Github and You

Ryan Moffitt
rmoff938@students.bju.edu

October 17 2024

Abstract
Do later

1. Git Basics

1.1. Termenology

Version Control System: Github uses and is a version control system. Meaning that everytime a new version of the project is uploaded to github it makes a *snapshot* of the project in its last state. This allows time travel to any point in the project where a new feature was added.

Commit: Adding new changes to the staged section of a project to be pushed later.

Remote Vs. Local: With version control systems there are often two versions of a project. The local version is the version and changes that exist on the users computer. The remote version is the hosted on the github servers and serves as the source of truth for the project that everyone references.

1.2. Repositories

Repositories are the folder-like structure that projects are stored in github. Repositories will also constantly be refered to as repos. Each repo will have its own project history stored its commit log, accessed by running `git log`.

1.3. Structure

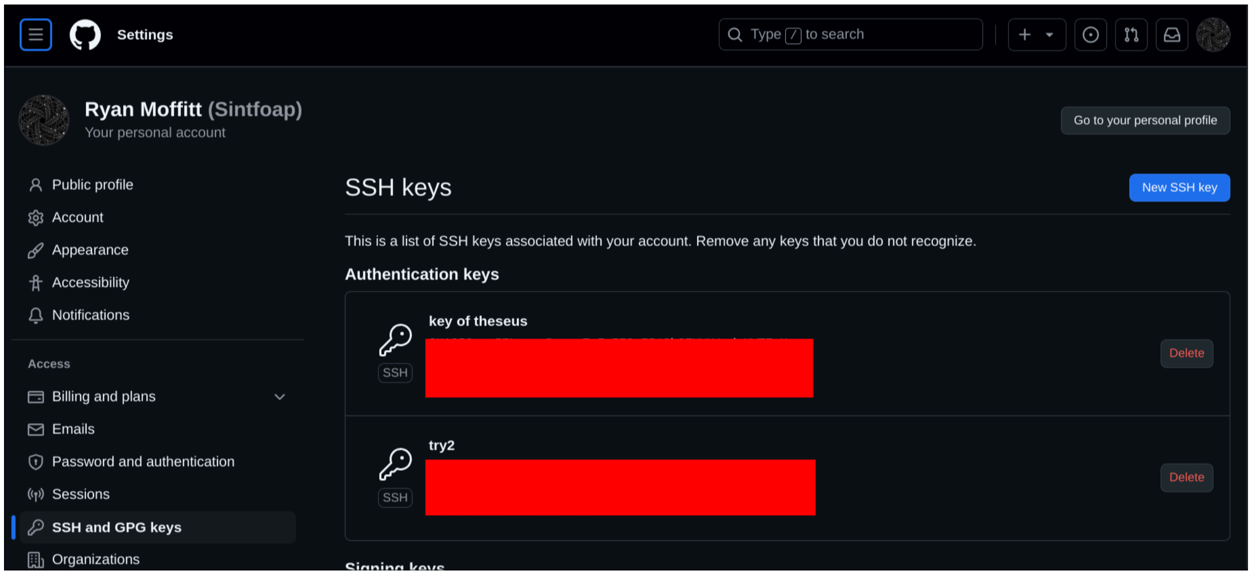
Each repository's structure will differ depending on what the project it's storing is, but they all will have some files in common. First off, they will all have a **`README.md`**. The contents of this file will be explained more further down, but in essence it holds the abstract of the project, installation instructions, and proper usage. Another file every repo will have is the **`.gitignore`** file. This is a special file that tells github which files to ignore when looking at changes to a file. It can also hold generalized rules for ignoring file types, for example one rule might be to ignore uploading all pdfs for a project that generates pdf outputs.

2. Github Authentication

Creating a github account is easy enough to do, but githubs authentication system using those credentials can be a bit interesting. There are two ways to authenticate machines with the github servers: Authentication tokens and ssh keys. However, authentication tokens are finicky at best, so this paper will document how to setup using ssh keys. Highlighted below are the ways of setting up credentails on the three main operating systems.

