MIDDLESEX Community College

Tools and Technologies for Tech Writers 2020

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 2020.

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:

> 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, *master*. This is a very important branch. Master 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 master. You make changes on the branch. When you're happy with the changes, you then merge the changes from the branch back into master.

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.

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

Again, this is greatly simplified. You generally are working in an evironment 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 master. 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 master into your branch.

Now, your feature may not actually work well with the other new feature. You could try a fix, merge it to master, 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 master.

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.

## 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
```

## 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 master
```

Do not confuse this with a *pull request*.

### 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.

```
git add path/filename
```

### 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.

### 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
```

### 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:

- master - 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 master. This is your branch. You have rights to push to it.

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

```
git checkout feature_12345
git pull origin master
```

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

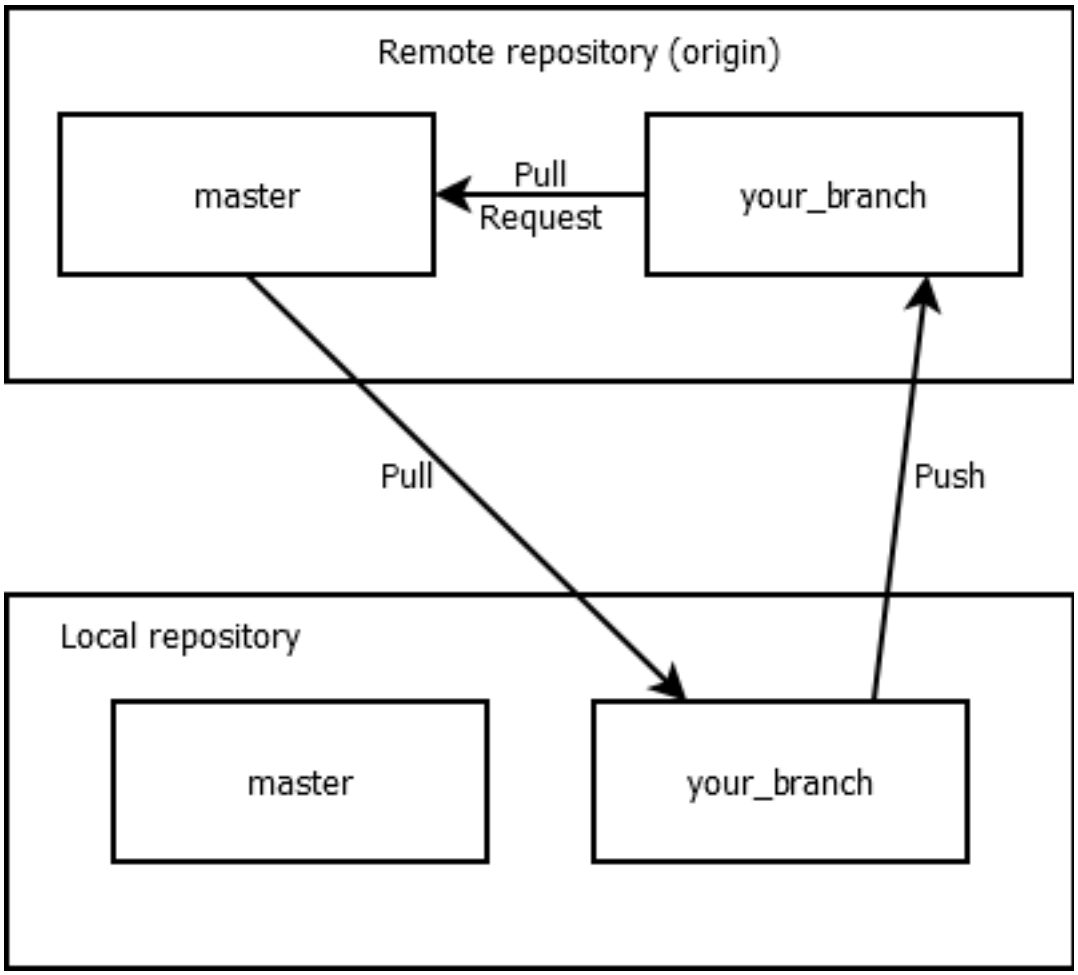
Understanding how local and remote repositories work with pushes and pulls takes a few tries to understand.

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 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.

## Week1 homework

1. Install Git on your laptop.
2. Make a GitHub account.
3. Fork the GitHub repository for this class.
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.

### Install Git

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

1. Go to https://git-scm.com/downloads and download version appropriate for your operating system.
2. Run the installer.

If you'd like, you can experiment with any of the GUI based versions. I'm partial to the command line, so all instructions are given using commands. There should be an equivalent in whichever GUI you use.

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

### Make a GitHub account

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

GitHub is public. Potentially, you can use your GitHub account as part of what you show potential empoyers. I recommend using a professional-ish email address.

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

   See https://help.github.com/en/github/setting-up-and-managing-your-github-user-account if you have questions.
2. Send me your GitHub user name or email address so I can add you as a collaborator to the mcc_tools_tech repository.

   I'm keeping this repo private so anyone doesn't stumble along and get all the class handouts for free.
3. To avoid typing in your user name and password every single time you run a command in Git, configure some automatic authentication with Git.

   See https://help.github.com/en/github/getting-started-with-github/set-up-git#next-steps-authenticating-with-github-from-git.

## Fork a GitHub repository

Fork the mcc_tools_tech repository into your repository so you can make changes.

Go to https://github.com/ZoeLawson/mcc_tools_tech and click **Fork**.

## Clone a GitHub repository

Clone your fork of the mcc_tools_tech repository to your laptop.

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.

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

I've made a branch called **Winter2020** for this class to use.

If we do anything that mucks up the Winter2020 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 Winter2020 branch.

```
git checkout Winter2020
Switched to branch 'Winter2020'
```

   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 Winter2020_Zoe`.

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

```
git pull origin Winter2020
```

   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 Winter2020
```

`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.

`Winter2020` 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 Winter2020 branch (or your personal branch). Confirm your text file is there.

## 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 class repository.

1. Go to the fork of your repository in GitHub.
2. Click a **New pull request** button.

You may need to go to the branch.

3. Confirm that the **base** is the original repository (ZoeLawson/mcc_tools_tech) and the **compare** is your branch in your fork.
4. Add some description of the changes you're requesting to have merged in.

   For this homework assignment, it doesn't entirely matter. But 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.
5. Click **Create Pull Request**.

I should get some sort of notification. I will then review your request and most likely merge the change. That should give you some sort of notification.

# 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://github.com and https://help.github.com/en/github

https://git-scm.com/doc

Version Control with Git