`git` - What and Why

- A "Source Control Management" system
  - or "Version Control" system (VCS)
- Tracks files and changes to files over time
- Replaces "MyDOC-V2-DRAFT-FINAL-FINAL.TXT"
- `git` is most common VCS
  - All employers/teams will use version control
- This course uses `git` like a job would

What is version control, really?

It's like an "Undo" between versions of a file

- Except you declare the different "save points"
- And it handles changes from OTHER people too

Goals:

- One clean, shared, approved version of the code
- But allow people to make changes
    - Clearly in the shared, approved version or not
    - Everyone can get changes once approved

Terminology

You will need to understand certain words and labels

- These will come up again and again
- Programming is abstraction and communication
    - Names and labels are a REALLY big deal
    - Precision of language is vital

A repository (repo)

repo/repository

- A collection of files and folders in version control
- **monorepo** - multiple distinct projects in one repo
- Course repo is a bit unusual
    - Many assignments/notes, not just one project
    - Is only shared with me and TA(s)
        - At job everyone shares the SAME repo
- Each dev has a copy of the repo
    - Git lets them sync up and exchange changes

Commit

As a noun ("a **commit**", "this **commit**")

- A collection of changes
- Often changes to multiple files
- New files, Modified files, Deleted files

As a verb ("**commit** your changes")

- Telling the repo to save a collection of changes
- Creates ability to come back to this set of changes
- A **commit** can be exchanged with other repos

Branch

A **branch** is a named collection of **commits**

- When you **switch branches**
  - Change the "current" files and folders
  - Files/folders might vanish/appear!
  - File changes might appear/go away!
  - The changes are in/not in the branch!
- **Branches are not folders**
  - Course often has similar names
  - Branches and folders are completely distinct

Clone

As a verb ("**clone** that repo")

- To make a copy of a repo
- Creates a new, separate repo
- New repo has all the same files
- New repo has the history of the commits
- Cloned repos can exchange changes

As a noun ("A **clone** of that repo")

- A repo that was copied from another repo

Github

github is a website that hosts repos

- Github.com has competitors (gitlab.com, etc)
    - Github is most popular, but is not unique
- Often acts as a common **centralized** repo
    - Allows many devs to coordinate
    - `git` isn't required to be centralized
        - But employers/teams often do

Github Flow

`git` is a tool for software development

- Can be used many ways
  - Often contradictory options
- **Github flow** is *one* way of using it
  - A very common way, but not the only way
- Course and presentation assume the Github flow

Running at the Command Line

- We will run **command line** commands
- Many students are unfamiliar
- Graphical tools exist
    - But most docs are for Command Line
        - Not just for `git`
    - Important to know how to use
        - And understand

Being "in" a folder/directory

- **folder** is UI term
- **directory** is command line term
- Same thing either way
- Locates the operation in the file system
- **EACH terminal shell** is "in" a folder/directory
    - This is a CRITICAL concept
    - Course is difficult until you understand this
- This is the **current path** or **current directory**

Current Directory Example

- Each Operating System is different
    - Each also has multiple Terminal Programs
    - And/Or multiple shell options
    - Basic concepts are the same
        - Syntax is different

The Prompt

Your Shell may tell you what directory you are "in"

- Mac defaults to NOT giving the full route 🤬
- You can change this
    - search `"show path in prompt"`
- `pwd` or `cwd` should give the full current path

Always know what directory you are "in"

- Commands will behave differently!

Navigating Directories: Basics

- `ls` will list files/directories in current path
  - `dir` in Windows (unless using gitbash)
- `cd SOMENAME` moves into the directory SOMENAME
  - (`cd` means `change directory`)
  - `/` shows a chain of directories
  - `.` means the current directory
  - `..` means the **parent** directory ("up a level")
  - `cd ..` goes up one level

