Git and Github

A developer’s best friend

What is Git?

What is Git?

- Git is a Version Control System (VCS) designed to make it easier to have multiple versions of a code base, sometimes across multiple developers or teams

What is Git?

- Git is a Version Control System (VCS) designed to make it easier to have multiple versions of a code base, sometimes across multiple developers or teams
- It allows you to see changes you make to your code and easily revert them.

What is Git?

- Git is a Version Control System (VCS) designed to make it easier to have multiple versions of a code base, sometimes across multiple developers or teams

- It allows you to see changes you make to your code and easily revert them.

- It is NOT GITHUB!

About Git

- Created by Linus Torvalds, creator of Linux, in 2005
  - Came out of Linux development community
  - Designed to do version control on Linux kernel

- Goals of Git:
  - Speed
  - Support for non-linear development (thousands of parallel branches)
  - Fully distributed
  - Able to handle large projects efficiently

  - *(A "git" is a cranky old man. Linus meant himself.)*

Ok, then what is Github?

- **Github.com** is a website that hosts git repositories on a remote server

Ok, then what is Github?

- **Github.com** is a website that hosts git repositories on a remote server
- Hosting repositories on Github facilitates the sharing of codebases among teams by providing a GUI to easily fork or clone repos to a local machine

Ok, then what is Github?

- **Github.com** is a website that hosts git repositories on a remote server

- Hosting repositories on Github facilitates the sharing of codebases among teams by providing a GUI to easily fork or clone repos to a local machine

- By pushing your repositories to Github, you will pretty much automatically create your own developer portfolio as well!

Confirm that you have git

- Open your terminal and run ‘git’
- If you see a ‘command not recognized’ error, you probably haven’t installed git yet.
- We can solve your issue when the lectures are over - sit tight for now!

Configuring git

Your commits will have your name and email attached to them. To confirm that this information is correct, run the following commands:

```
$ git config --global user.name
> should be your name, i.e. Jon Rosado
$ git config --global user.email
> should be your email, i.e. jon.rosado42@gmail.com
```

To fix either, just add the desired value in quotes after the command:

```
$ git config --global user.name “Jon Rosado”
$ git config --global user.email “jon.rodado42@gmail.com"
```

Using Git / Github

Connecting git with Github

- In order to prevent having to enter your password each time you push up to Github, you must configure git and Github to recognize Secured Shell (SSH) keys that you generate.

- To check and see if you have any recognized SSH keys active on Github, go to https://github.com/settings/keys

- If you do not see any SSH keys listed, you do not have SSH configured with Github. We can assist with this later.

- If using Mac OS X, you can also configure your keychain to automatically enter your password via HTTPS.

Connecting git with Github

- From your project directory, run `git init` to initialize a git repository.

- Go to Github, and create a new repository with the name of your project.

- Follow the instructions on Github to connect your initialized git repository to the remote server on Github.

- *Please Note: you must have files in your project directory to commit in order to push anything to your remote server.

Basic git / Github workflow

- From your project repo on Github, navigate to the Issues tab and create a new Issue

- From the command line, use git to create a new branch off of master to make your edits to. To tie the branch to your issue on Github, make sure to include the issue number after the branch name, e.g. branchName #1

- Stage edits to be committed to your git repository by using `` `git add <file name>` `` to track the files that you want to add directly or `` `git add .` `` to add all files at the current directory level that you have worked on.

- Commit changes using `` `git commit -m <message>` `` and be sure to leave a short but descriptive message detailing what the commit will change when merged to the master branch.

- Push changes to save them by using `` `git push origin <branch name>` ``

Basic git / Github workflow

- On Github, create a new pull request for the branch that you have just pushed, and add any clarifying comments that you deem necessary.

- In order to close the issue associated with the pull request, tell GitHub to do so by adding ‘closes <issue number>’ in the comments. (you can also use ‘closed, fix, fixes, resolve, resolves, and resolved’ before the issue number as well).

