MIDDLESEX Community College

Tools and Technologies for Tech Writers 2022

Week 1

Using Git

Notices

This document was prepared as a handout for the Middlesex Community College Tools and Technologies for Technical Writers class, Winter semester 2022.

Prepared by Zoë Lawson, course instructor.

Contents

Brief introduction to Git

There are many books and websites available that can explain more about Git than I can. This is a brief explanation to get you started.

Git is a very popular source control system. Personally, I don't really like it. However, it's currently very "in" and therefore you should know how to use it.

I learned to tolerate Git after reading the following:

> *[Linus] Torvalds sarcastically quipped about the name git (which means unpleasant person in British English slang): "I'm an egotistical bastard, and I name all my projects after myself. First 'Linux', now 'git'." The man page describes Git as "the stupid content tracker". The read-me file of the source code elaborates further:*
>
> *The name "git" was given by Linus Torvalds when he wrote the very first version. He described the tool as "the stupid content tracker" and the name as (depending on your way):*
>
> * *random three-letter combination that is pronounceable, and not actually used by any common UNIX command. The fact that it is a mispronunciation of "get" may or may not be relevant.*
> * *stupid. contemptible and despicable. simple. Take your pick from the dictionary of slang.*
> * *"global information tracker": you're in a good mood, and it actually works for you. Angels sing, and a light suddenly fills the room.*
> * *"goddamn idiotic truckload of sh*t": when it breaks* [1]

I generally think of it as a truckload...that said, Git is very powerful and worth using when it's all you have. A lot of the time, tech writers get to use whatever development is using. That way tech writers (or the IT group) don't have to maintain (or pay for) another source/version control tool or content management system.

Advantages:

* Any source control system is better than no source control system.
* There should be a developer or two available who can help you out when you get stuck.
* The ability to easily make branches to work on revisions can be used to great good.

Disadvantages:

* A 'collaborative' type of control system, so possible to clobber (overwrite) other's work if you're not aware of what you're doing.
* Sometimes the automatic merge undoes work.
* No easy way to undo a specific change or revert a specific file.

---

[1] https://en.wikipedia.org/wiki/Git

Major concepts

These are my simplified understanding of the major components that make up Git and how they interact.

Repository

A repository is the major container. In general, each project should be in one repository. Most companies have several repositories, one repository for each product.

Repositories contain multiple *branches*. (Branches get their own description later.)

You can have a Git repository (also called a "repo") on your laptop. You can commit files to it and have a version history on your laptop. This is great because you get the ability to see the history of changes to files. However, it's just on your laptop.

Most companies/projects have some Git Server, dedicated to storing everything and so all the people working on project can access it. While Git is open source, there are many companies out there that run and maintain additional wrappers around Git. BitBucket and GitLab are products you can host. There are also many cloud solutions such as GitHub (and a web version of BitBucket).

Compared to your laptop, this great Git server is the *remote* repository. It's also considered the source or *origin* that people work from. You *clone* the repository on your laptop. Now you have a copy of the repository on your laptop. You can work on it and make changes. When you're happy with your changes, you push your changes to the remote repository so other people can get them.

This explanation is grossly simplified, but you need to understand the idea of a remote repository (where others can get the files, too) vs your local repository (where only you have access).

Branch

When you first create a repository, you have one branch, *main* or *master*[2]. This is a very important branch. Main is generally the set of files that equates to whatever is going to be released.

However, you don't want folks updating master willy-nilly. Someone might be trying to make a fix that breaks other people's work. Or you want to start working on something for a release that won't go out for a while. So, you make a branch. Initially, a branch is a copy of main. You make changes on the branch. When you're happy with the changes, you then merge the changes from the branch back into main.

When to branch, why to branch, branching naming conventions are vitally important. There are many different theories and methods. What works best depends on your environment and other processes. Common theories are:

• Have a different branch for each released version of the software.
• Create a new development branch for each new feature.
• Create a new development branch for each new fix.

---

[2] Many companies are now attempting to use more diverse and inclusive language. Some terms, such as "master" or "execute" can be offensive or triggering. Companies are trying to clean up their terminology to use more inclusive terms. The old original branch name was "master", but in the past few years it has been renamed to "main". Examples and documentation may not reflect this change.

For example, you are assigned to work on a new feature. You branch from main and work in your feature branch. When your code is working, you merge your feature branch back into main.

Again, this is greatly simplified. You generally are working in an environment with many, many branches. You can merge changes from any branch to any branch, and you often have to.

While you're working on your feature, someone else finishes their feature and merges it into main. You want to get their changes into your branch so you can make sure your new feature works with their new feature. So you are constantly merging changes from main into your branch.

Now, your feature may not actually work well with the other new feature. You could try a fix, merge it to main, have it not work, and keep repeating that process, breaking other people's work in the meantime. Or, you could choose to merge from another feature branch into your branch and work out all the issues just between your two branches before merging into main.

Merging is the wrong term. You *push* to and *pull* from various branches and repositories. More on these terms later.

Major actions

There are dozens of commands in Git and they are all incredibly useful when you need them. However, there are a few commands you should just know.

All examples are done with the command line as those are the actual terms used with git. However, your git client may use different terminology, or combine actions. For example, GitHub Desktop combines the `add` and `commit` actions.

```
Content formatted like this is the command you enter.
```

ItalicizedText is a variable that you must provide depending on what you're actually doing. For example for *branchname* you would use `Winter_2022`.

```
Content formatted like this is the result of the command.
```

Clone

The clone command is how you get a copy of a repository on your system.

```
git clone https://github.com/ZoeLawson/mcc_prep.git
```

```
Cloning into 'mcc_prep'...
remote: Enumerating objects: 3, done.
remote: Counting objects: 100% (3/3), done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (3/3), done.
```

Checkout

This is a major misnomer in my book. The Git checkout command is how you switch between branches.

```
git checkout branchname
```

```
Switched to branch 'branchname'
```

Pull

After you have cloned a repo, you pull from a repository and a branch to get the latest and greatest changes. In general, you pull from the remote repository to your local repository.

A common command is to pull from master into your current local branch to make sure you have the latest files.

```
git pull origin Winter2022
```

The first time you do this, it may take a very long time. You are downloading all the files in the repository. It also includes a list of all the changed files, so it may scroll past.

```
remote: Enumerating objects: 11, done.
remote: Counting objects: 100% (11/11), done.
remote: Compressing objects: 100% (8/8), done.
remote: Total 11 (delta 2), reused 5 (delta 2), pack-reused 0
Unpacking objects: 100% (11/11), done.
From https://github.com/ZoeLawson/mcc_tools_tech
 * branch            Winter2022 -> FETCH_HEAD
   2119afd..7cf5490  Winter2022 -> origin/Winter2022
Updating 2119afd..7cf5490
Fast-forward
 Week01-IntroGitHub/Homework/ZoeLawson.txt | 1 +
 Week01-IntroGitHub/Homework/mcc_demo.txt  | 1 +
 2 files changed, 2 insertions(+)
 create mode 100644 Week01-IntroGitHub/Homework/mcc_demo.txt
```

Do not confuse this with a *pull request*.

There is also the command `fetch`. `fetch` is similar to `pull`, but there are differences. The most important difference is that fetch also retrieves new data, such as the existence of new branches.

Add

Every time you make a change, you need to add the change to Git. This stages the changes. This is something you do on your local repository only.

You always add your files, whether they are new files or changes to existing files.

To add a specific file:

```
git add path/filename
```

Use forward slashes, even on a Windows system.

To add everything in the current directory you're working in:

```
git add .
```

The command line doesn't give a response to adding files.

Status

A quick check of what state all the files are on your system. Useful to figure out what git thinks is going on.

```
git status
```

```
On branch Winter2022
Your branch is up to date with 'origin/Winter2022'.

Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        modified:   Week01-IntroGitHub/using_git.pdf

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   Week01-IntroGitHub/Homework/ZoeLawson.txt
```

Commit

After you stage your changes, you need to actually really, really save them or *commit* them. When you commit, you can (should) add a comment explaining what you've changed.

Tip: In the moment, commit messages seem stupid. All you want to write is "Fix typo". However, when something goes terribly wrong, the commit messages are your breadcrumbs. The more detailed you can be the better.

You can only push committed changes. It is possible to stage some changes and commit others, so you can get files up to the remote repository in pieces.

```
git commit -m "Intial draft of changes for new feature"
```

```
[Winter2022 4c3ad7b] Intial draft of changes for new feature
 1 file changed, 0 insertions(+), 0 deletions(-)
```

Push

Push is how you get your changes back to a remote repository. In general, you push to a branch on the remote repository.

Most people restrict who can push to important branches (such as master).

```
git push origin branchname
```

```
Enumerating objects: 7, done.
Counting objects: 100% (7/7), done.
Delta compression using up to 8 threads
Compressing objects: 100% (4/4), done.
Writing objects: 100% (4/4), 1.51 MiB | 1000.00 KiB/s, done.
Total 4 (delta 2), reused 0 (delta 0)
remote: Resolving deltas: 100% (2/2), completed with 2 local objects.
To https://github.com/ZoeLawson/mcc_tools_tech.git
   7cf5490..4c3ad7b  Winter2022 -> Winter2022
```

Pull request

When people restrict access to specific branches, instead of pushing to a restricted branch, you make a *pull request*. You are asking someone with permission to make changes to the restricted branch to pull the changes from your branch into that branch.

Pull requests generally also have some sort of review process. This is to attempt and make sure the changes you request don't break things.

In general, pull requests are best done using whatever mechanism your Git Server has. For example, BitBucket and GitHub both have some sort of **Create Pull Request** feature.

Sample day in the life

The details will always be specific to your development environment, but this is a common example.

Your environment has dozens of branches, but you only care about the following:

- main - The current main line of code, working towards the 3.3 release. This is a restricted branch.
- release_3.2 - The previous release of the software. This is a restricted branch.
- feature_12345 - The branch you're mostly working on. Its for a new feature for 3.3 and was originally branched from main. This is your branch. You have rights to push to it.

1. First thing in the morning, pull from main into your feature branch to make sure you have the latest and greatest.

```
git checkout feature_12345
git pull origin main
```

2. Work on your feature. You add and commit your changes and push them to the remote version of your branch as you go.

```
git add .
git commit -m "enabled the widget"
git push origin feature_12345
```

3. Run the build that makes the software from your feature branch. Discover your change didn't quite work. So you make more changes.

```
git add .
git commit -m "forgot to connect the sprocket to the widget"
git push origin feature_12345
```

4. Run the build and it's still not quite right. However, you get assigned a bug on the previous version of the software that you need to fix right now. So you make a new branch from the release_3.2 branch and work on the fix.

   a. Switch to the release_3.2 branch.

   ```
   git checkout release_3.2
   ```

   b. Get the latest and greatest from the release_3.2 branch.

   ```
   git pull origin release_3.2
   ```

   c. Make a new branch to work on.

   ```
   git checkout -b release_3.2_fix_bug678
   ```

   d. Make your changes. When you're happy with the fix, add and commit the files.

   ```
   git add .
   git commit -m "Fix for bug 678 - thingamajig does not budge"
   git push origin release_3.2_fix_bug678
   ```

   e. Amazingly enough, this one change fixes the issue. Make a pull request from the release_3.2_fix_bug678 branch into the releaes_3.2 branch.

5. Now get back to the feature work.

```
git checkout feature_12345
```

Push me Pull you

Understanding how local and remote repositories work with pushes and pulls takes a few tries to understand. Eventually, your environment will become habit.

## Working with a private git server

Most companies have some sort of private git server. This could be hosted within their own IT infrastructure, or a special cloud account, such as GitHub Enterprise. On a private git server, there is much more control over who can have access to it.

Pull from a remote repository master branch to get changes onto your local system.

Commit your changes to your local repository.

Push your changes to your branch on the remote repository.