See course readings for more

First Steps - Our Class repo

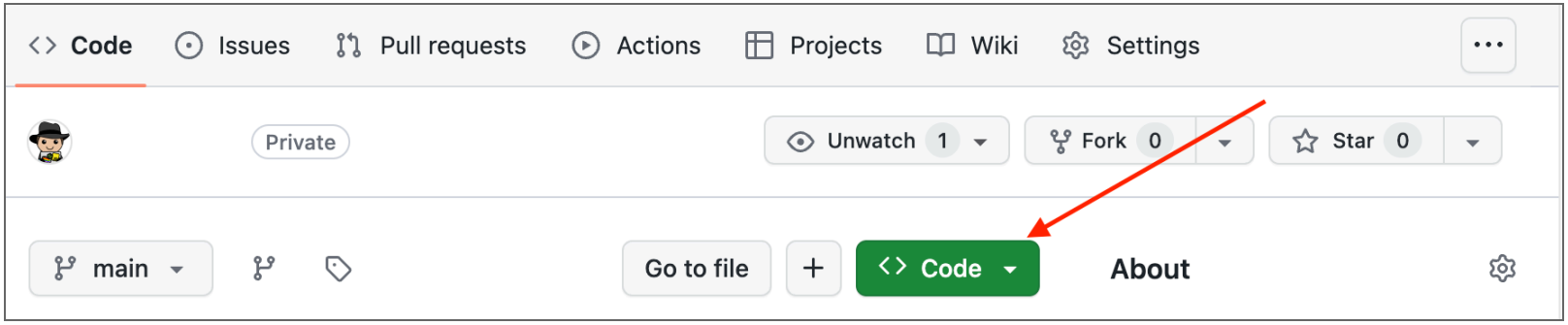
When you follow link, you will get a repo on Github

Next we will **clone** the repo to your computer

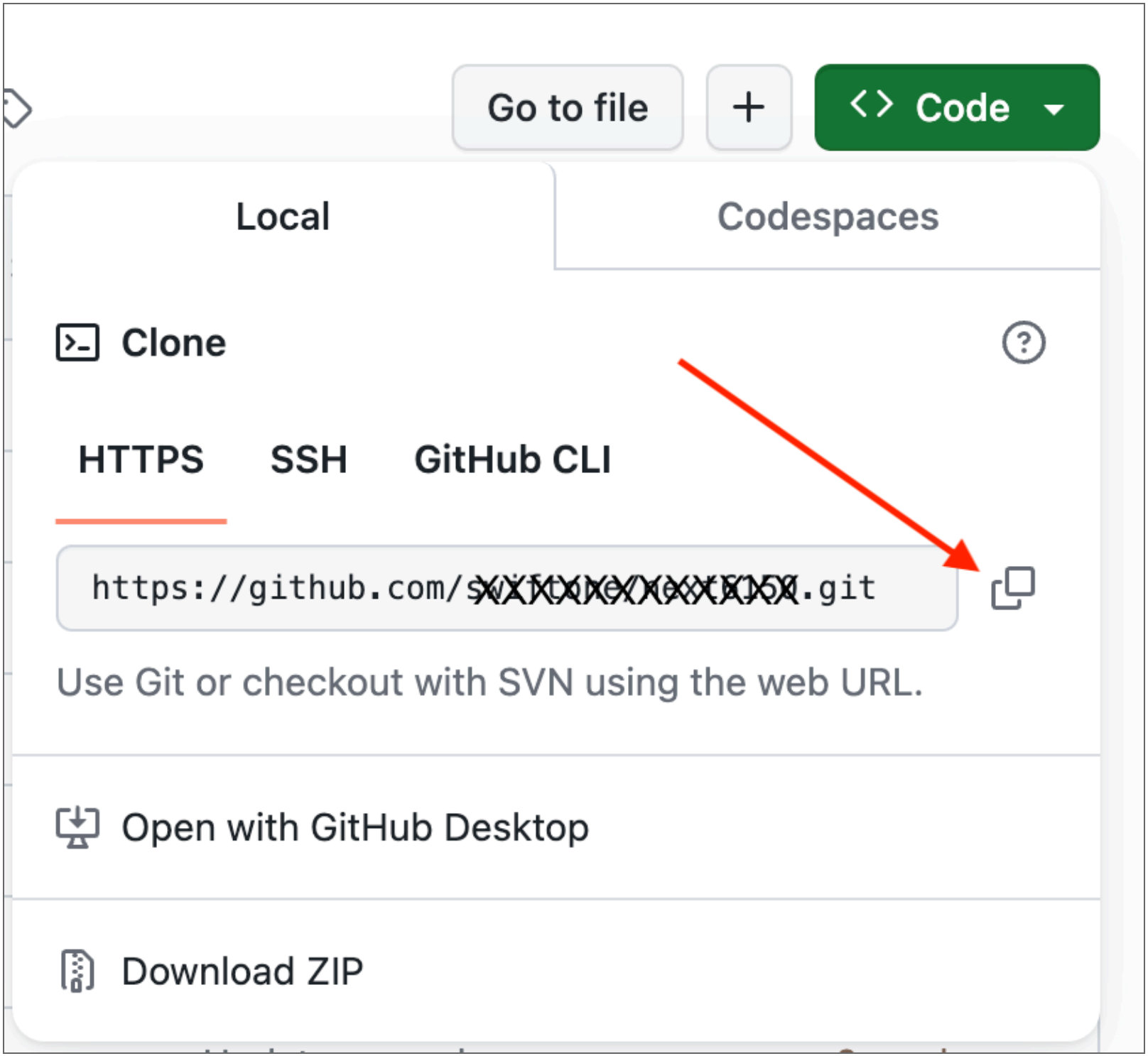
- You must have installed `git` on your computer
- **https://github.com/git-guides/install-git**
- Next step **assumes you have installed git**

