Git Workflows, GitHub, and More Git

...this Git thing seems important

Preface I

1. Enhance Learning Experience:
    a. These slide decks, being dense, aim to provide comprehensive review material for Git, a tool frequently used in and outside the course.
2. Adaptation and Complementary Resources:
    a. Understand that the field evolves rapidly, so explore additional resources to supplement your learning and stay up to date.
    b. Our goal is to support effective teaching and learning, so feel free to seek clarification and guidance from your instructor.

Preface II

- Need-to-Know Git Workflows:
  - These workflows are foundational for navigating and traversing projects.
  - Consider them as repeatable and templated working practices.
  - They provide a good, functioning base for individuals and team members with Git.
  - Other workflows exist, but ours serve as a reliable foundation.
- Key GitHub Features:
  - Leveraging these features enhances collaboration and project management:
    - Organizations: Create collaborative environments.
    - Collaborators: Invite team members to contribute.
    - GitHub Projects: Manage projects efficiently.
    - Issues: Track tasks, bugs, and feature requests.
    - Branch Protection: Secure the main branch with passwords.
    - Commits and Branches: Easily view and manage code changes.
    - GitHub Gists: Share syntax-highlighted code snippets.
    - GitHub Pages: Deploy static websites effortlessly.

Preface III

- Learning Curve
  - Despite the pre-work and previous slide decks, there is still much to learn about Git and GitHub and how they work together.
  - It's normal to encounter productive struggles as you navigate the remaining learning curve of Git workflows.
- Spaced-Repetition for Retention
  - Every lesson and skill in this course requires spaced-repetition to ensure long-term retention.
  - Revisiting and practicing the concepts regularly will solidify your understanding.
- Slide Deck Focus
  - The primary focus of this slide deck is on commonly used commands and scenarios to develop and refine independent and team-based Git workflows.
  - Topics covered include branching, merging, conflicts (via stash and merges), and essential features of GitHub for Assignment 1.

Slide Deck Outline 1
Independent Git Workflow with Upstream Repository (via GitHub)

- Initializing the Workflow:
  - Initialize Git: git init
  - Create Upstream Repository: Set up a repository on GitHub.
  - Connect to Upstream: git remote add <name> <URL>
- Working on the Workflow:
  - Add and Commit Changes: git add, git commit (with semantic naming convention)
  - Push Changes: git push --set-upstream <name> <branch>
- GitHub Integration:
  - Utilize GitHub Issues for task management.
  - Leverage GitHub Projects for project organization.
  - Deploy via GitHub Pages.

Slide Deck Outline 2

Golden Git Team-Based Workflow with Established Project (via GitHub)

- Before Starting a New Feature:
  - Update Main: Pull changes from upstream repository to local main branch.
  - Create Feature Branch: git checkout -b <feature-branch> (with semantic naming convention)
  - Switch to Feature Branch: git checkout <feature-branch>
- Making Changes:
  - Make changes in your workspace, stage them, and commit them to the feature branch.
- Pushing Changes:
  - Push Branch: git push origin <feature-branch>
  - Create and Accept Pull Request: Collaborate and review changes.
  - Merge Changes: Merge feature branch into upstream main branch.
  - Update Local Main: git checkout main, git pull
- Golden Workflow Benefits:
  - Ensures seamless collaboration and integration of features.
  - Facilitates team-based development and efficient project management.

Slide Deck Outline (continued)

3. More Git

a. Glossary
   i. HEAD
   ii. Detached HEAD
   iii. Origin
   iv. Main vs Origin/Main vs Origin Main

b. Some Brief Gotchas
   i. Pushing and Pulling
   ii. Nested Submodules
   iii. Conflicts
   iv. .gitignore

1) Independent Git Workflow

Independent Git Workflow with an Upstream Repository (via GitHub) and GitHub Issues as well as GitHub Projects and GitHub Pages

Independent Git Workflow I

- Goal
  - Build a local repository and deploy it to GitHub Pages, following professional practices.
- Composed on Independent Parts
  - Understanding each part individually will help in this scenario and other scenarios.
- Benefits:
  - Following professional practices
  - Understanding each step contributes to general Git Knowledge