Make a pull request to get your changes from your branch in the remote repository to master.
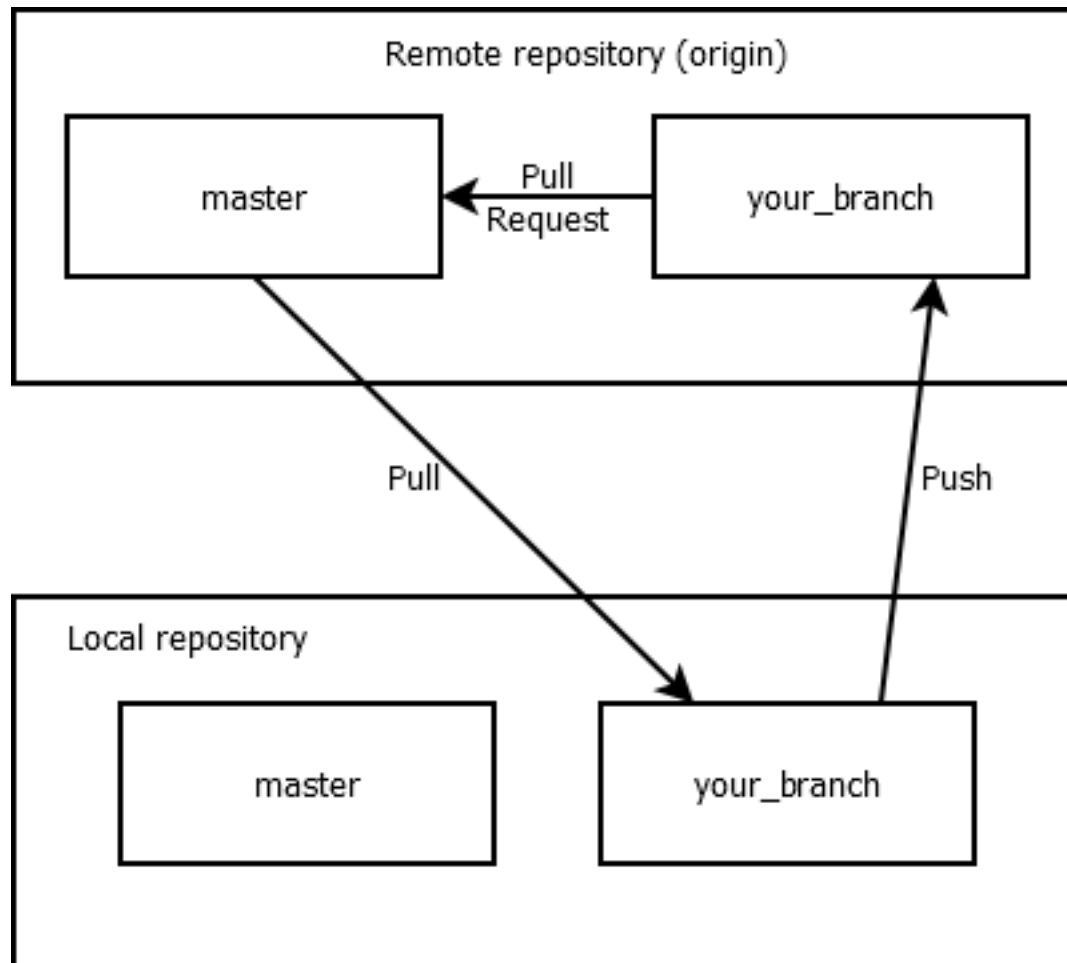
Working with GitHub

GitHub is a collection of public repositories. The intention is for anyone and everyone in the world to be able to easily access open source projects and contribute. For example, the DITA Open Toolkit is an open source project, available from https://github.com/dita-ot/dita-ot. Technically, I can make a copy of the project (called a *fork*) and make changes to it. I could fix a bug or develop a new feature. I then make a *pull request* to the owner of the DITA-OT project and ask them to accept my change.

(This is theoretical. There's a whole process to follow, but that's the general premise.)

Another good example is themes for Jenkins. Jenkins is an open source automation tool. People provide different themes or skins for it. If you look at https://github.com/jenkins-contrib-themes/jenkins-material-theme you can see it was forked from a different theme.

GitHub and forking adds an additional quirk to the process. You're going to have your local repository, your forked repository in your account of GitHub (remote or origin), and the original repository (upstream).
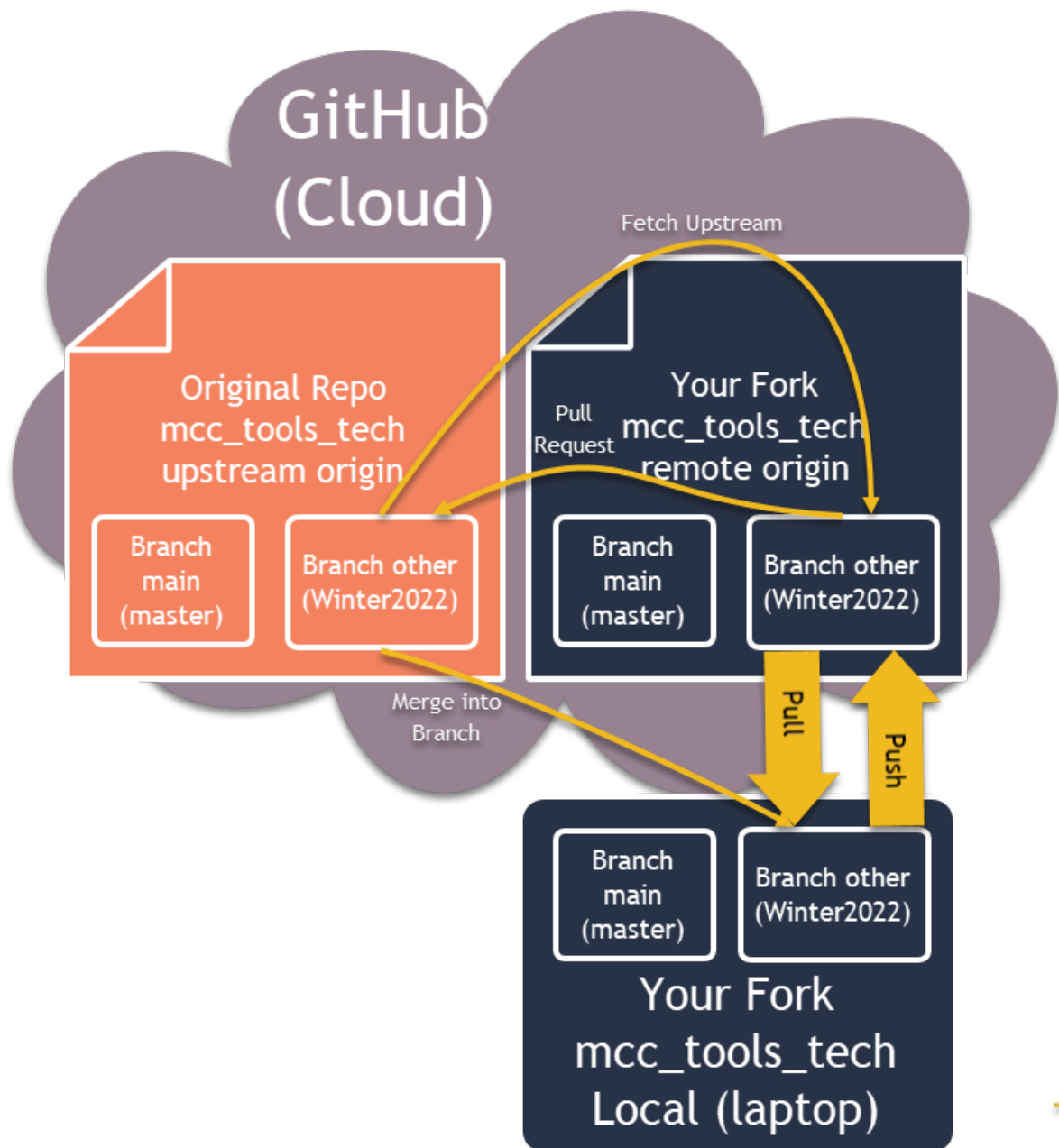
1. You fork the original repository.

   This creates a copy of the original repository in your GitHub account.

2. You clone your forked repository to your computer.

   This creates a copy of files on your computer.

3. You make changes on your local computer (your local repository is ahead of the remote repository) and *push* them to the remote repository (forked repository) in your GitHub account.

4. If changes are made to the files on your remote repository (your local repository is behind your remote repository), for example, if you use **fetch upstream**, you *pull* them into your local repository.

5. If your forked repository is *behind* the original repository, you can solve this two ways:

    •   In GitHub, you can use **fetch upstream** to get changes made to the original repository to your remote repository in GitHub. You then need to pull (or fetch) those changes from the remote repository to your local repository.

    •   In GitHub Desktop, you can use **merge into branch** to get changes made to the original repository to your local repository on your computer. You then need to push those changes to your remote repository in GitHub.

Working with GitHub

GitHub is a great big Git Server in the cloud. Most of the commands are the same, but there are a few different concepts.

GitHub is in the cloud and is designed around many people all around the world sharing content. To accommodate that idea, GitHub has an additional concept of a *fork*.

When working with normal Git, you clone a repository locally. But GitHub is where you need to store your repository (in the cloud). So with GitHub, you can fork someone else's repository into your account. This is basically a clone of a repository that stays in the GitHub cloud. A repository fork lives in your GitHub account/project and is still connected/associated with the original GitHub account/project. You can then clone the fork onto your laptop to actually work on it.

You can still have all the fun of dealing with different branches on top of all the different repositories. However, in this class, we will be working with one branch only. You will generally be pulling and pushing between different repositories, but only dealing with the Winter2022 branch.

Week1 homework

1. Make a GitHub account.
2. Fork the GitHub repository for this class.
3. Install GitHub Desktop on your laptop.
4. Clone the fork on to your laptop.
5. Make a text file in the Week 1 homework folder.
6. Commit the file to your local branch.
7. Push the file to your GitHub account.
8. Make a pull request to get the file into the class repository.

Make a GitHub account

To do anything besides download files from GitHub, you need an account.

To open a GitHub account, you need a valid, working email address. Consider the following as you set up the account:

*   GitHub uses this email address for various notifications. This includes potential conversations including "There's something wrong with your homework. You must fix X before I can merge your pull request." Make sure you pick a real account you have easy access to and check often.
*   GitHub is public. Potentially, anyone can see the email address, and definitely the user name you choose. Remember, you can use your GitHub repository as a basis for a public portfolio. Choose names you don't mind being associated with you in your professional career.

1. Go to https://github.com/ and make a new account.

   You want to make a free personal account.

   See https://docs.github.com/en/get-started/signing-up-for-github/signing-up-for-a-new-github-account if you have questions.

   GitHub is constantly updating and changing. How I made an account a few years ago will be slightly different now. Please use the GitHub documentation, which will be much more up to date than anything I can provide. However, I can try to help as much as I can.

2. Verify your email address with GitHub.

   See https://docs.github.com/en/get-started/signing-up-for-github/verifying-your-email-address for details.

Fork a GitHub repository

Returning to the GitHub on the web, fork the mcc_tools_tech repository into your account so you can make changes.

"Fork" is a concept in GitHub. It is not a basic command in Git. In GitHub, you may not know the owner of the content you want to work with. You may not be able to be a contributor to that repository. So you "fork" or make a copy of the repository in your GitHub account. You can then work in the fork, and make a pull request into the original repository.

Go to https://github.com/ZoeLawson/mcc_tools_tech and click **Fork**.
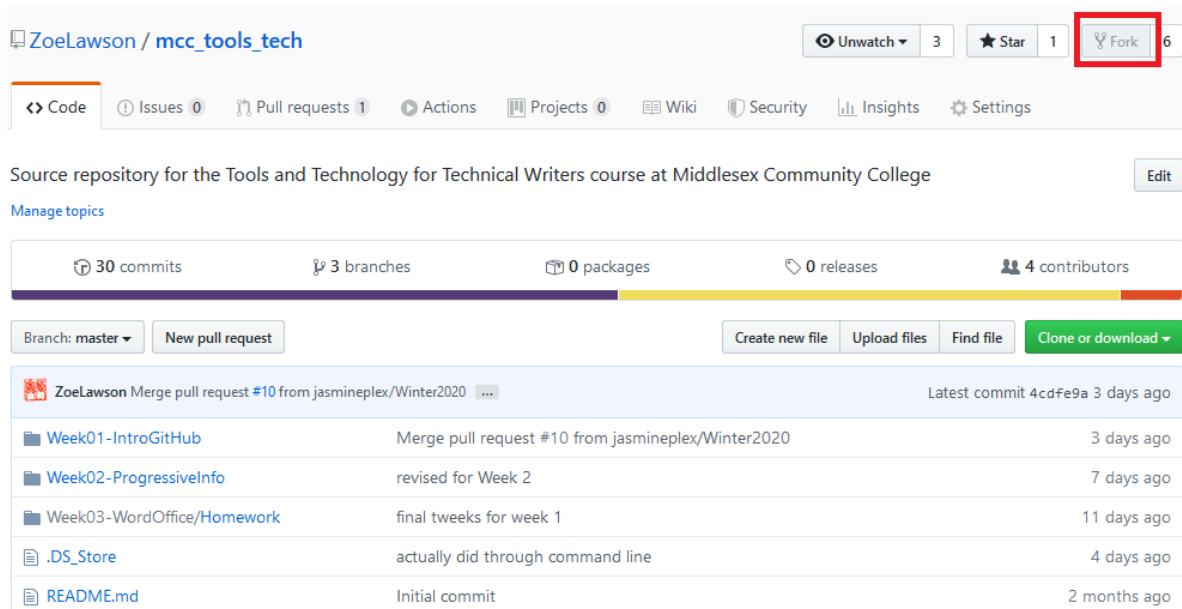
You have created a copy of the repository in your GitHub account. This copy is currently only available "in the cloud" and isn't on your computer yet.