Finding the Repo URL to Clone it

Find this:



Copy the Repo URL


Go to file
Code
Local
Codespaces
Clone
HTTPS
SSH
GitHub CLI
https://github.com/
.git
Use Git or checkout with SVN using the web URL.
Open with GitHub Desktop
Download ZIP

Clone at the command line

- Run this command
  - **in the folder that will hold your repo folder**
  - Ex: `/Users/GoodStudent/courses/`
- `git clone PASTE_REPO_URL_HERE`
  - Do NOT type "PASTE_REPO_URL_HERE" 🤦
- This creates a new folder
  - "in" the folder you ran command in
  - New folder is named for your repo

You have a copy of the repo on your computer

- Your **local** copy of the repo
  - Github is a **remote** copy of the repo
  - Instructor will also have a **remote** copy
- You have all the folders and files from the repo
  - The **main branch**

Command: `git status`

- run `git status` **in the repo directory**
- See where it tells you **current branch**
    - Ex: `On branch main`
    - VERY IMPORTANT to know!
    - Everything harder when in the wrong branch

Command: `git checkout -b NEWBRANCH`

- Run: `git checkout -b testing`
- Run: `git status`

Two things happened

- Created a **new branch**
    - Based on the branch we were in (**main**)
- We switched into the new branch (**testing**)
    - `On branch testing` (No longer `main`)

We "**passed**" the new branch name (**testing**) to the command

Branch names

- The name of a branch is "just" a name
  - No magical properties
  - `main` a common name
    - In past: `master`, may still see occasionally
- **Names are IMPORTANT**
  - **Programming is Communication**
  - You AND team need to understand, remember, and use
- But Name doesn't have any "magic"
- "`testing`" here is a name chosen for meaning
  - But not any special behavior

What does the testing branch give us?

- Nothing is different yet
- **testing** branch contents same as the **main** branch it was created from
    - For now
- A **branch** tracks **commits**
    - **commits** are collections of changes
    - Changed from some **base** branch

Let's make a change in **testing**

Making a change in `testing` branch

- Create a file named `changed-file.txt`
  - Use whatever editor you want
  - **Make sure the file exists IN repo folder!**
  - Contents: "Hello World"
- Run `git status`
  - On branch **testing** (not `main`)
  - `changed-file.txt` listed as **Untracked file**

What is an Untracked File?