- Github will automatically check for merge conflicts. If there are any, check to see what they are and resolve them.

- Once everything is up to date, Github will allow you to merge using three separate options: Merge; Squash and Merge; Rebase and Merge.

git branching

- `git branch` to view current branches in repo
- `git checkout -b <branch name>` to create a new branch with branch name
- `git checkout <branch name>` without ‘`-b`’ flag to switch to existing branches

Merging vs Rebasing

- From a conceptual standpoint, git merge and git rebase are used to achieve the same ultimate goal: to integrate changes from one branch into another branch. There are, however, distinct mechanics to both methods.

- We will be discussing how to use each from the context of adding changes from your master branch into your current working feature branch.

git merge: pros

- Generally the easiest option to merge your master branch into your current working feature branch.

git merge: pros

- Generally the easiest option to merge your master branch into your current working feature branch.
- You can `git checkout feature` and then `git merge master` or you could just do it with one command: `git merge feature master`

git merge: pros

- Generally the easiest option to merge your master branch into your current working feature branch.

- You can `` `git checkout feature` `` and then `` `git merge master` `` or you could just do it with one command: `git merge feature master`

- By doing this, you create a new ‘merge commit’ in your feature branch, which is a non-destructive operation that ties the histories of both branches. This preserves the exact history of your project

git merge: cons

- The branch that you merge will always have an extraneous merge commit that will be tracked every time you need to incorporate upstream states.

git merge: cons

- The branch that you merge will always have an extraneous merge commit that will be tracked every time you need to incorporate upstream states.
- In other words, it essentially creates a forked history at the point where you merge.

git merge: cons

- The branch that you merge will always have an extraneous merge commit that will be tracked every time you need to incorporate upstream states.

- In other words, it essentially creates a forked history at the point where you merge.

- This can lead to muddling the history of your branch, thereby making it more difficult for yourself or other developers to track the history of changes using `git log` and/or roll back to previous states.

git rebase: pros

- To rebase, you would `git checkout feature` and then `git rebase master`.

git rebase: pros

- To rebase, you would `git checkout feature` and then `git rebase master`.
- Instead of creating a merge commit, rebase will move the entire feature branch to start from the tip of the master branch by rewriting the project history and creating brand new commits for each commit in the original branch.

git rebase: pros

- To rebase, you would `git checkout feature` and then `git rebase master`.

- Instead of creating a merge commit, rebase will move the entire feature branch to start from the tip of the master branch by rewriting the project history and creating brand new commits for each commit in the original branch.

- The result is a singular history with no forking of the commit history.

git rebase: cons

- Because rebase rewrites project history, you lose the context provided by a merge commit, i.e. you won’t be able to see when upstream changes were actually integrated into the feature branch.

git rebase: cons

- Because rebase rewrites project history, you lose the context provided by a merge commit, i.e. you won't be able to see when upstream changes were actually integrated into the feature branch.
- More importantly, you could potentially cause extreme difficulty by rebasing master to the tip of your feature branch, leading git to think that your master branch's history has diverged from the rest

git rebase: cons

- Because rebase rewrites project history, you lose the context provided by a merge commit, i.e. you won't be able to see when upstream changes were actually integrated into the feature branch.

- More importantly, you could potentially cause extreme difficulty by rebasing master to the tip of your feature branch, leading git to think that your master branch's history has diverged from the rest

- In doing so, everyone else would still be working from the original master branch, and both masters would need to be merged together.

Merge conflicts

- Merge conflicts arise when two members of the same development team work on the same file and try to merge in their respective changes.

- The proper way to avoid merge conflicts would be to ensure that only one branch is fully committed, pushed, and merged to master, allowing the other branch to integrate any changes before attempting to push and merge to master.

- If merge conflicts arise, don’t fret! Many text editors (as well as Github) provide tools to help track down conflicts and resolve them. Often times, it will show incoming changes juxtaposed with the current state of your file, and allow you to choose which to keep (one or both).