Things to notice on this screen:

- The name of the repository is *Your_User_Name*/mcc_tools_tech. There is also the label explaining that it was forked from ZoeLawson/mcc_tools_tech.
- Notice what branch you're viewing. The default for this year is Winter2022.

Install GitHub Desktop

In order to use GitHub fully, you need a version of Git installed locally.

Git is a command line tool. It was originally written for the command line, and the command line gives you the most control. However, most people don't like using the command line, and there are some things (such as comparing files) where a GUI version is much nicer.

There are many, many different git clients out there. (See https://git-scm.com/downloads/guis for a list.) If you have to use one in a workplace, use the one your company recommends. For this class, I am arbitrarily picking GitHub Desktop.

1. Go to https://desktop.github.com/ and download version appropriate for your operating system.
2. Run the installer.

I'm partial to the command line, so all instructions are given using commands and GitHub Desktop. That way you can see both what the commands do and what the GUI is doing.

GUI versions are fabulous for comparing different versions of files. However, the GUI sometimes hides the Git behavior from you. This is occasionally problematic.

If you want to use the command line, download and install the command line tool from https://git-scm.com/downloads.

Initial GitHub Desktop setup

The first time you use GitHub Desktop there's some configuration work you need to do. You need to connect the GitHub Desktop to your GitHub account so it can download and upload files.

These screen shots were taken the end of December, 2021. I also use a "dark mode" as much as possible. Being a hosted, cloud application, the website and the GitHub Desktop may change at any time without warning. What you see on your computer may be a little different, and that's probably okay.

I've learned that with the speed that applications change, it's best to accept and adapt. You can complain about buttons being hard to find, but learning how to extrapolate is more useful. Everything updates automatically these days, your tools are going to change underneath you all the time.

1. After you install GitHub Desktop, it should start automatically.
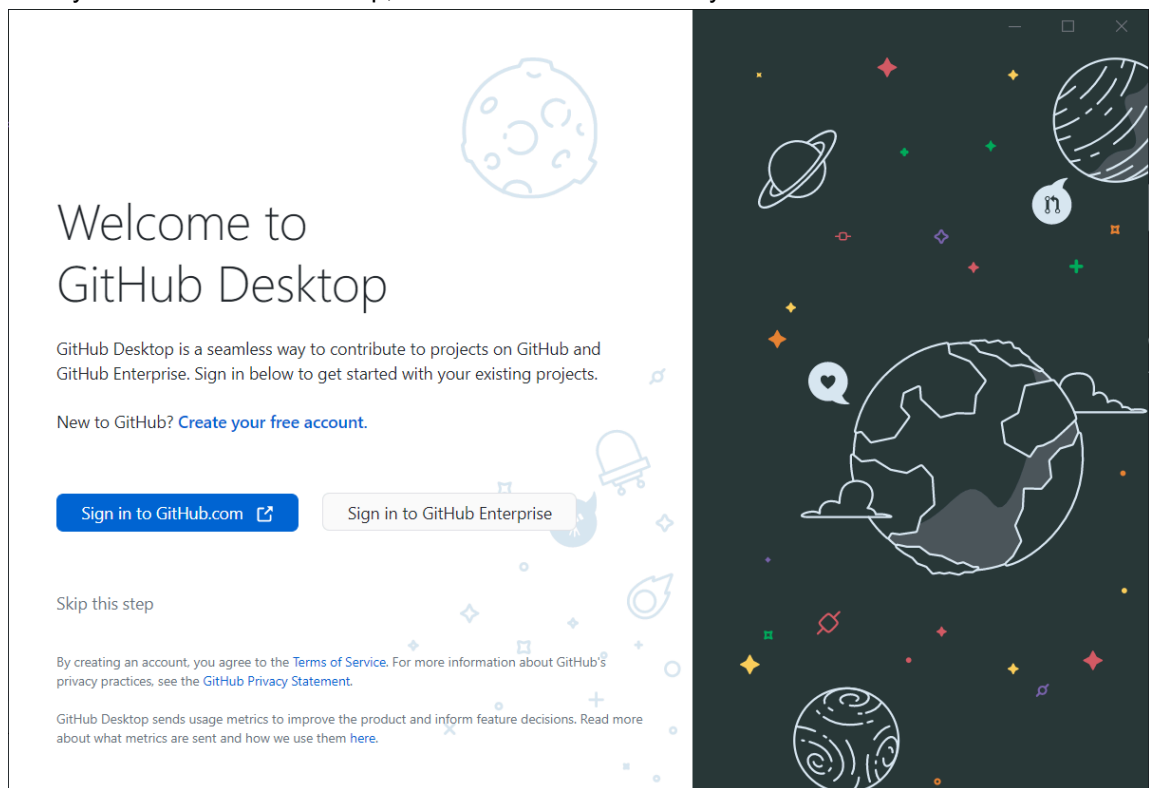
2. Click **Sign in to GitHub.com**.

   The system should switch to your browser and a sign in screen may open. If you're already signed in in an open browser, this may be skipped.
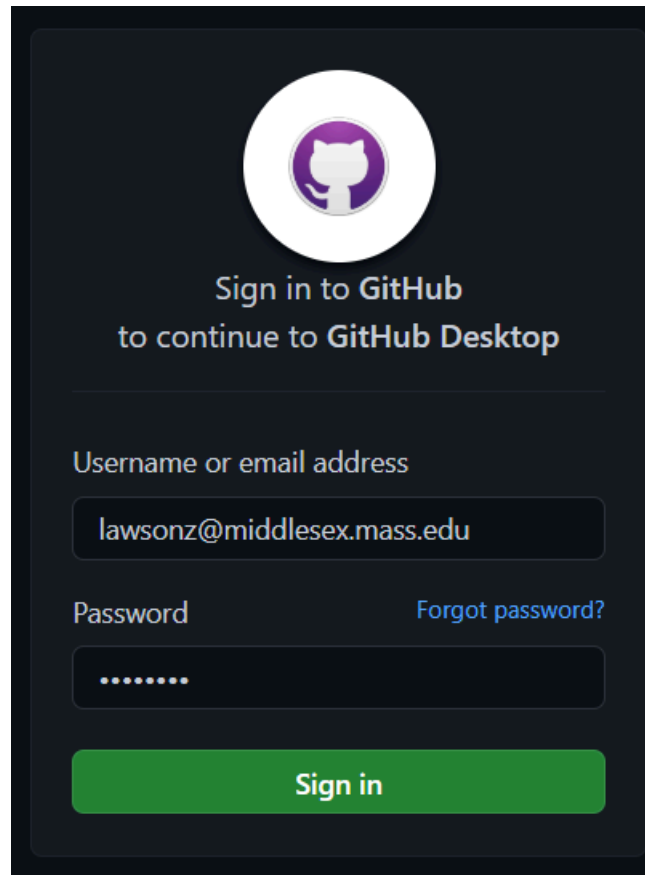

3. If prompted, enter your GitHub user name and password.

   After logging in to GitHub, you need to authorize GitHub Desktop to work with your GitHub account.

   Because there are lots of terrible people out there on the Internet, security and authentication is constantly changing. Two years ago, you did not need to perform this step. Next month, there could be a new security breach, zero-day attack or something else we haven't heard of yet that

may require everyone to change how you authenticate with an application. For example, you could enable two-factor authentication which is more secure, but more annoying to use.
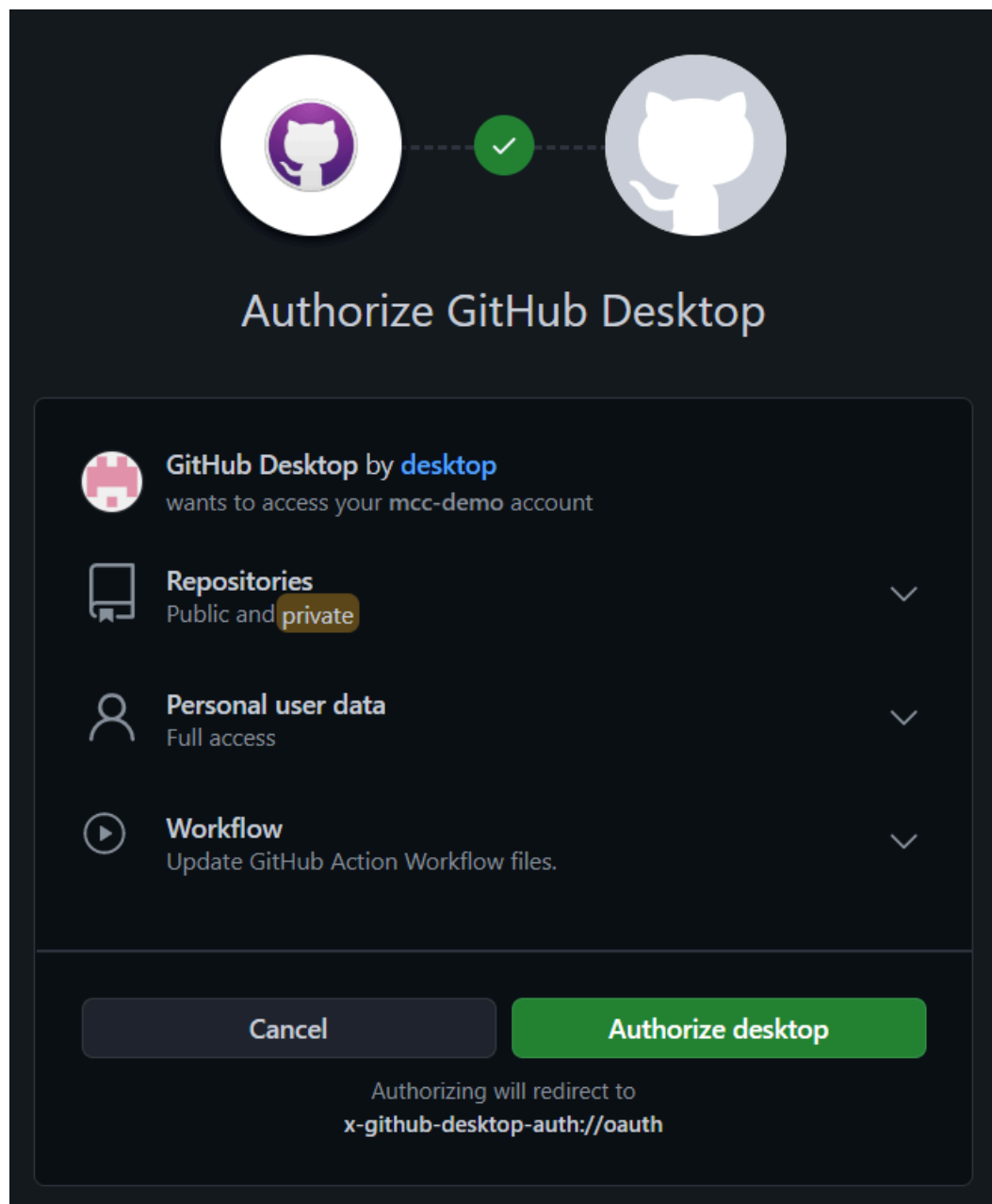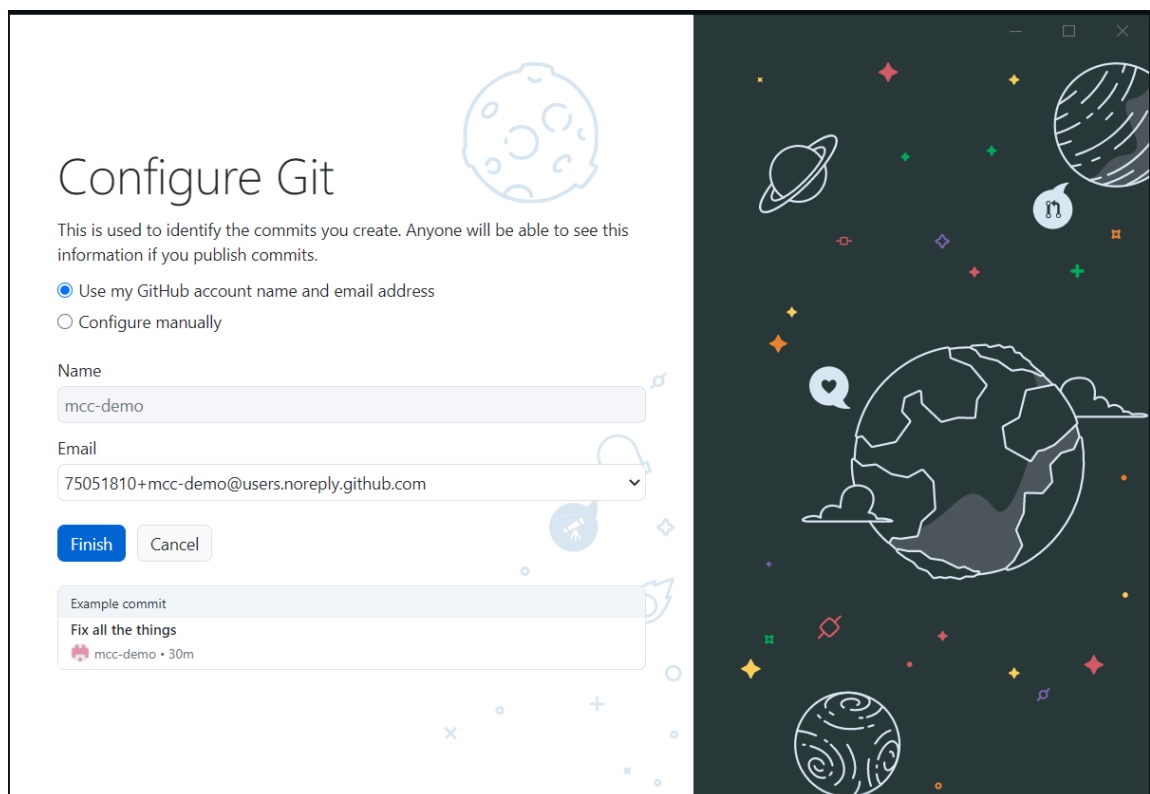
4. Accept the defaults and click **Authorize desktop**.