Independent Git Workflow I

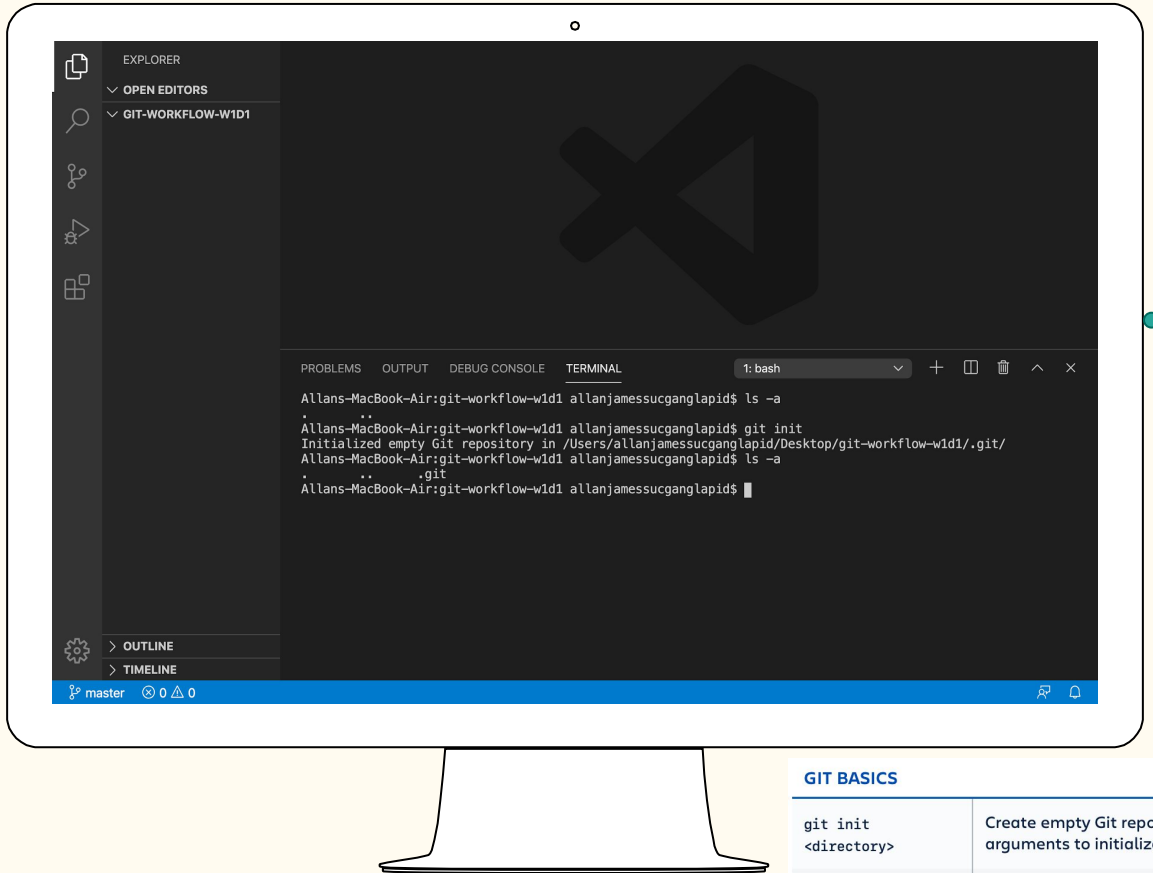
- List of Steps:
  1. Initialize Git: git init (after creating and changing into the directory)
  2. Create Upstream Repository: Set up a repository on GitHub.
  3. Connect to Upstream: git remote add <name> <URL>
  4. Add and Commit Changes: git add, git commit (with semantic naming convention)
  5. Push Changes: git push --set-upstream <name> <branch>
  6. Create Issues: Manage tasks or developer tickets.
  7. Manage Branches: git branch, git checkout, git checkout -b, git branch --all
  8. Handle Pull Requests: Collaborate and review changes.
  9. Automate PR and Issue Closing: Sync up and automate the closing of pull requests with issues.
  10. Deployment via GitHub Pages: Deploy the project on GitHub Pages.

