Table of Contents

Introduction to Git & Github

Brief distinction between Git and Github

Git is a "version control" system that helps track changes in your code, while github is a web-based platform that allows you to share and collaborate on your code using git's version control system.

Creating a Digital Carleton Github repository

1. Visit your Github dashboard
2. If you're not already logged in, then login to your github
3. Visit Digital Carleton's repos
4. Click on `New Repository`
5. Enter a name for your repository that suits your ticket
6. Set your repository either to `public` or `private`
7. (optional) add a readme / .gitignore / license

Setting up your Digital Carleton github repository

Once you create a github repository you will be presented with the following instructions on setting up your repo:

```
## Create a new repository on the command line

1. touch README.md
2. git init
3. git add README.md
4. git commit -m "first commit"
5. git remote add origin git@github.com:username/<reponame>.git
6. git push -u origin main

## Push an existing repository from the command line
```

```
1. git remote add origin git@github.com:username/<reponame>.git
2. git push -u origin main
```

... but what do all these instructions mean/do?

- `git init` creates a local repository
- `touch` is a bash command for creating a file in your current directory
- `git add` adds files to the staging area.
- `git commit` commits your changes to the git timeline.
- `git remote add origin ...` adds an origin to your local git repository that ties back to your remote github repository
- `git push -u origin main` "pushes" all your latest commit "updates" to the origin or i.e. your github repository

Note: You may have noticed that some of these commands talk about staging, commiting, and pushing, and may want to review the Understanding Git section to better comprehend what this means.

Understanding Git

Git has multiple modes for files. These modes include `untracked`, `unstaged`, `staged`, and `committed`. Below is a quick summary on them:

- **Untracked:** Files that are not being tracked by git
- **Unstaged:** Files that are being tracked but have yet to be "staged" or i.e. have yet to be marked for commit.
- **Staged** Files that have been added to the staging area. They are ready to be committed and will be included in the next snapshot once committed.
- **Committed:** Files that are committed to your repo and are "snapshots" or "milestones" along the timeline of a git project. You usually commit with a very brief message that summarizes your changes.

Throughout your work on a project you will want to make commits at set intervals where you feel it adequately appropriate. Additionally, you will want to routinely `push` these commits to the github repository so that anyone else working on your project has the latest version of your code.

Additionally, git has multiple moving parts which you may want to familiarize yourself with such as `branches`. Branches are snapshots of a repository where your snapshot deviates from your `main` branch or i.e. timeline. These branches are usually used for when multiple people are working on multiple features and you want to work on a feature independently from the main branch.

Once the feature is complete you usually pull the the latest version of the main branch, `merge` it with your current branch, and push it to origin. Once this branch reaches origin (the github repository), either you or an administration reviews the code and merges it in to the main branch.

Below is a bit of a cheat sheet that you may wish to refer to here and there when working with git

> Note: Anything in angle brackets (i.e. < >) is of that type, and so you're to decide on the proper input

Example Workflow:

Updating Git Project

1. Switch to main branch and pull the latest code
   `git checkout main`

   `git pull --rebase origin main`
2. (optional) Create a feature branch and switch to it (if you're not working with anyone else in this git repository then you really don't need to create separate branches)
   `git checkout -b <branchName>`
3. Make changes to code
4. Add all changes to the staging area. Below are multiple ways to how this can be done:

   Stage all tracked files with changes:
   `git add -u` (i.e. git add --update)

   Add all files in the current directory and subdirectories to the staging area:
   `git add --all`

   Add the files in the current directory to the staging area:
   `git add .`

   Add the file individually to the staging area
   `git add path/to/file`
5. (optional) Check that your changes were made and that your changes were committed to the staging area (text should be green)
   `git status`
6. Make a commit with a descriptive message
   `git commit -m "Message"`
7. Make sure to have the latest version before pushing
   `git pull origin main`
8. Push branch to your repository origin
   `git push origin <branchName>`
9. Update Trello Board
10. Visit the github repository, review the code, and merge it.

Additional Git Commands

- create a branch
  `git branch <NAME>`

- switch to branch
  `git checkout <NAME>`

- create a branch & switch to it simultaneously
  `git checkout -b <NAME>`

- ammend latest git commit (use any of the following commands)
  `git commit --amend -m "New Commit Message"`

  ```
  git commit --amend
  :x
  ```

- ammend older git commit
  `git rebase -i HEAD~N`

  Replace N with the number of commits back from the current HEAD you wish to review for editing. Git will open a list of the last N commits in your default text editor. In the editor, replace pick with reword next to the commit you wish to modify, save the file, and close the editor. Git will then reopen the editor for you to reword your commit message.

- display previous commits (use any of the following commands)
  `git log`

  `git log --oneline`

  `git log --graph`

  `git show <COMMIT-HASH>`

  `git reflog`

  -n : (optional flag) This limits the number of commits displayed to the last commits

- Undo a commit without discarding the changes made in the commit
  `git reset --soft HEAD~N`

  Git will move the HEAD pointer to the commit N commits before the current HEAD, keeping the changes from those commits staged in the index. This effectively undoes the last commits while leaving your changes from those commits in the staging area

- delete local branch
  `git branch -d <branchName>`

- command for managing remote repositories in Git
  `git remote`

  -v or --verbose: This option provides additional information, specifically the fetch and push URLs for each remote URL of the remote repositories

- ignore files from all future commits
  `git update-index --assume-unchanged <filePath>`
- remove unstaged commits
  `git restore .`
- file to setup ignore rules specific to anyone with that repo
  `touch .gitignore`
  include whatever files / fileTypes you want to ignore in this file
- file to setup ignore rules specific to your clone of the repo.
  `echo “<files>” >> .git/info/exclude`
  include whatever files / fileTypes you want to ignore in this file
- manually tell Git to assume that a file is unchanged even if it actually is changed
  `git update-index --assume-unchanged <filePath>`
- to start tracking changes for assume-unchanged files use
  `git update-index --no-assume-unchanged`
- display all files that are assumed to be unchanged by git
  `git ls-files -v | grep '^[a-z]'`

Git Stashing

- List all stashes
  `git stash list`
- Stash uncommited code for later use
  `git stash`
- Apply a stash, and remove the stash
  `git stash pop`
- Apply a particular stash and remove the stash
  `git stash pop <stashRef>`
- Apply the most recent stash
  `git stash apply`
- Simply apply a particular stash
  `git stash apply <stashRef>`
- Drop a specific stash
  `git stash drop <stashRef>`
- Clear all stashes
  `git stash clear`

Bash & Terminal

- View the "man" (i.e. manual) page of a given command via terminal
  `man <cmd>`
- create a file via bash (your terminal window)
  `touch <fileName>`
- list the files in your current directory
  `ls`
- change to another directory
  `cd <directoryPath>`
- print a files content to your terminal
  `cat <fileName>`
- open a file / folder in its default app
  `open <filePath/folderPath>`
- If you have Visual Studio Code installed and the code command was added to your system path, then you may be able to open a folder in Visual Studio Code directly, by doing the following:
  `code <filePath>`
- search terminal history
  `ctrl + R (for linux and osx)`
- jump to end of history / stdout / terminal response
  `shift + >`
- exit out of log
  `:q`

Bonus: "." is a path to your current directory & ".." is your parent directory. This means you can use these with commands such as "code" to open that directory, or the parent directory respectively.

Final Notes:

This not a comprehensive list and there is honestly too much to cover to try to put it into one single document. This document only aims to get you started with git, github, and even bash / your terminal. At the end of the day to truly build comfort with all these tools you simply need to use them and play around with them often.