   These actions have happened in your browser. Now that you've approved the application, GitHub (cloud) needs to start working with GitHub Desktop. Most likely, your browser asks for your permission to open GitHub Desktop.
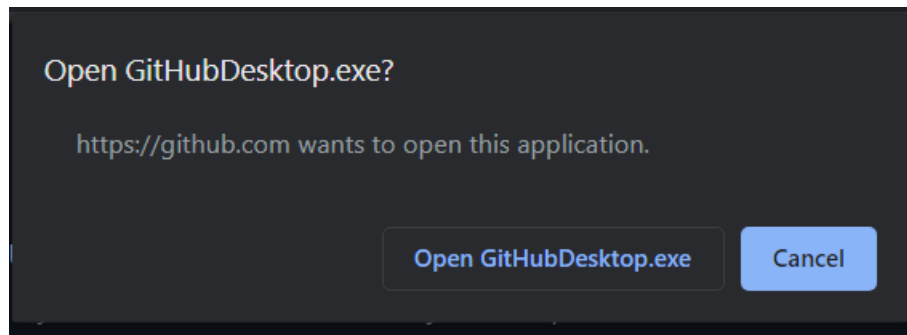
5. Click **Open GitHubDesktop.exe**.

   Your system should switch back to GitHub Desktop.

6. Accept the defaults (**Use my GitHub account name and email address**) or select **Configure manually** and provide a user name and email address, and click **Finish**.

   Remember, this user name is going to display everywhere in GitHub.

Congratulations! You have installed your first tool for the course! Now to get files from GitHub down to your computer by "cloning the repository".

# Clone your GitHub repository

It's great that you have made a copy of the repository in your account in GitHub, but to really work with the files, you need a local copy on your computer. Downloading the files to your system in a way that they remain connected is called *cloning* in git.

You can do some basic work in the GitHub UI. You can make a tweak to a text file. However, it's not a real editor. In general, it's always better to clone locally so you can really edit files.

There are actually two ways you can get the files onto your computer:

• Download Zip - This grabs a copy of all the files and you can download them as a zip file. Once you unzip the files, that's the copy of the files you have. You can't upload a change easily.
• Clone - This makes a copy of the files locally that's connected to the remote (also known as "origin") set of files. If you make a change to a file locally, it's "easy" to upload the change (push). If there are changes made remotely, you can "easily" get the changes (pull).

Because you're going to be making changes and uploading files, you want to clone the repository. Assuming you're coming from Initial GitHub Desktop setup on page 17, cloning your first repository should be easy.
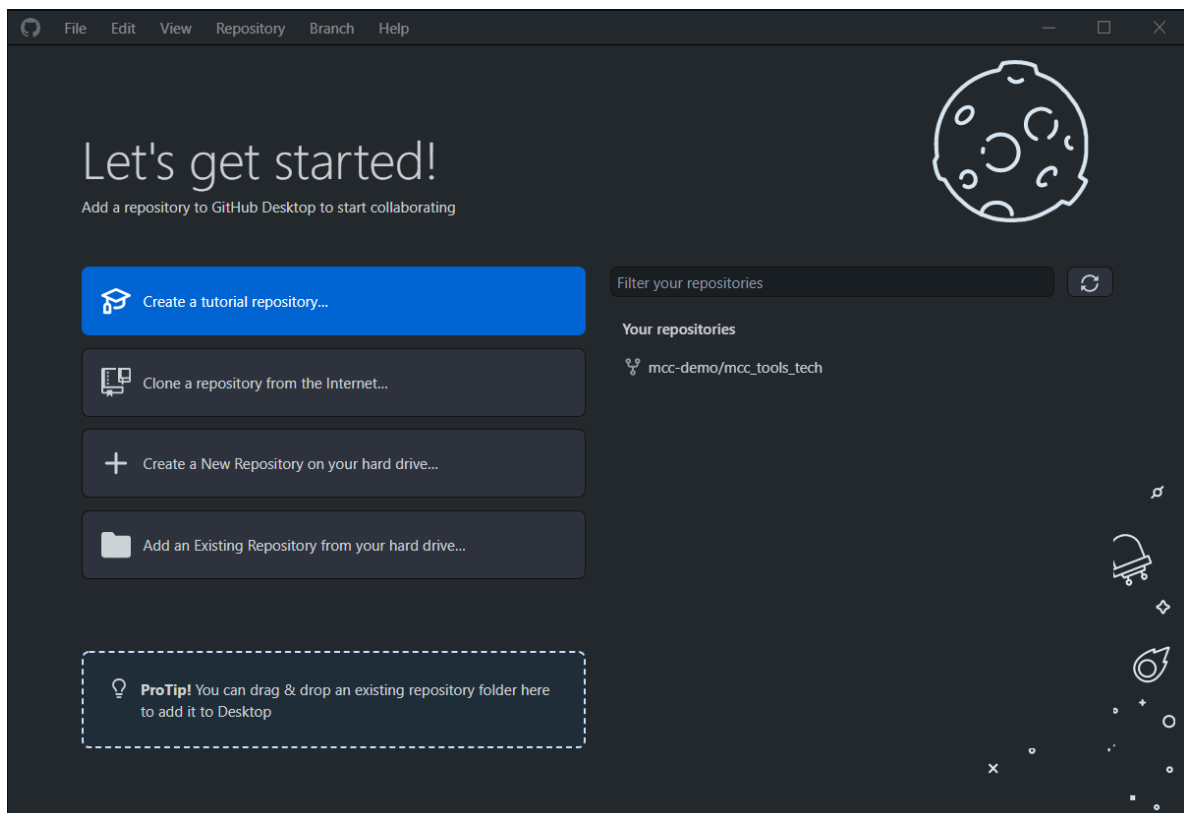
1. From the initial **Let's get started** screen in GitHub Desktop, click **Clone a repository from the Internet**.

    If something has happened and you're not at the GitHub Desktop welcome screen, select **File** > **Clone repository**.

    This opens the Clone a repository dialog.
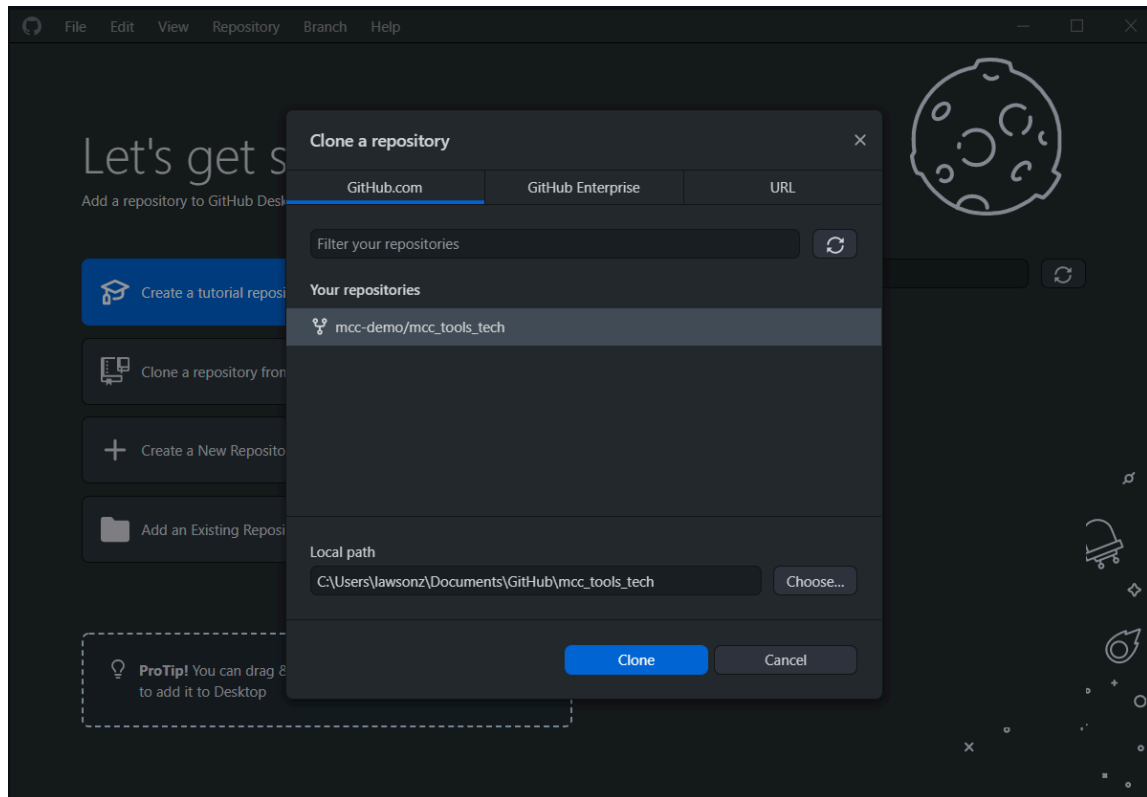
Notice that since you're logged in to your GitHub account, it automatically lists the repositories in your account.

2. Select the mcc_tools_tech repo you forked earlier. If you want, you can change the default location of the files by editing the **Local path**. Click **Clone**.

    Personally, I try to pick some place other than my user's Documents directory. Git repositories can get large and deep. Especially in work scenarios, I've run into a series of nested folders that were longer than what Windows could handle. Nothing like PDF creation failing because the path to the file is too long. Also, because I am usually doing scripting things associated with my

content, I want to avoid spaces in file and folder names. While they are allowed, they can make things complicated.

