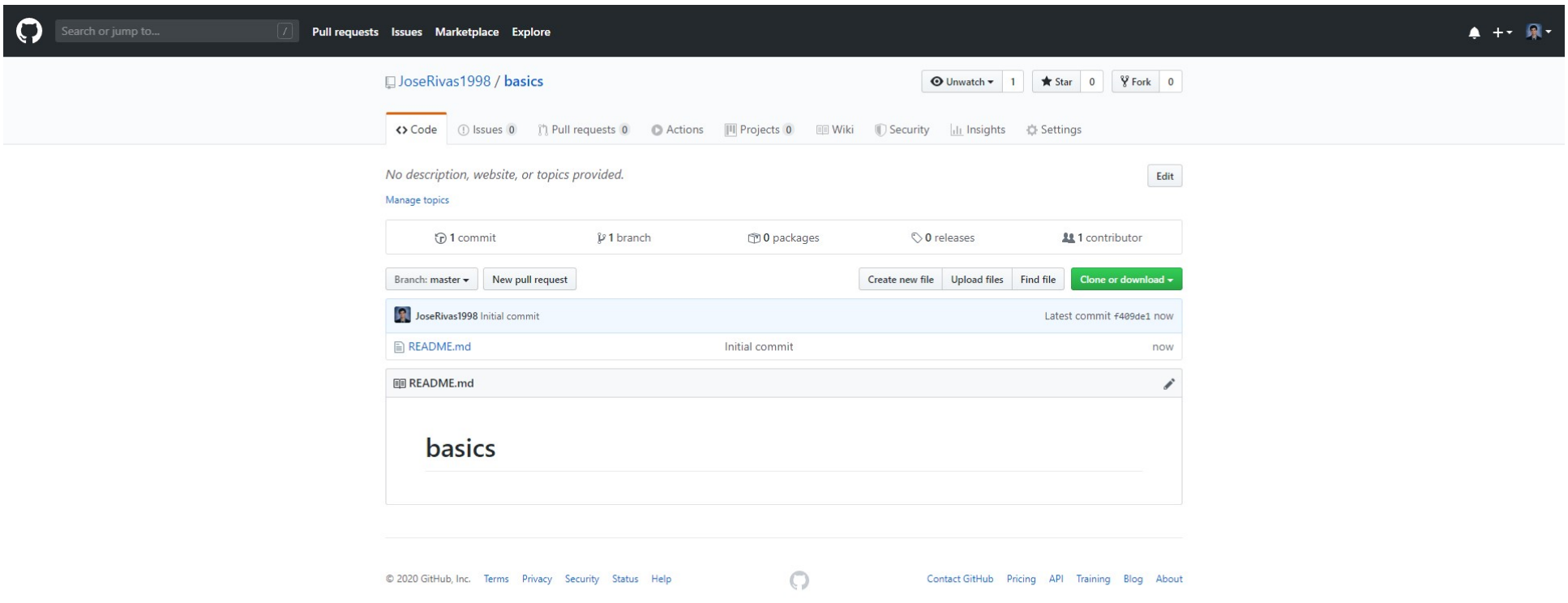
Then click the "New Repository" option.



Now add the details, here I will check off the option to create a `README.md` file.



Now my repository has been created:



Now, what I want to do is publish the repository I have been working on to this new one I just created. To do this, I need to create a **remote**, which is a reference to an external Git repository. This can be a Git repository on a different computer, or hosted on GitHub. To create a remote, we will use the following command:

```
git remote add origin https://github.com/JoseRivas1998/basics.git
```

Here, I created a remote called `origin`, which is the common name for remotes on GitHub. Now I want to actually upload my changes to GitHub using the following command:

```
git push origin master
```

You will be prompted to log into GitHub, go ahead and enter your credentials. The two parameters for the `git push` command are the remote name, and then the branch on the remote I want to post my changes to. However, I get an error:

```
To https://github.com/JoseRivas1998/basics.git
 ! [rejected]        master -> master (fetch first)
error: failed to push some refs to 'https://github.com/JoseRivas1998/basics.git'
hint: Updates were rejected because the remote contains work that you do
hint: not have locally. This is usually caused by another repository pushing
hint: to the same ref. You may want to first integrate the remote changes
hint: (e.g., 'git pull ...') before pushing again.
hint: See the 'Note about fast-forwards' in 'git push --help' for details.
```

This is because there was a commit on the remote that is not in the local repository. To fix this, I must first download the changes made on the remote. I can do this with the following command:

```
git pull origin master
```

Though this will normally work, but there are conflicting commit histories I need to include the following options:

```
git pull --allow-unrelated-histories origin master
```

This gave me the following output:

```
From https://github.com/JoseRivas1998/basics
 * branch            master     -> FETCH_HEAD
Merge made by the 'recursive' strategy.
 README.md | 1 +
 1 file changed, 1 insertion(+)
 create mode 100644 README.md
```

Now running `git push origin master`, I get the following output:

```
Enumerating objects: 37, done.
Counting objects: 100% (37/37), done.
Delta compression using up to 16 threads
Compressing objects: 100% (31/31), done.
Writing objects: 100% (36/36), 3.36 KiB | 1.68 MiB/s, done.
Total 36 (delta 16), reused 0 (delta 0)
remote: Resolving deltas: 100% (16/16), done.
To https://github.com/JoseRivas1998/basics.git
   f409de1..400ea2c  master -> master
```

Now, you can see that on GitHub, only the `master` branch is on the remote. I could also checkout `some-branch` and push it with the following command:

```
git push origin some-branch
```

Since `some-branch` did not exist on the remote, the branch will be created:

```
Total 0 (delta 0), reused 0 (delta 0)
remote:
remote: Create a pull request for 'some-branch' on GitHub by visiting:
remote:      https://github.com/JoseRivas1998/basics/pull/new/some-branch
remote:
To https://github.com/JoseRivas1998/basics.git
 * [new branch]      some-branch -> some-branch
```

A couple notes before moving on:

- Commit Early and Often: You do not need to necessarily push always, but it's best to have a large amount of small commits than a small amount of large commits.

- Pushing and pulling from a remote behaves like a merge, so conflicts are very possible.
- To help avoid conflicts, you can use the following order when you are working:
    1. Pull from remote
    2. Resolve conflicts
    3. Commit to local repository
    4. Pull again
    5. Resolved conflicts
    6. Push to remote

  Following these steps in this order will prevent messy merge conflicts between developers. Merging code with the remote is the job of the person writing the code.

Editing an Existing GitHub Repository

Suppose you found a repository on GitHub that you would like to contribute to. You could, of course, create a blank repository on your local machine, add the remote and then pull. However there is a better way. Use the following command:

```
git clone [url]
```

This will create a new folder with the name of the repository, with the remote set up to the repository at `[url]`, and pull the the master branch. If you would like to download all the branches, you can run the following commands:

```
git fetch --all
```

Now, if you checkout the missing branch, they will be automatically be pulled.

Now you can updated the codebase and push your changes.

And those are the basics! Congrats, you are now ready to work with Git! Git is great to use for projects with other developers, or on your own personal projects. I strongly recommend using it as much as you can, because recruiters and hiring managers often use it when looking at potential employees. It is also a great place to backup all your code.