In the terminal run "`ssh-keygen -t ed25519 -C "your_email@example.com"`" where the email is the email you registered your github account with. This generates an ssh key that github will use to authenticate your account with everytime you attempt to push or pull from a repository. When prompted to enter the file in which to save your key hit enter. When prompted for a passphrase enter an easily memorable passphrase.

The ssh key is stored at "`C:\Users\{username}\.ssh\`". Navigate there and copy the value in id_ed25519.pub. It should look similar to "ssh-ed25519 AAAAC3NzaC1lZDI2NTE5AAAAIMmfODTLc8XR2G/aDkIX2Q8lVdjGBcZbdH/M+I4n5Q8Q " but of course with a different hash value. After copying this, navigate to *https://github.com/settings/keys* . The menu should look similar to Figure 1.



This is a list of SSH keys associated with your account. Remove any keys that you do not recognize.

[redacted]

[redacted]

Figure 1: SSH Key Menu

Select New SSH Key and enter a meaningful title into the the title box. Then paste the value that you copied from the id_pub file into the key box. Add the ssh key and you're set.

The above instructions are for Windows. If you have a mac or linux computer the only thing that changes is the location where the ssh keys are stored by default, which is in the " /.ssh/" directory.

3. Initialization

With some terminology under the belt, the programmer can start looking at how to set up a project.

3.1. Creating

There are two ways to create a git repository. The first and easier way to initialize a repository is to navigate to *`https://github.com`* and after logging in navigate to the user menu under `Your repositories`:

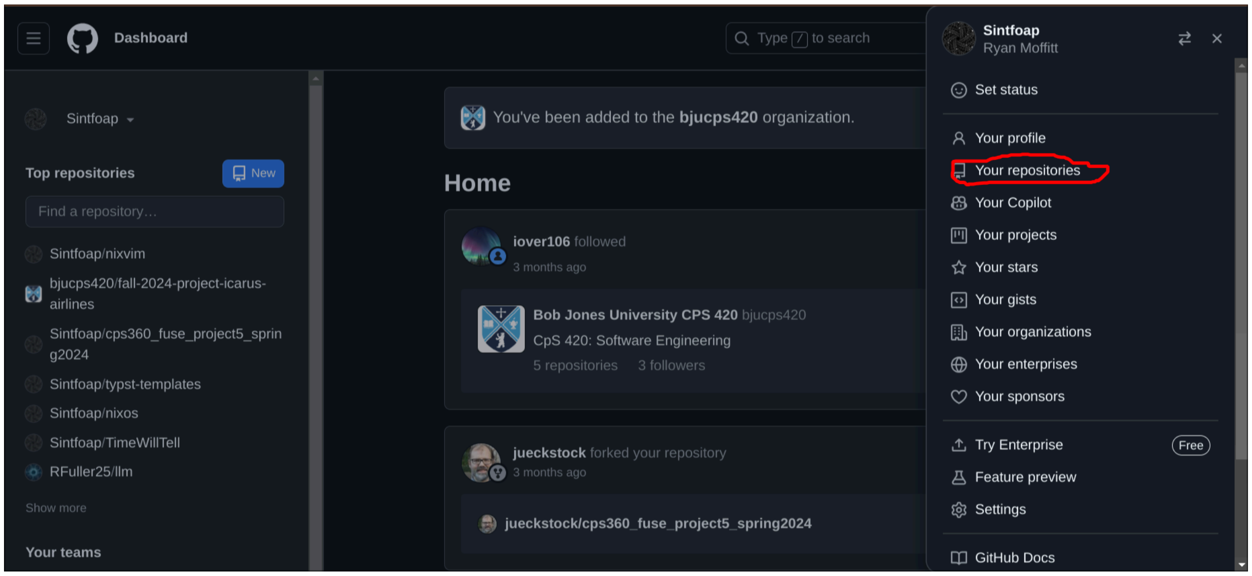


Figure 2: Navigation to Repository Menu

Inside repositories, there is a button for creating a new repository.