- Means the file isn't (yet) tracked by git!
    - Won't be saved or tracked
- We want to track this file
- But first lets see the impacts

Command: `git checkout SOMEBRANCH`

- Run: `git checkout main`
- Run: `git status`
  - Now on branch `main` (not `testing`)
  - `changed-file.txt` still listed
    - Untracked files not part of a branch

`git checkout SOMEBRANCH` (no `-b`)

- **switches** to an *existing* branch
- Does NOT create a new branch (that requires `-b`)

Switch back to branch "testing"

- Run: `git checkout testing`
- Run: `git status`

Now on branch `testing`

If we were on `main` the commands we are about to run would be done on `main`

We never want to make random changes to `main`

- `main` is for our team to share
- Should always be a working, approved version

Command: `git add SOMEFILE`

- Run: `git add changed-file.txt`
- Tip: Tab key can usually autocomplete filenames
  - Ex: `git add ch` then hit Tab to autocomplete
  - Programmers are Lazy, it's a Virtue for us
- Run: `git status`
  - On branch `testing`

`changed-file.txt` listed as "Changes to be committed"

- We call this **staged**
  - Not yet **committed**

Command: `git commit -m"SUMMARY"`

Run: `git commit -m"Testing Git"` Run: `git status`

- On branch `testing`
- "nothing to commit, working tree clean"

Command: `git log`

- Run: `git log`

Lots of info!

- But you can see your "Testing Git" at the top
- Every commit is listed
    - Over time, log gets long!

Commit Messages

- You ran `git commit`
  - Passed a **commit message** (using `-m`)
- A **commit message** is a summary of the changes
  - Should be brief
  - Must be *helpful*
    - Actually describe *purpose* of changes
  - Reader won't know which project/assignment!
    - You must provide any context
- Team will use commit messages to understand
- See real-world commit messages:
  - **https://github.com/facebook/react/commits/main/**

I'm serious

Programming is largely about communication

- About organizing large, complex code collections

Your workplace WILL expect **quality commit messages**

- Important/Helpful to practice now
- Messages DON'T have to be INTERESTING
- Messages DO have to describe purpose

Change is only in branch `testing`

- Run: `git status`
  - See: On branch `testing`
- Run: `ls` (or `dir` on some Windows)
  - See: `changed-file.txt`
- Run: `git checkout main`
- Run: `git status`
  - See: On branch `main`
- Run: `ls` (or `dir` on some Windows)
  - See: **No** **`changed-file.txt`**!

Where did our file go?

We created `changed-file.txt`, where is it?

- It was there until we switched branches!
- Run: `git checkout testing`
- Run: `ls` (or `dir` on some Windows)
    - See: `changed-file.txt`

When you make, add, and commit changes to a branch

- Changes are only to THAT branch

Notice: Branches are NOT directories

- Course Repo has directories (folders)
- Assignments say to create specific branch names
    - These are often similar to directory names
    - But **branches are NOT directories!**
        - Common misunderstanding

"Switch to SOMEBRANCH"/"Checkout SOMEBRANCH"

vs

"Go into SOMEFOLDER"/"Are you in SOMEFOLDER"?

How do we update `main` branch?

If we never make changes in our `main` branch

- `main` is supposed to be a shared, approved branch
- How do we get updates into `main`?

This is a process!

- Multiple approaches exist
- Course emulates most common model
- Most employers will do it this way

Step 1: Feature Branches

All changes should be made in a branch

- A branch *specific* to that particular set of changes
- A **feature branch**

All of our assignments are **each in different branches**

- Will follow process to merge into `main`
- Same process as most workplaces
- You must manage repo over time
- This requires EFFORT (for you AND me)
    - But important skills for webdev

Assignments as Feature Branches

This course has one repo/student

- Each Assignment is a feature branch

Most other courses will NOT do this

- It's a big pain to keep everyone in sync
- Instead they may have a repo for each assignment
    - If using `git` at all
- Those courses aren't about teaching `git`
    - Neither is this one
    - So this is all you get (all you `git` 🤣)

Creating your Feature Branch

- A branch is **based** on another branch
  - Collects changes from that **base** branch
  - Changes since that "point in time"
- Often want those changes to be based off of `main`
  - Specifically, the latest version of `main`
  - Most up to date shared, approved version
  - For this course, not "Often". **Always**

Basing Feature Branch off of main

The easiest way to keep things straight

- Change to `main`
- **Pull** any approved, merged updates from Github
  - More on this soon
- Create your feature branch
  - **Must be in `main` branch first!**
  - `git checkout -b NEWBRANCH`
- **Always be in branch before adding/committing**

Adding/Committing Files

- `git` works like a powerful "Undo" system!
    - Lets you revert/see any previous **commit**
    - But that's only helpful IF you committed!
- When do you commit?
    - An advanced programming skill
    - This course doesn't care
        - Can commit just once when done
        - Can commit many times as you code
    - BUT:
        - Each commit message must be **quality**

What about advice/best practice?