Note: Remember the location of the Local path. This is where your files are. You need to be able to find this as you work in this course.
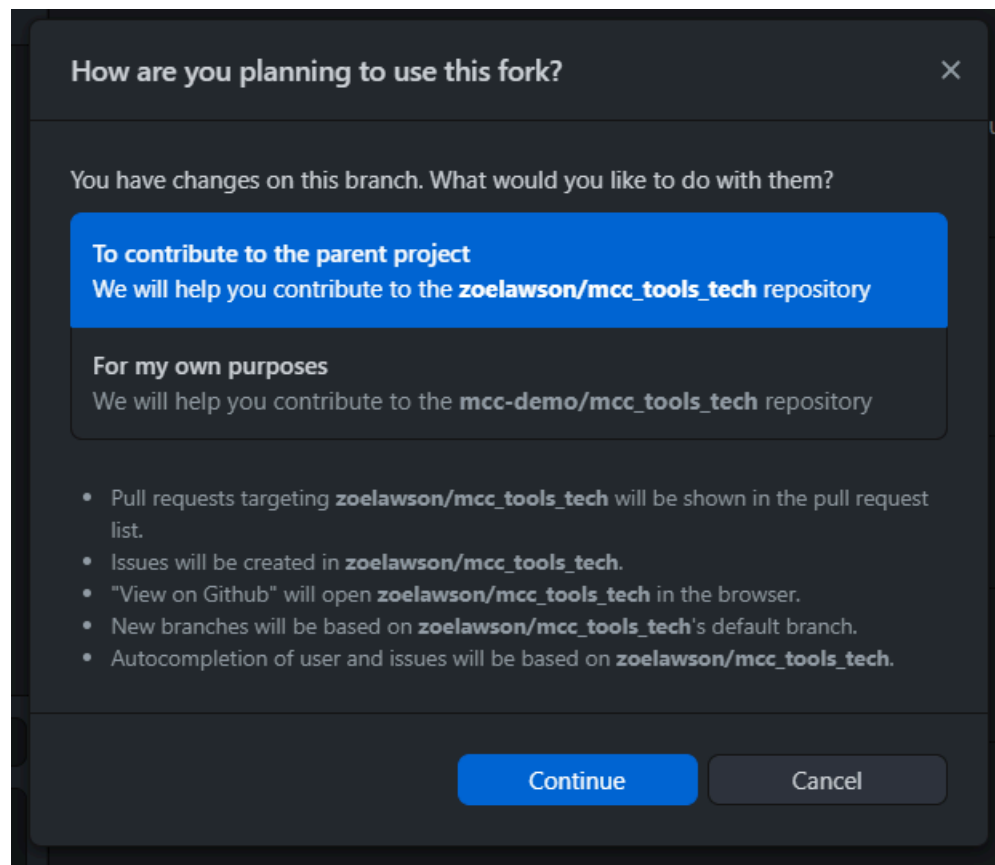

GitHub Desktop is now offering more behind the scenes configuration options. If you were forking the repository because you wanted to make your own version of it, you would select For my own purposes. In this class, however, you are going to be wanting to update the original ZoeLawson/mcc_tools_tech repository.

3. Select **To contribute to the parent project** and click **Continue**.

Congratulations! You have set up GitHub Desktop to work with the fork of this class's repository. You're now ready to start the homework.

Introduction to the GitHub Desktop

Now that you have GitHub Desktop installed, let's take a few moments to get acquainted with the application.
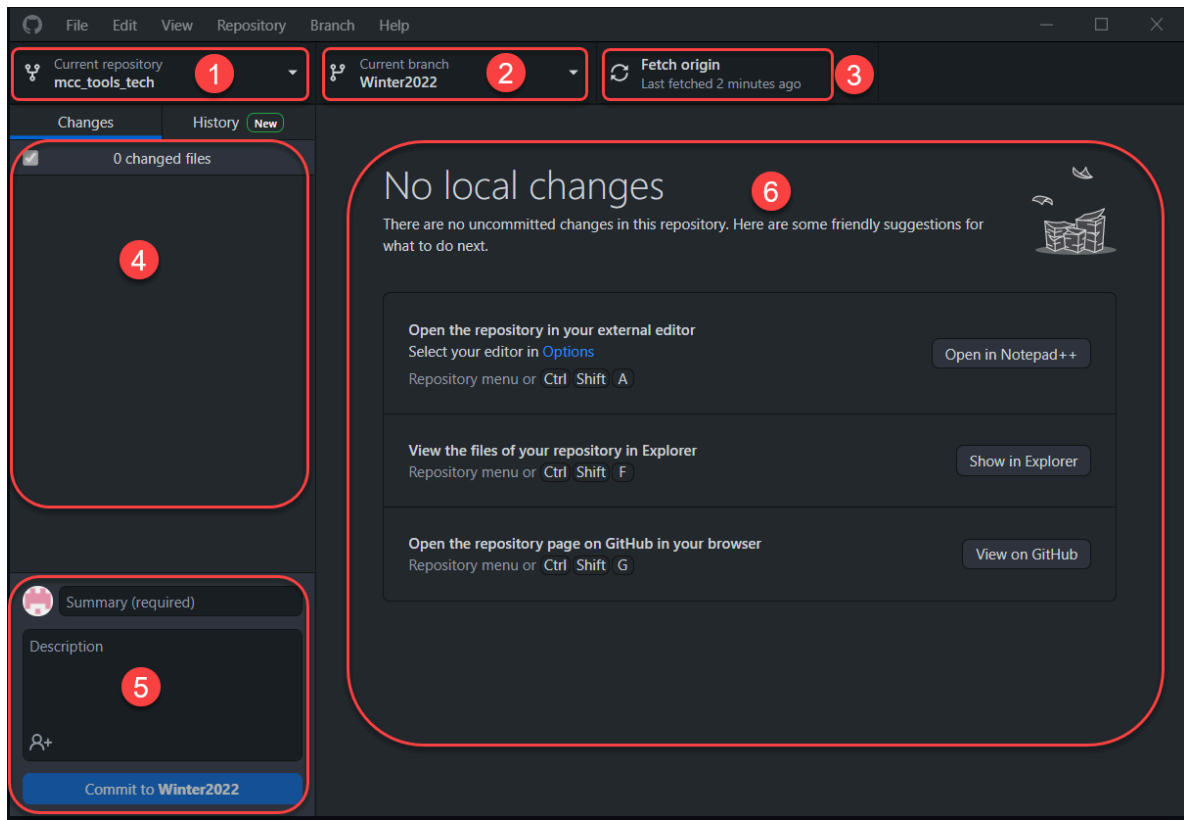
1. The current repository. Right now, you are only connected to one repository, but when working on multiple projects, make sure you're working on the correct project.

2. The current branch. This is super important. In general, different branches contain the same files, but with different changes. Pay attention to what branch you are working in. For this class, you want to be working in Winter2022.

3. Sync with your remote repo. Click this to download the latest and greatest from the remote version of the repository and branch currently selected. This most likely isn't as important for this class as you will rarely be making changes to your remote fork. When working in an enterprise scenario, this is probably used much more often.

4. List of files that have changed. Any time you make a change to any file in the repository, that file shows up here.

5. Commit message and description. Every time you commit a change in git, you must provide at least a summary of the change. You can provide a lot more details in the description. This summary appears in the commit history, which can help you troubleshoot in the future. The more exact you can be with your messages, the happier you will be in the future.

6. File preview and suggested next actions. If you have changes listed in the list of files, you can select the file and possibly view it in this panel. If there are no changed files, GitHub Desktop suggests actions you might want to take, such as a push to remote repository or create pull request.

Create your first commit

Your homework is to create a file and add it to GitHub.

1. Create a new text file in *GitHub Desktop Local Directory*`\mcc_tools_tech\Week01-IntroGitHub\Homework`.

   Name the file `FirstNameLastName.txt`. You can see the example `ZoeLawson.txt` in the folder already.

   Add whatever text you'd like to the file. Just remember this is going into a public, professional web site that might be searched by future employers.

   See Create a text file on page 35 if you need more details on making a simple text file.

2. Switch back to GitHub Desktop.

   If you made your file in the correct location, the file should appear in the list of changed files.
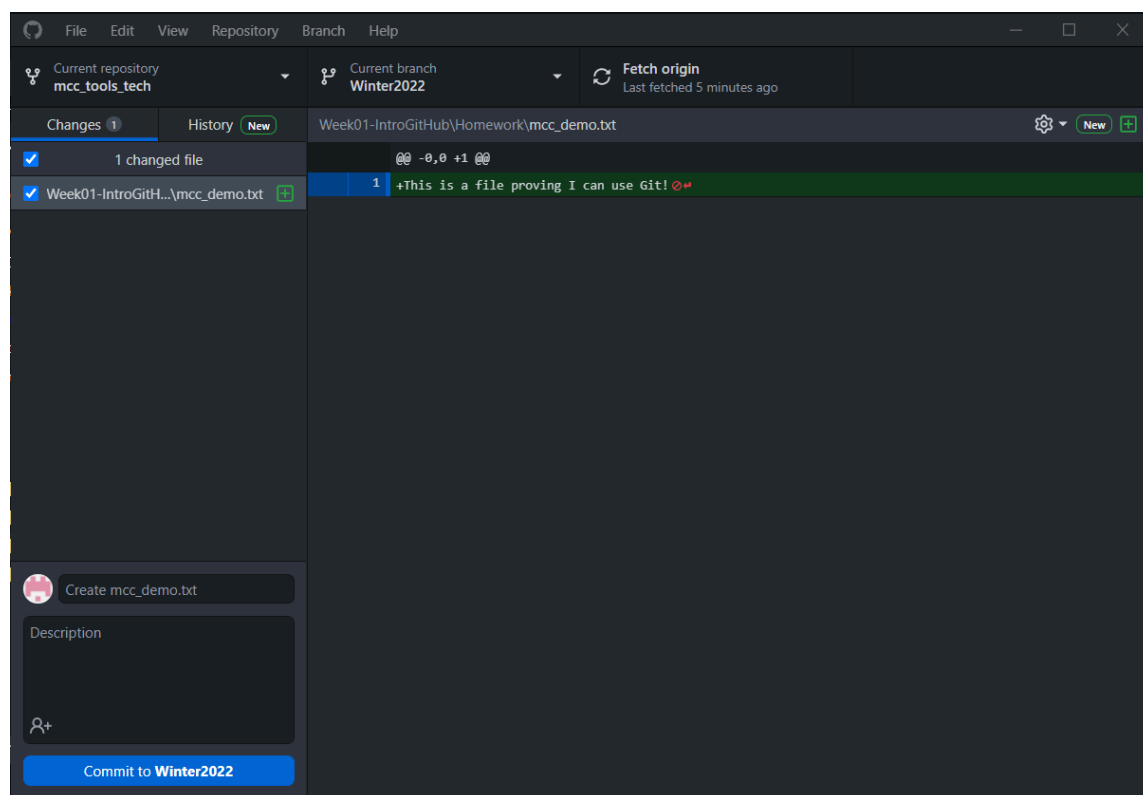

The green plus ⊞ indicates this is a new file to be added.

3. Update the summary for the commit with something more meaningful than "Create mcc_demo.txt".

   I cannot stress enough how important commit messages are when you are troubleshooting things later. This summary is visible when you view the history of the file. At some point in your career, you will be asked "When was this change introduced?" If you have good, clear descriptions in your commit, it will help you down the line. When you look at the history of a file

and all you see is "Updates", that really doesn't help you figure out which of the 15 versions of the file you should be looking at.
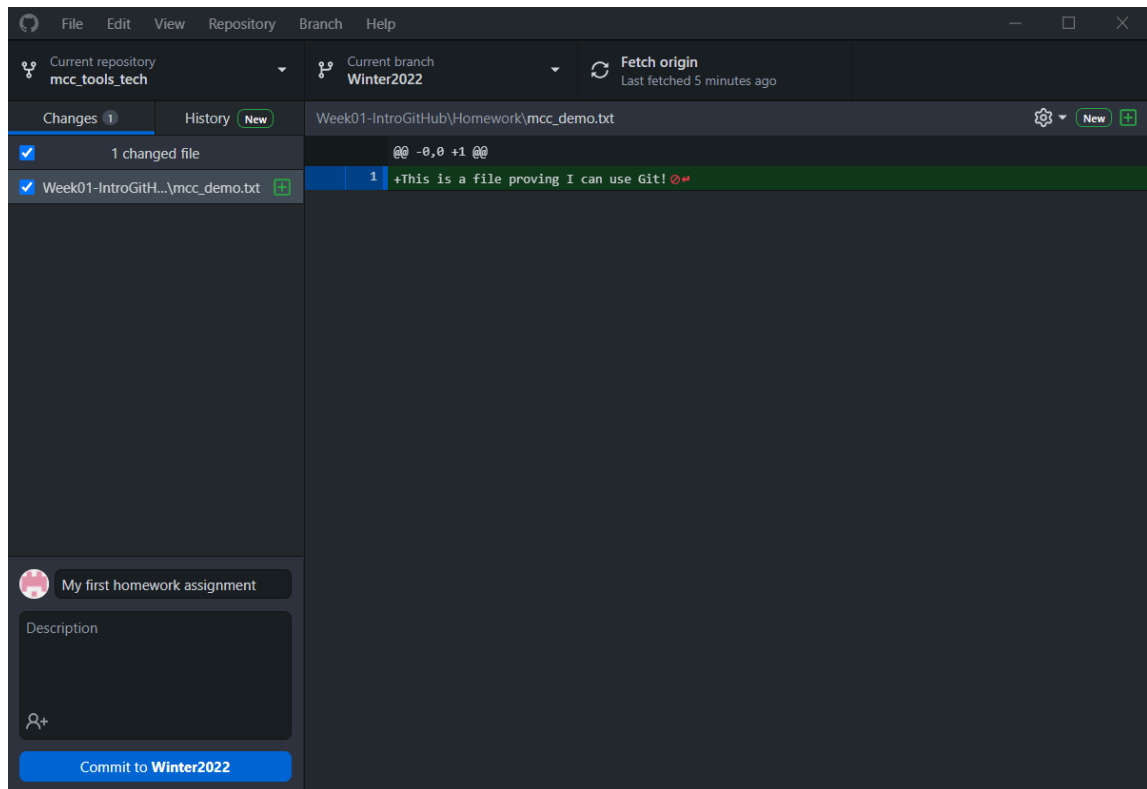

The Description is optional. You can put a lot more detail into the Description in git, but it's harder to view. The Summary (also known as the commit message) is seen everywhere.