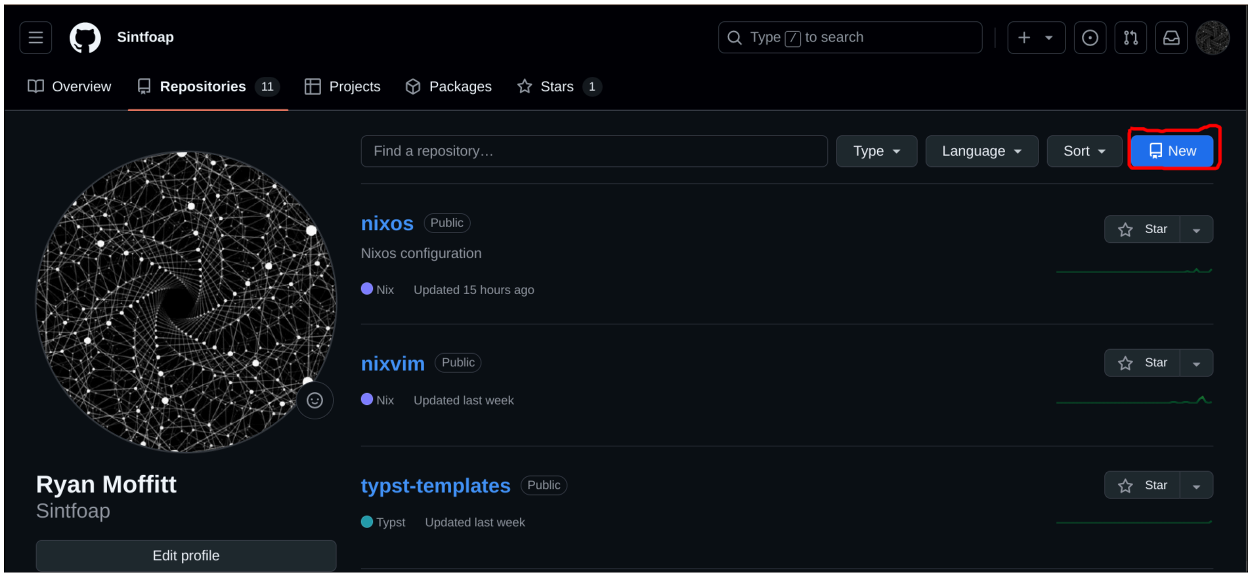


Figure 3: Repository menu

Inside that menu there is a form that contains all the options for configuring the new repository:

- Repository template
  - Generic repository configurations
  - Leave on no template
- Owner (See figure 3)
  - Choose the owner to be your username
  - Input a repository name in the following box
- Description
  - A meaningful introduction to the project and what it does
- Add a README file
  - More on the README later. Check this box.

- Add .gitignore
  - Leave on none for customized .gitignore later
- Choose license
  - Leave on none

The second way to create a repository is more of a derivative of the first way. Using the command line, run:

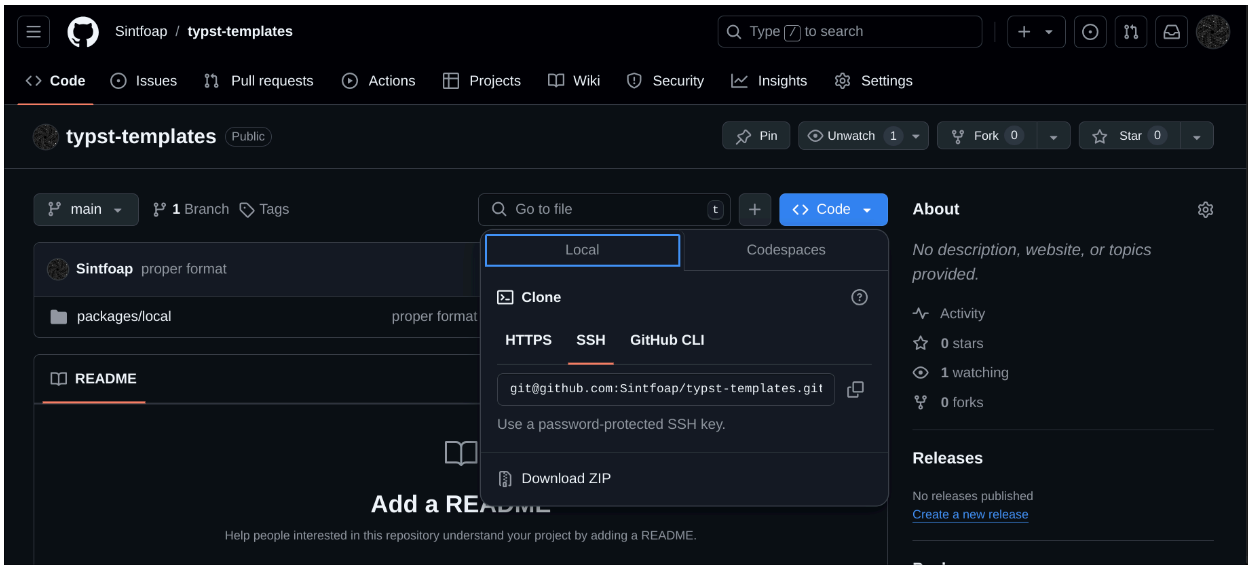
```
git init {foldername}
```

This turns {foldername} into a git repository, and after that it will need to be attached to a remote repository for collaboration and data loss prevention reasons. You can do this by using the following command:

```
git add remote origin {github repository url}
```

3.2. Cloning

After creating a repository, the next step is to download a version of the repository to the programmers machine. This is what we call **pulling down a repository**. To do this, go to the desired repository on github.com. Under the "Clone" Button, copy the ssh url (see Figure 4).



```
git@github.com:Sintfoap/typst-templates.git
```

Use a password-protected SSH key.

Help people interested in this repository understand your project by adding a README.

No description, website, or topics provided.

Figure 4: Clone Repo

After copying that, open a terminal and go to the directory that you wish to place the repository under. Then run the following command:

```
git clone {url}
```

This will clone the repository to a local version of it in the repository indicated.

3.3. Pulling

When another user has made changes to a repository and pushed them to the remote repository, you'll need a way to pull those changes down to your working version of the repository. To do this, run the following instruction:

```
git pull
```

Later we'll discuss some of the unexpected side effects of this operation.

4. Modification

After changing a repository we need to update the remote version with our local changes. The following workflow walks through that process. To see what needs added run `git status` for a list of filenames.

4.1. Staging

The first item in this process is to get the changes into git. Do this by running the following command:

```
git add {options}
```

The options here can be a couple of things. It can be a specific filename, or a -A to add all of the changes, or a -p to cherry-pick through changes and add some while ignoring others. After you've done this, if you want to revert some of those changes that you accidentally staged, then run the following command:

```
git restore {filename}
```

4.2. Committal

After staging the data, the user should then prep the changes in packets to be pushed to the remote version of the project. The packets that are prepared and then pushed are called commits. A couple attributes of commits are the commit ID, which can be used to revert back to a commit or reference it at any point in time, the commit message, which is a terse description of what was changed in the commit, and the commit diff, which shows the changes that actually happened in the commit. The most common way of creating a commit is as follows:

```
git commit -m "{message}"
```

The message should be a sentance or two that outlines what was changed in the commit proper. For example, if the project was a todo application and the commit added some functionality for creating notifications a message could be, "Add functionality for sending push notifications." Commit messages should always be in the imperitive and should stay under 120 characters by convention.