Best Advice:

- Commit when you have completed USEFUL work
- Generally code should RUN
    - Not necessarily complete
    - But it runs/renders
- If you can't make a good commit message
    - If changes don't complete a purpose
    - Bad time to commit

More info: **https://jasonmccreary.me/articles/when-to-make-git-commit/**

Adding files to commit

- You can add files one-by-one
  - `git add filename`
- You can also add many files at once
  - `git add *.html`
  - `git add SOMEFOLDER`
    - adds all changed files in SOMEFOLDER
  - `git add .`
    - adds all changes in current directory

Adding many files

- use `git status` to verify files BEFORE commit

Seriously - Check **before** you accidentally commit hundreds/thousands of generated files

- It happens every semester
- It happens on the job
- Much easier to fix before commit
- If you fix before commit, coworkers never know

Oh no! I added files I don't want!

Not a problem! `git status` says what to do:

- `git restore --staged FILENAME`
- `git restore --staged SOMEFOLDER`

Verify things fixed using `git status`

- Always confirm current branch!
- All needed files added!
- No unwanted files added!

Other issues, see **https://ohshitgit.com/**

Cleaning your commit history

Some programmers have Thoughts and Feelings

- Including some Employers (ex: Amazon)
- They feel you should clean up your commits
- Your collection of commits
    - May not be what you want in the shared repo
- Others...simply don't share these concerns

Either way, that's a more advanced concept

- Way outside the scope of this course

Ending Step 1: Ready to Push

Once you have:

- All changes made and working
- Files **add**ed to feature branch
    - Based off of `main` branch
- Changes **commit**ted to feature branch
    - Using **quality commit message**

We are ready to **push the feature branch**

- And **create a pull request** (PR)

Side Note: Ignoring certain files

Often some files and directories we NEVER WANT

- Files created by our OS
  - `.DS_Store`
- Temporary Files
  - `*.tmp`
- Config files for personal editor
  - `.idea`
  - `.vscode`
- Files specific to installations
  - `node_modules/`

`.gitignore` file lists patterns to ignore

`git` automatically "ignores" files based on `.gitignore`

- File impacts directory containing that `.gitignore`
    - And any directories "inside" that directory
        - "subdirectories"/"subfolders"
- Repo might contain many `.gitignore` files
    - Each affecting different subdirectory "trees"
- `.gitignore` file itself just a file tracked by git
    - Like any other file
    - Edit, add, commit it

`.gitignore` File Content

- Each line is a "pattern" or a comment
    - Text to match file/directory names against
    - May include wildcard characters
- Read more: **https://git-scm.com/docs/gitignore**

Example contents:

```
# sample minimal .gitignore file
node_modules/
.DS_Store
```

Step 2: Pushing Feature Branch

You **push** a branch from **local repo** to a **remote repo**

- We want to push to Github's remote repo
- For us, this remote repo is called **origin**
    - Because this repo was cloned from it
- `git push REMOTENAME BRANCHNAME`
    - Ex: `git push origin testing`

Results of Pushing to Github

You should see some text that looks like:

```
remote: Resolving deltas: 100% (1/1), completed with 1 local object.
remote:
remote: Create a pull request for 'testing' on GitHub by visiting:
remote:      https://github.com/SOME_ORG/YOUR_REPO/pull/new/testing
```

- This is what you want to see
- You will want to follow that link!