4. Click **Commit to Winter2022**.

The changes are now added and committed to your local repository. The git repository on your computer knows about this file and it's logged and tracked.

For facts to file and forget, if you want to undo your change right now, its mildly complicated. We'll cover that some other time.

While the change is committed locally, your remote repository up on GitHub.com has no idea there's been a change. Only people with access to your laptop can find it.
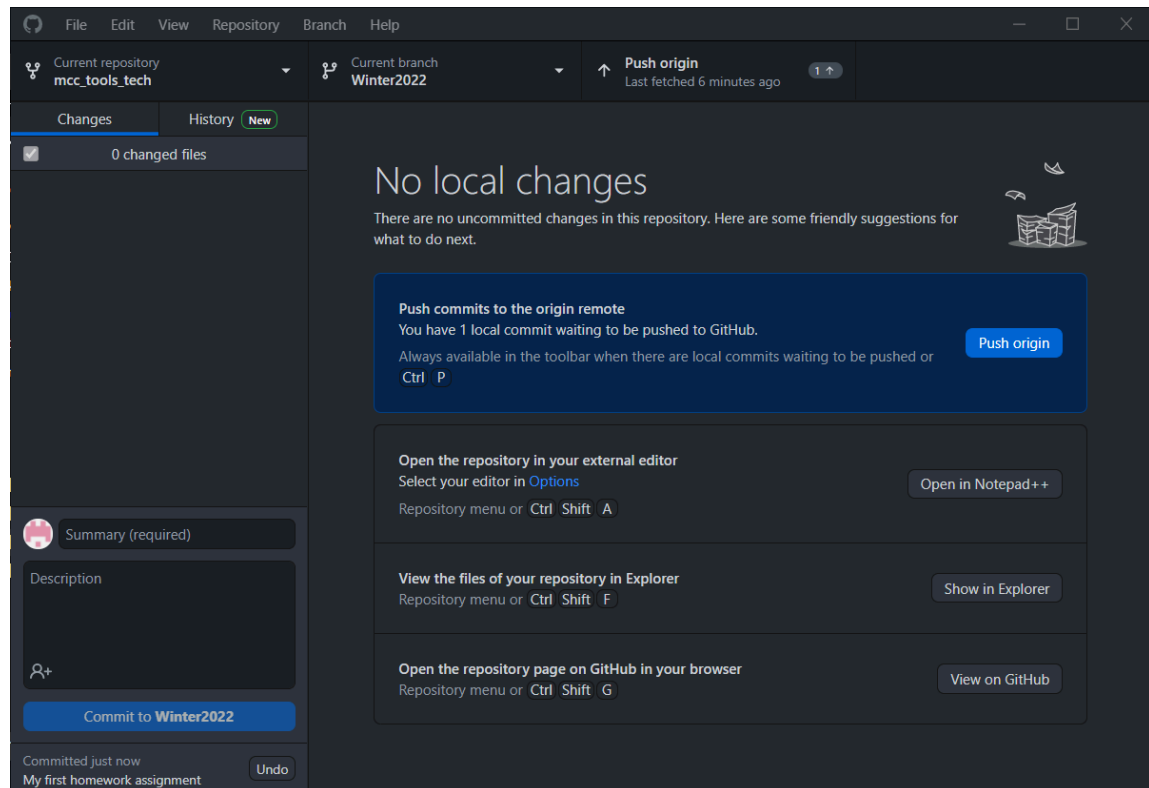

Notice that GitHub Desktop is offering helpful suggestions about what to do next. Now you need to get your changes from your computer to the cloud.

5. Click **Push origin**.

Twiddle your thumbs while the files are uploaded to GitHub.com.

GitHub Desktop performs a push, and uploads the changes to your forked repository up on GitHub.com.

If you go back to your browser, you can see that the file is now there in your remote repository.
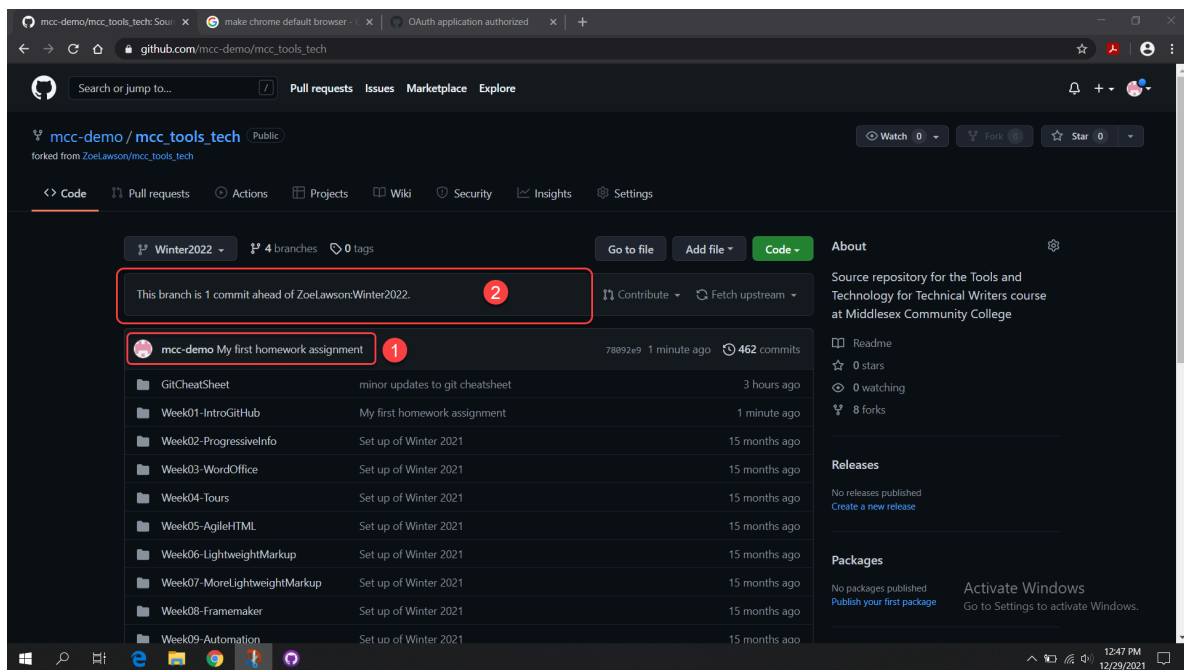
1. See, your commit message is everywhere. GitHub is displaying who made the last commit, and what it was. Any folder or file affected by that commit also has the commit message. Since the change was to `Week01-IntroGitHub\Homework`, you can see that the **Week01-IntroGitHub** folder also has the same commit message.

2. The statement `This branch is 1 commit ahead of ZoeLawson/Winter2022.` indicates that there is a change (a commit) in this fork (mcc-demo/mcc_tools_tech) that does not exist in the original (upstream) repository. We solve this by making a pull request.

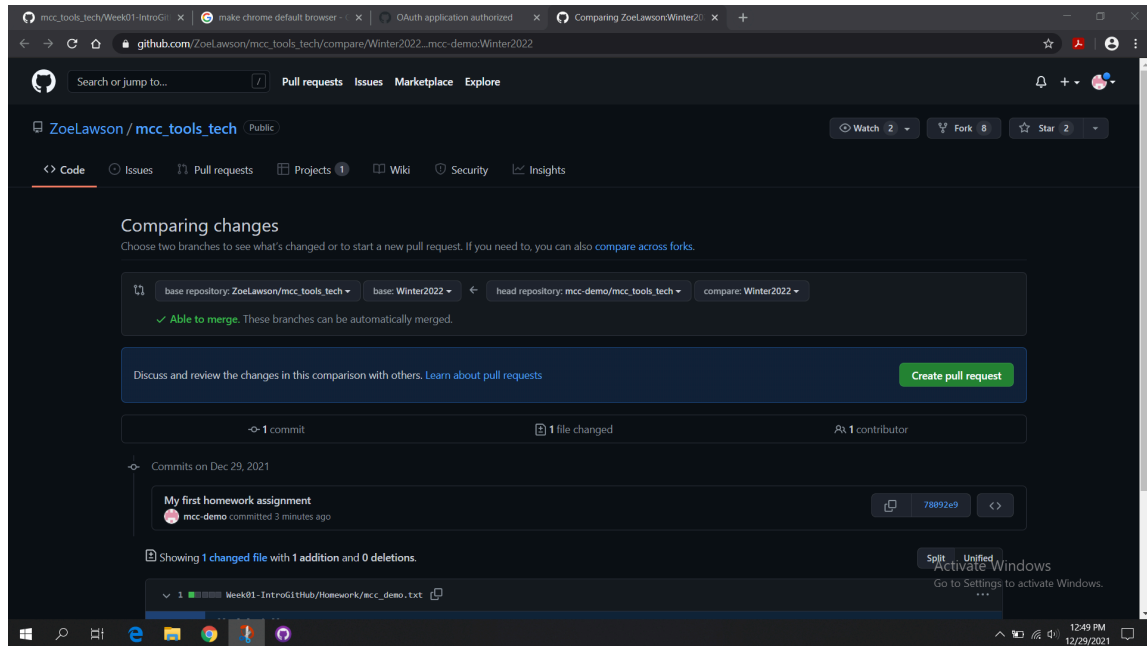## Make a pull request

Now that you've made a change to your forked repository that you are happy with, make a pull request to get that change into the original repository for the class.

What you are doing is saying "Hey owner of original repository! I have a change I think you should have! I want you to pull my changes from my repo to your repo!" This is why it's a "pull request", not a push.

1. In your browser, go to the original repository, ZoeLawson/mcc_tools_tech in GitHub.

   If you were following along from the previous step, you're probably already in your GitHub repository in your browser.

   If your in GitHub Desktop, you'll probably be seeing a prompt for **Create Pull Request**. When you click the button, it switches to your browser and automatically goes to ZoeLawson/mcc_tools_tech.