4.3. Pushing

Now that the changes are packaged and ready to go we need to actually push those changes up to the remote repository. In order to make sure that you don't overwrite changes someone else made to the remote repository, you should first run:

```
git pull
```

And then, assuming there are no merge conflicts, which if there are we discuss how to fix them later, run:

```
git push
```

And you should be done!

5. Branches

In a project where there are a many users modifying the same project at once, it is necessary to compartmentalize the workflow in order to have as little overlap in modifications as possible. The avenue by which we do that is called branching.

5.1. Definition

Branches are a functionality in version control systems that allow two independant paths of development. If this operation did not exist, then if two programmers were working on the same project at the same time, one programmer might push a change to a file that he changed, say example.txt. Then when the other programmer who modified the same file in a different way attempted to push his changes he would be at risk of overwriting the changes of the previous programmer. Now branches by themselves don't get rid of this issue, but they allow mitigation through abstraction and workflow for merging branches. This is all possible using a single branch workflow, but using multiple branches makes it more intuitive. To see all the branches for a project, run

```
git branch
```

This will show a list of **local** branches. This doesn't include the branches that have been made by someone else and pushed to the remote repository. If you perhaps want to see one of those to pull it down and do some work on something that someone else has started, you will rather have to run:

```
git branch -r
```

The -r flag represents setting the remote flag to true, making git show both local and remote branches. The remote branches will most likely be of the patter "origin/{branchname}"

5.2. Switching

There are multiple ways to switch and create branches depending on where you might be through your workflow. The base case is where you're on the main branch, conventionally called "main", or "master". The first way of creating a branch for this case is to use the branch instruction for git:

```
git branch {new-branch-name}
```

Where {new-branch-name} is whatever you wish to call your branch, conventinally written in kebab case and briefly outlining what you're planning to introduce in the branch. For example, bringing back the todo app mentioned above, you might want to fix an issue where some todo notifications aren't showing up because of some mistake programmed in by another person. In that case you might call your branch "fix-notification-bug". Or if you wanted to introduce a feature that auto-completes todos past their completion date, you might call it "add-autocomplete".

Now the above method only *creates* the branch for development, it doesn't actually *switch* you to the branch itself. To do that you can either run:

```
git checkout {branch-name}
```

or:

```
git switch {branch-name}
```

Both in essence do the same thing, but they have specific cases that they might be used for as shown later on. You might think that it's a lot of work to have to create a branch and then switch to it, and might think it a bit odd that you wouldn't just automatically switch to it since you can assume that you'd begin working on a branch you just created, and you'd be right. That's why the following command exists:

```
git checkout -b {new-branch-name}
```

This is semantically the same as calling:

```
git branch {new-branch-name}
```

```
git checkout {new-branch-name}
```

just in one step.

Now there is a specific case when you have accidentally developed on your local version of main when you were supposed to be developing on a branch off of main, say test-branch. If you commit before you switch to test-branch, then you've got problems. If you haven't committed it's fine to go ahead and run checkout -b {test-branch} and then commit, but if you already committed you can't create a new branch in that way. Thankfully, there's another command for that case:

```
git switch -c {new-branch-name}
```

which both creates a new branch and moves your local commits to it. You may have to revert main back to its current version, but we'll discuss that later.

5.3. Merging

When it's time to update the main project with the features or fixes you added in your branch, it's not quite as simple as just running git push. If you were to do that, it would push to the remote version of your local branch, creating a parallel branch in the remote next to main, which there are cases for doing. For example, if you added some changes in your branch and need to hand off development to another person then you'd want to push your branch up to the remote so that they can pull it to a local version on their machine. There is another step to doing this as well though. The remote repository currently knows nothing of your local branch, so you can't just push to a branch that doesn't exist. To remidiate this you can run the following command:

```
git push -u origin {branch-name}
```

This tells git to set an upstream version of the branch based off of origin, which signifies our remote. This in essence boils down to generating a remote point to base our local branch off of that everyone else can see as well.