Where we are so far

- You finished your work in a **feature branch**
    - on your **local repo**
- You **pushed** your **feature branch** to a **remote repo**
- Now Github has `main` AND `testing` branches
- Want to **merge** the changes in `testing` into `main`
- Before we do so, we want want **approval** to do so
    - A "**code review**"
    - Most employers will have this
    - EVERYONE has their changes reviewed

We will create a "Pull Request" (PR)

- A **pull request** to **merge** a branch into another
  - Often merged to `main`
  - But could be `next-release`
  - Or a specific version
  - Varies by workplace
- This course: Always PR to merge to `main`
- Follow that link from when we pushed
  - Or go to Github on web
  - Will have a "Compare & pull request" button

Before Creating the PR

- This web screen offers to create PR
  - We will, but FIRST....
- Review the changes (scroll down)
  - Any unexpected files?
  - Do changes look to be correct?
- Last chance to see issues before coworkers see!
- "Title" of PR autofilled by commit message
- Add "Reviewers"
  - Not "Assignees"
- "Unable to merge" because approval required
  - Expected, not a problem

Create the Pull Request (PR)

- Not a "Draft", Create the actual PR
- Reviewers will be informed
    - They will need to Approve
    - They may comment
    - They may request changes before Approving
- For this course: Evaluating work quality
- For a job:
    - Less: "Will this work?"
    - More: "Do we want to maintain this?"

Now you wait

Once the PR is in, there's nothing further you can do

You CAN work on other things

- Remember to switch away from feature branch!
- I recommend changing to `main` immediately
    - Even if you're not (yet) working on anything
- Can work on and switch between many branches
    - Be sure to add/commit to correct branch!

Making Changes!

- Maybe reviewer requested changes
- Maybe you noticed something wrong

Update your local feature branch

- Same branch, not a new one!
- Add/commit changes
    - w/Good commit messages!
- Push the update branch to `origin` (Github)

PR will automatically reflect changes!

- Because PR was to merge BRANCH
    - Not specific commits
- No need to Close PR and create a new one

Approval! Now What?

Approval doesn't change any code

- Doesn't change repo
- Just ALLOWS a "**merge**"
- For Course: TA or myself will merge on approval
- For Job: YOU will have to merge
    - Go to Pull Request in Github and hit "Merge"

The PR Merge Race

- Merging a PR can be a race
- Other people are merging too
    - They may have incompatible changes
        - Even if it only requires a small tweak
    - First merge wins
        - Others will need to update before merge
        - Which means a second race to merge
- No race if all changes are compatible
- No merge races for this course
    - But does happen on the job

After the merge

- The merge updated `main` on Github
- Does not magically update your **local repo**
- You need to grab the latest `main`
- Run: `git pull origin main`
    - "pull any changes from origin `main` branch"
    - **Be sure to be in `main` branch**
    - Otherwise changes come to current branch!
        - Can be very confusing

Understand the flow

- You have a `main` branch
  - A shared, approved version of the code
- You create a feature branch
  - Based on the current `main` branch
- You make changes in feature branch
  - Rest of team not impacted by your changes
  - They are relying on `main`
- You create a PR to merge your changes to `main`
  - Teammates review and approve
- You update `main` to include approved changes

Why such a complicated process?

- Can revert to any previous approved version
- Can track changes over time from many people
- `main` only includes reviewed/approved changes
- Can work on many features in parallel
- Can share batches of incomplete features
    - Can push and pull from other branches
    - Approval needed only to alter `main` branch
        - Or whatever branch(es) your team uses

Why such a complicated process for this course?

- Because you will need it on the job!
    - Not a lot of places to get this experience
- Not just web development
    - Almost all programming fields
    - Often including adjacent disciplines
        - Documentation
        - Design
        - Project Management

Tips for Using Git

- Use tutorial links in course readings
- Make heavy use of `git status`
    - **Be sure which branch you are in!**
- Read error messages!
    - **If you get an error, STOP**
    - Do NOT just continue through more steps
    - Often end up changing wrong branch
- **Practice quality commit messages**
- Confirm PR contents before Creating Pull Request

More Git Tips

- `git` _IS_ confusing
  - Easier to work on understanding it now
  - Otherwise you will struggle on the job
  - Really is complex: Don't let it depress you
- `git` is harder if you try to skip it
  - A lot of tech can be learned a little at a time
  - `git` is hard to passively learn a little at a time
  - Like CSS and Regular Expressions
    - We also cover these

Command line pays off

There are valuable GUI tools for working with `git`

- Including integration with most code editors

But for many commands, command line is best

- Most documentation will show command line
- Other web tools also expect command line

Using Graphical git tools is fine!

- You will want to understand command line too