2. Click a **Create pull request** or **New pull request** button.

   If there are files in your repository that are not in the upstream repository, GitHub generally puts a **New pull request** button somewhere obvious. If you can't find one, go to the **Pull Requests** tab. There is always a **New pull request** button there.
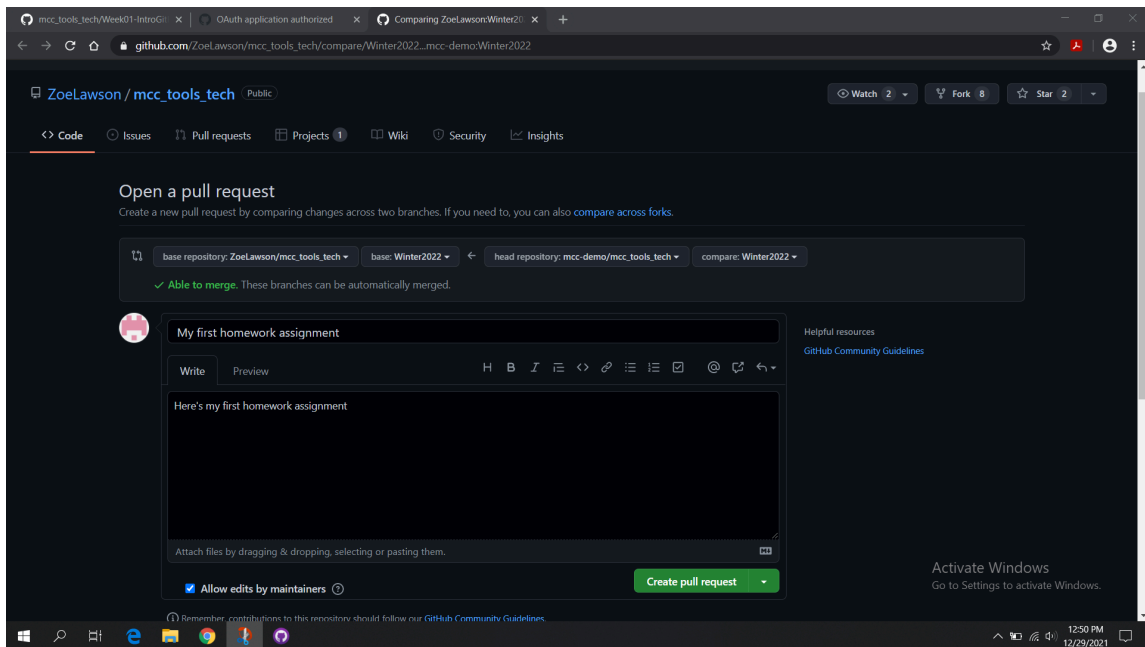
3. Confirm the following:

   - the **base repository** is the original repository (ZoeLawson/mcc_tools_tech)
   - the **base** branch is the Winter2022 branch
   - the **head repository** is your forked repository (*YourGitHubUserName*/mcc_tools_tech)
   - The **compare** is also the Winter2022 branch

4. Add some description of the changes you're requesting to have merged in.

   In this example, there's only one commit in the pull request, so the commit message is adequate. Often, a pull request may contain multiple commits. You may want to summarize the changes. Remember in the future, you may not know the person reviewing the pull request. Even if you know the person, they may not be intimately aware of whatever you're working on.

   I review the pull requests for my team of writers. All of our books are in a single repository. We've had issues with bad push/pull practices in the past where folks have overwritten other people's work. Therefore I try to check if coworker A's pull request only contains files for the books they're currently working on. But, I don't always know everything they're working on. So I ask that people include something meaningful in the pull request description. For example, `Carl's edits` is not okay, but `Carl's review comments for API Guide` is better.

5. Click **Create Pull Request**.

The Pull Request is now open.
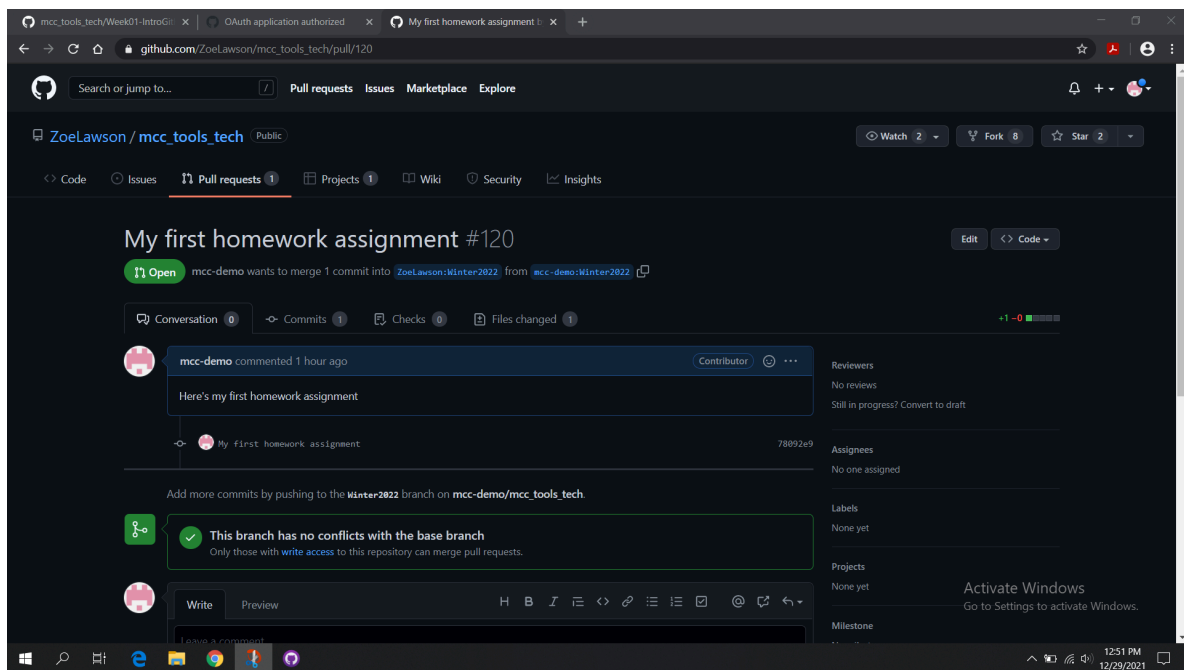
I (or whoever owns the upstream repository) get some sort of notification. I will then review your request and most likely merge the change. Or I will contact you if I can't merge it and suggest changes. I generally respond quickly, but remember, I have a day job and on a rare occasion, a social life.

Important fact to remember: Until I merge and close the pull request, any commit you make and push to your remote repository is automatically added to the existing pull request. If you forget to add a file, or find a typo, you might be able to fix it before I merge it.

That should give you some sort of notification, and then you should be able to see your files are merged in.
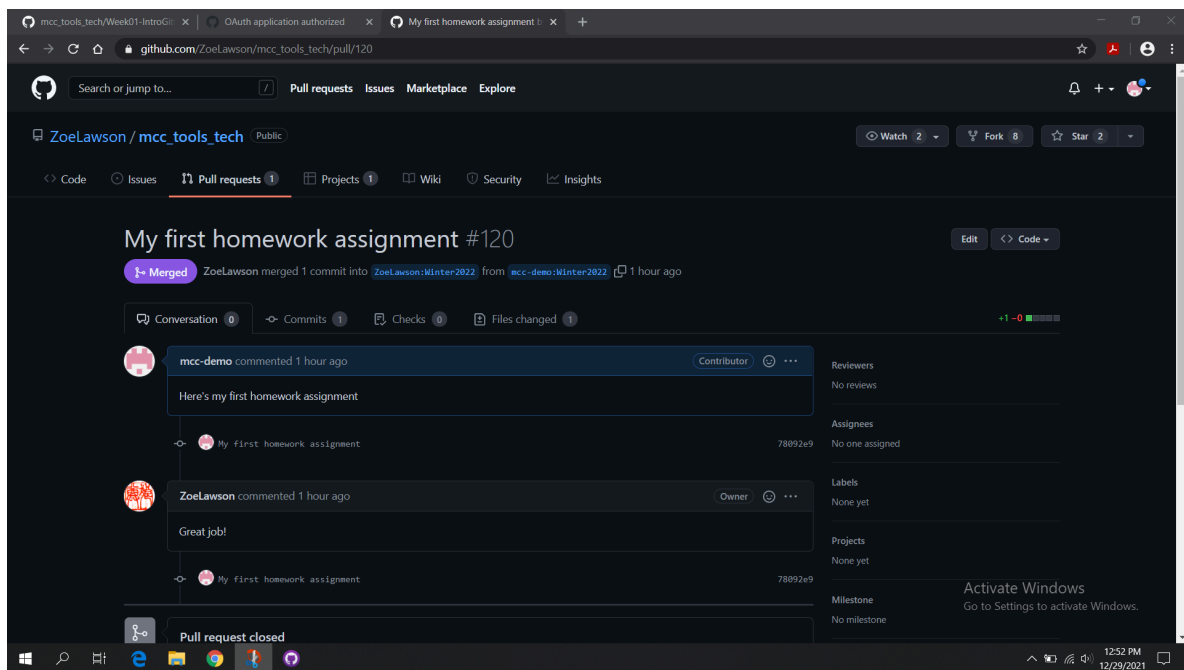
Something that took me a few minutes to notice: All pull requests are in the original repository (ZoeLawson/mcc_tools_tech). You will not see your pull requests in your forked repository in GitHub.

However, you can always go to the Pull requests tab in ZoeLawson/mcc_tools_tech and look at the Closed pull requests. Also notice that you can see the pull requests from previous classes.

Remember, this is all public, and can be found by future employers.

Using the command line

If you'd like, you can use the command line instead of GitHub Desktop.

You must have installed the git command line tool before you can perform these steps. GitHub Desktop does not contain a command line version. These instructions are available at the end of Install GitHub Desktop on page 16.

Learning the command line is not required. However, there are often specific things in git that can only be performed using the command line. This is usually advanced troubleshooting or to handle complicated merge actions or undoing commits.

Clone your GitHub repository using the command line
If you don't want to use GitHub Desktop, you can use the command line to get the files on your laptop. One advantage is that you easily control where the files are.

1. Go to your fork of the mcc_tools_tech repository in GitHub.

    You want to be at the root of the **Code** tab.

2. Click **Clone or download**.

3. Use the default of https or switch to SSH depending on how you configured your Git Credentials, and copy the web URL.

4. On your system, open a command window and navigate to where you want to download the files.

   I tend not to use any of the User directories because on Windows the paths can get stupid long. They also often have spaces in them which make using the command line more difficult.

   I have a directory `C:\ZoeStuff\MCC\github`.

5. Run `git clone https://github.com/ZoeLawson/mcc_tools_tech.git`.

   This makes a `mcc_tools_tech` folder in the `C:\ZoeStuff\MCC\github` directory. The `mcc_tools_tech` folder contains all the files in the repo.

Change branches

The main or master branch is usually protected. Change to the branch associated with the work you need to do.

I've made a branch called **Winter2022** for this class to use.

If we do anything that mucks up the Winter2022 branch, that's fine because the master branch should remain unchanged.

1. When you clone a repository, you are in the default branch, which is usually master. You can check using the status command.

```
git status
On branch master
Your branch is up to date with 'origin/master'.

nothing to commit, working tree clean
```

   The status command is extremely useful. It gives you all sorts of important information such as which branch you're on, how many files/folders are staged and how many are changed and may need to be staged.

2. Change to the Winter2022 branch.

```
git checkout Winter2022
Switched to branch 'Winter2022'
```

   If you want to be super clever, you can also make your own branch. Just add a -b to the command, such as `git checkout -b Winter2022_Zoe`.

3. Optional: Pull from the branch to make sure you have the latest and greatest files.

```
git pull origin Winter2022
```

   You've just cloned, so you should have the latest and greatest, but it's just a good habit to get into.

Add and commit a file

Practice making a change to a file in Git.

1. Navigate to `mcc_tools_tech\Week1\homework`.

2. Make a text file in the directory.

   Please follow the naming convention of `FirstNameLastName.txt`. You can use any text as the content of the file.

3. Optional: Run the status command to see what it does.

```
git status
```

4. Add (or stage) the file.

```
git add .
```

   The `.` indicates to add all the files (including files in sub-folders). You can be more specific and add a particular file or folder.

```
C:\ZoeStuff\MCC\github\mcc_tools_tech>git add Week1\homework
\ZoeLawson.txt
```

5. Optional: Run the status command to see what it does now that you've staged files.

```
git status
```

6. Commit your change.

```
git commit -m "Working on my homework for week 1"
```

   Always use a meaningful comment. A lot of the time it doesn't matter, but when troubleshooting, or trying to find when you made a change, good comments are invaluable.

7. Optional: Run the status command to see what it does now that you've commited changes.

```
git status
```

   Note: Beware the `nothing to commit, working tree clean` message. This means that the files are committed locally only. They may not be up in the remote repository, which means other people can't get to them.

Push to a GitHub repository

The file is saved to your laptop. Now you need to get the file up to GitHub so it's backed up and reachable by anyone with access to your repository.

1. Run the status command to confirm what state you're in.

```
git status
```

   This will remind you which branch you're on.

2. Run the push command.

```
git push origin Winter2022
```

`origin` is the name/alias of the remote repository. Technically you can change it, but most people rarely do. Origin generally equals the remote repository in GitHub.

`Winter2022` is the name of the branch you want to push to. Generally, its the same as the branch you're on locally. Technically, you can link different branches to each other, such as remote/branchA is mapped to your local/branchB, but that can get extremely confusing. You can also push to different branches, but that can also get ugly.