Now when you want to merge your branch into main, you first should pull main down to your local version of main to make sure you've got the current updates. Then you'll run a merge command on main and your branch and deal with any of the issues that come out of that. We'll discuss merge conflicts and other issues in the next section. After that you can go ahead and push your local main into the remote main, solidifying your changes in the project itself for everyone. Let's walk through an example of this.

In our example, we'll have two branches:
- main
- test-branch <-

We're currently on test-branch and our main hasn't been updated with the remote changes yet, so we're first going to have to switch over to main to get those changes pulled down. After making sure that you've committed your changes on test-branch, run:

```
git checkout main
```

which updates our example to look like so:

- main <-
- test-branch

Then to get the changes run:

```
git pull
```

Which gets all the changes from the remote version of main. Then we want to merge our changes from test-branch into main, so we call:

```
git merge test-branch
```

which merges test-branch into main. Assuming there are no issues, which again we'll handle issues in the next section, then run:

```
git push
```

pushing main to the remote repository and finishing our workflow.

6. Conflicts

When dealing with a multi-user project, there are a number of cases in which we have the possibility of one user overwriting another users changes. To try and avoid that, git has implemented a couple detection and issue avoidance features that may crop up from time to time. In this section we'll outline how to deal with those.

6.1. Conflict Cases

In the section above I outlined a case where two users updated the same section of code and then tried to push them to main. In a naive solution the user who pushed last's changes would be kept while the previous users changes would be discarded. However, git has a bit more of an elegant solution for that. This solution is called merge-conflicts, and can be pretty daunting.

Let's look at the example in more depth. In the step where you attempt to merge test-branch into main in the above example, if there are conflicting changes git will tell you that and you'll have to resolve them before you can finish merging and push to the origin/main. The conflicts will look like the one below:

INSERT PICTURE OF MERGE CONFLICT

As you can see there are two versions of the same code with slight differences. The first is the version that existed in the remote, and the second is the version that existed in your local branch. Occasionally you'll just want to keep one or the other, but most of the time it's a combination of the two changes. To see all the conflicts in the code when you're in merging mode just run git status and it will show you all the files that have conflicts in them.

INSERT PICTURE OF MERGE CONFLICT GIT STATUS

6.2. Resolutions

When making resolutions to merge conflicts its important that you be respectful of other users code. Do not ever just accept your changes without thinking through the ramifications of deleting the other users progress. Most of the time it is acceptable and reccomended that you bring in other users that wrote the code to look at doing a merge between the two branches to make sure that all opinions and changes are brought into consideration. There are few worse feelings in development than introducing a bug that could've been avoided if someone had talked to another developer about what their code was doing rather than just assuming it's old and deprecated.

After you've fixed all the merge conflicts, you'll have to add all of the updated files to staging again and commit them again. The message should be something like "merging {branchname} into main" or something similar. After that you will be clear to push to main like normal.

7. Reverting

Occasionally someone will introduce a change that either creates a bug or doesn't really need to be there. In that case it's nice to be able to go backwards in the commit history to a certain point in time. There's also the case when you've accidentally committed on the wrong branch and want to set it back to it's base. Both of these are easily delt with using built in git capabilities.

7.1. Hard Reset

So your branch is royally messed up. We've all been there. The best case here is to reset your branch to before you started working on your current feature, back at the HEAD ref. The HEAD ref is a reference order pointer to the base of your current feature branch, where it diverged from the main production branch. This gives you the option to always have a point to reset to when things inevetably go pear shaped. The two ways of reseting to the HEAD ref are the following commands:

```
git reset
```

```
git revert HEAD
```

'git reset' is shorthand for 'git revert' under the hood, but both of those will put your branch back into, hopefully, the last working position.

7.2. Commit Based

If you need to only jump back a commit or two, the procedure is a bit longer, but still pretty straight forward. First run `git log` in the directory that you wish to revert to a previous stage. The output should look something like this:

8. Documentation

8.1. README

8.2. Wiki