If you were clever and made your own branch, push to the branch you made.

You can go into the repo in your GitHub area. Switch to the Winter2022 branch (or your personal branch). Confirm your text file is there.

Create a text file

Text files are the most basic of files you can create on a computer.

A text file is a file that contains text. Apologies for being circular. Text files are generally raw ASCII text, although it might be possible to encode them in UTF-8 or other encodings. An encoding is a way of stating "use this pattern to interpret the 0s and 1s in this file". If you've ever seen a website where suddenly there are some gobbledy-gook characters in the middle of text, that's probably an encoding issue. Usually the Roman alphabet and numbers are okay, but special characters, such as "smart quotes", or apostrophes can get mangled.

There are many, many different ways of creating a text file. This is probably the easiest in Windows.

1. In Windows Explorer, right-click in a folder and select **New** > **Text File**.
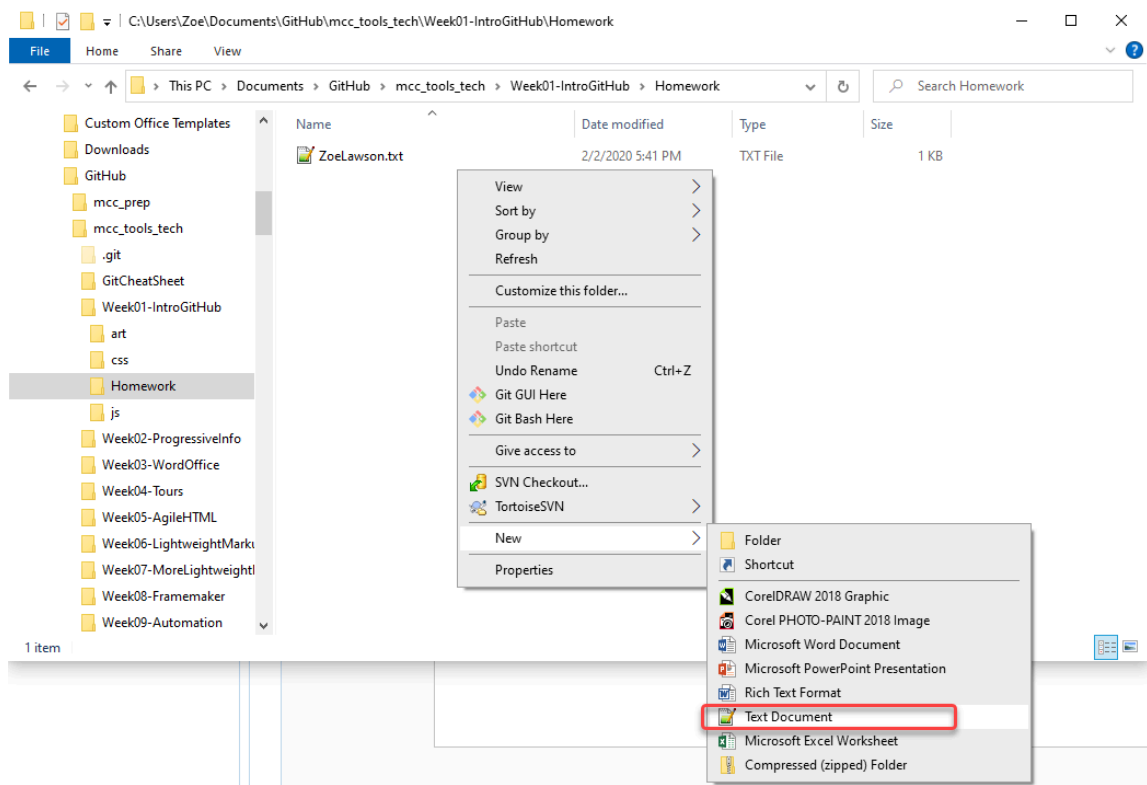
The icon next to Text Document may look different, and that's okay. The icon depends on the default application used for that file type. I have text files associated with Notepad++, an excellent open source text editing tool. I highly recommend it, and it will be useful for future classes. This open source tool is available from https://notepad-plus-plus.org/.

2. Provide a name for your text document. For this Week 1 homework, it should be
*YourFirstNameYourLastName*`.txt`.
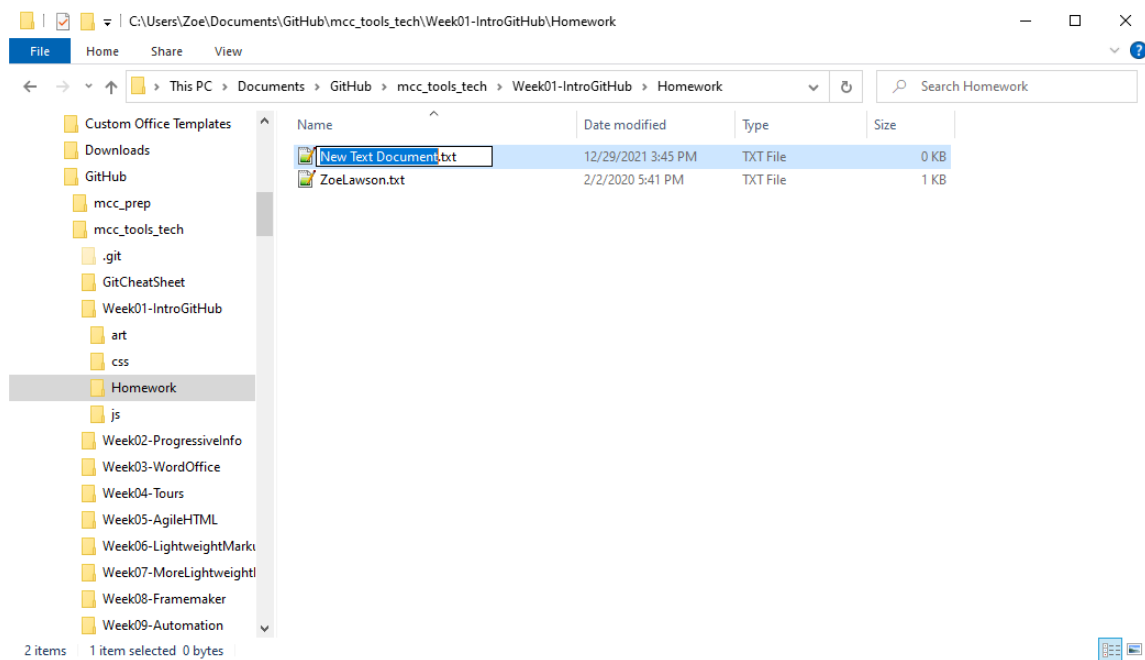
Follow the example of `ZoeLawson.txt`.

You may not see the `.txt`. By default, Windows hides file extensions. I always turn this off, first time I interact with a new PC. Windows is incredibly dumb, and the way it figures out what application it should use to open a file is based on this file extension. So if you change `ZoeLawson.txt` to `ZoeLawson.jpg`, Windows will try to open the text file in the Photos app, and complain.

3. Double click your text file to open it in your default text file editor.

My system is configured to use Notepad++. The default Windows text file editor is Notepad.

4. Add some text to your text file.
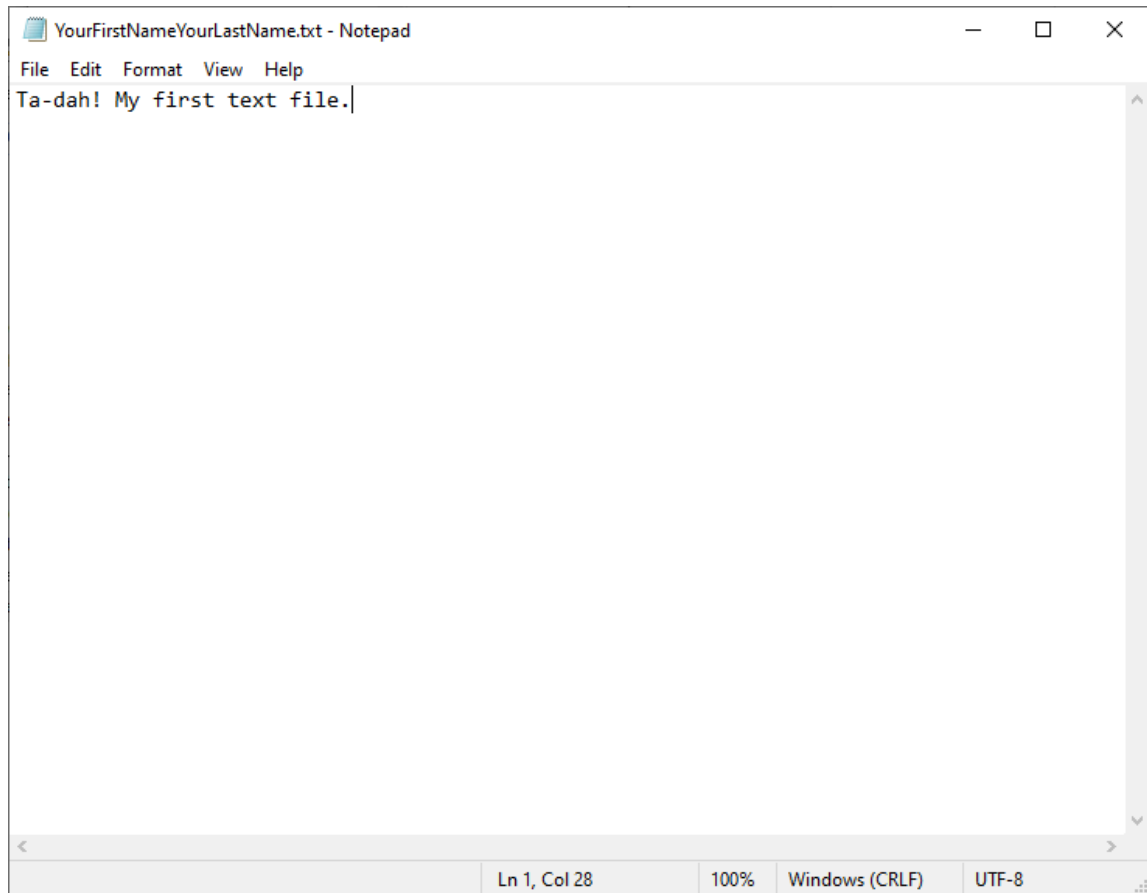
5. Use **CTRL+S** or **File** > **Save** to save your file.
6. Close your text editor.

About conflicts

Conflicts are the scary part of working with Git. They are a fact of working with collaborative source control systems. Learn to not fear them.

Whenever you pull files into your branch, Git is doing a *merge*. Git has some amount of crazy logic that can generally merge files successfully.

If the remote file has changed, but yours hasn't, Git uses the remote file.

If the local file has changed, but the remote hasn't, Git uses your file.

When two people work on the same file, generally people make changes in different parts of the file. If I am editing paragraph three and you're editing paragraph seven, Git generally figures that out and merges the file without issue.

However, if both of us make changes to the same paragraph, Git looks at it, realizes it has no clue and waves a flag asking a human to come and figure it out.

When there is a conflict, Git marks up the text file with something like the following:

```
<<<<<<< HEAD
This is the version of the code on your local version
================================================
This is the version of the code on the remote system
longindecipherablehexcodecommitid >>>>>>>>>>>
```

You then go in and make it work. Delete the text you don't want, keep the text you do want. Sometimes that involves a bit of rewriting.

Be aware that Git can be stupid. If two people are making related changes in the same area, it's possible that Git will overwrite things.

We used to use a lot of conditional text based on version. Two people were working on the same guide for different software versions at the same time. One person went in and added version7 conditions, commited changes and made a pull request. The other person went in and added version9 conditions, commited and made a pull request. The version9 change did not have the version7 condition. Git understood that to mean to remove the version7 condition, not to add the version7 and version9 condition.

This could have been avoided if the other person had done a pull from master before merging...except they both made their pull requests about the same time. The version7 content was not in master before the version9 pull request was made.

This is a rare, esoteric edge case, but it can happen.

Further reading

There is a lot more to Git than what I can cover in an evening. Many others have written much better information.

https://help.github.com/en/github	The help for GitHub.
https://git-scm.com/doc	This is the official home of the git command. All of this information is good and accurate, but a smidge technical.
Version Control with Git	This is the book that was recommended to me when I was informed I had to start using git. It helped me understand how it works and start to accept it as a useful tool. If you need a reference tool, the web is generally a better resource. To learn and understand how to use git, this is a fabulous book.