git init	✅
create upstream repository	TBD
git remote add	TBD
git add	TBD
git commit	TBD
git push	TBD
create issues	TBD
git branch	TBD
git checkout	TBD
create pull requests (PR)	TBD
sync and close PRs	TBD
deploy	TBD

Create repositories

When starting out with a new repository, you only need to do it once; either locally, then push to GitHub, or by cloning an existing repository.

```
$ git init
```
Turn an existing directory into a git repository

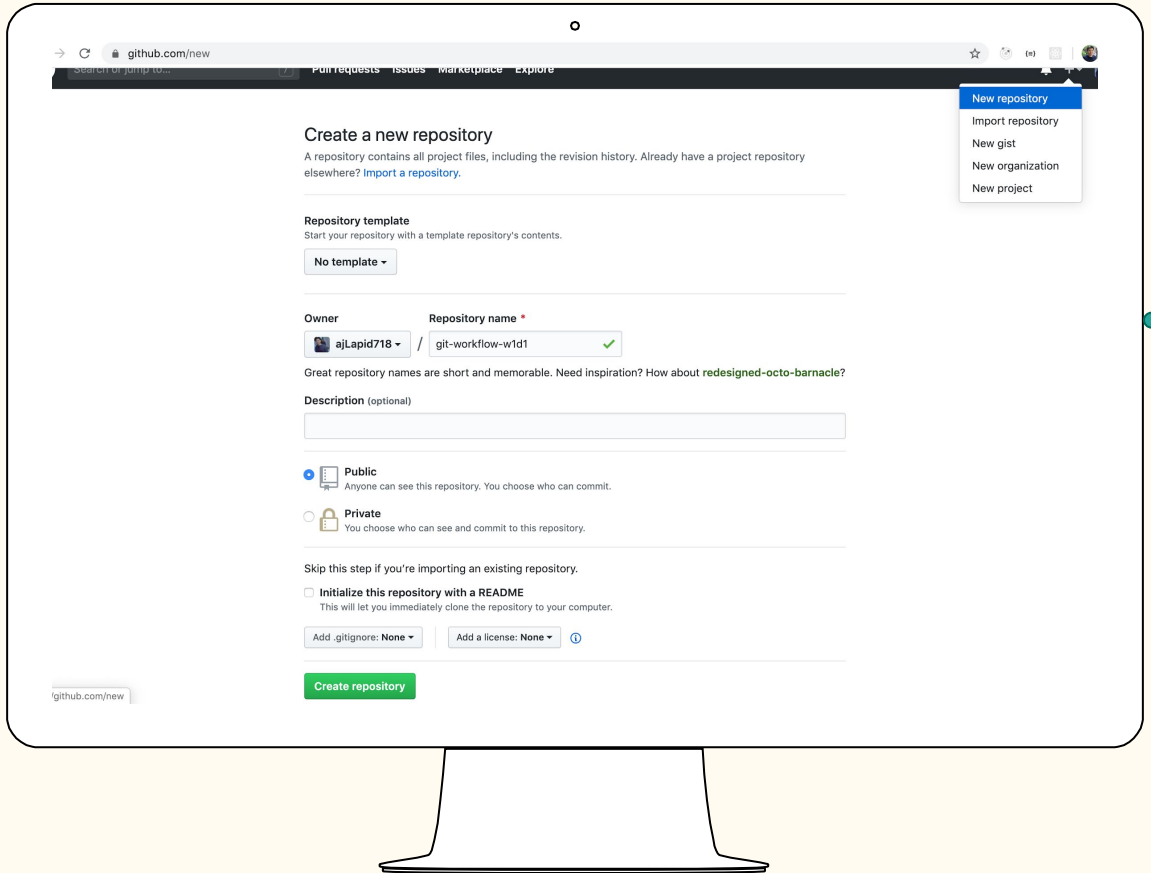


ls -a allows you to list all folders and files, even hidden ones

GIT BASICS

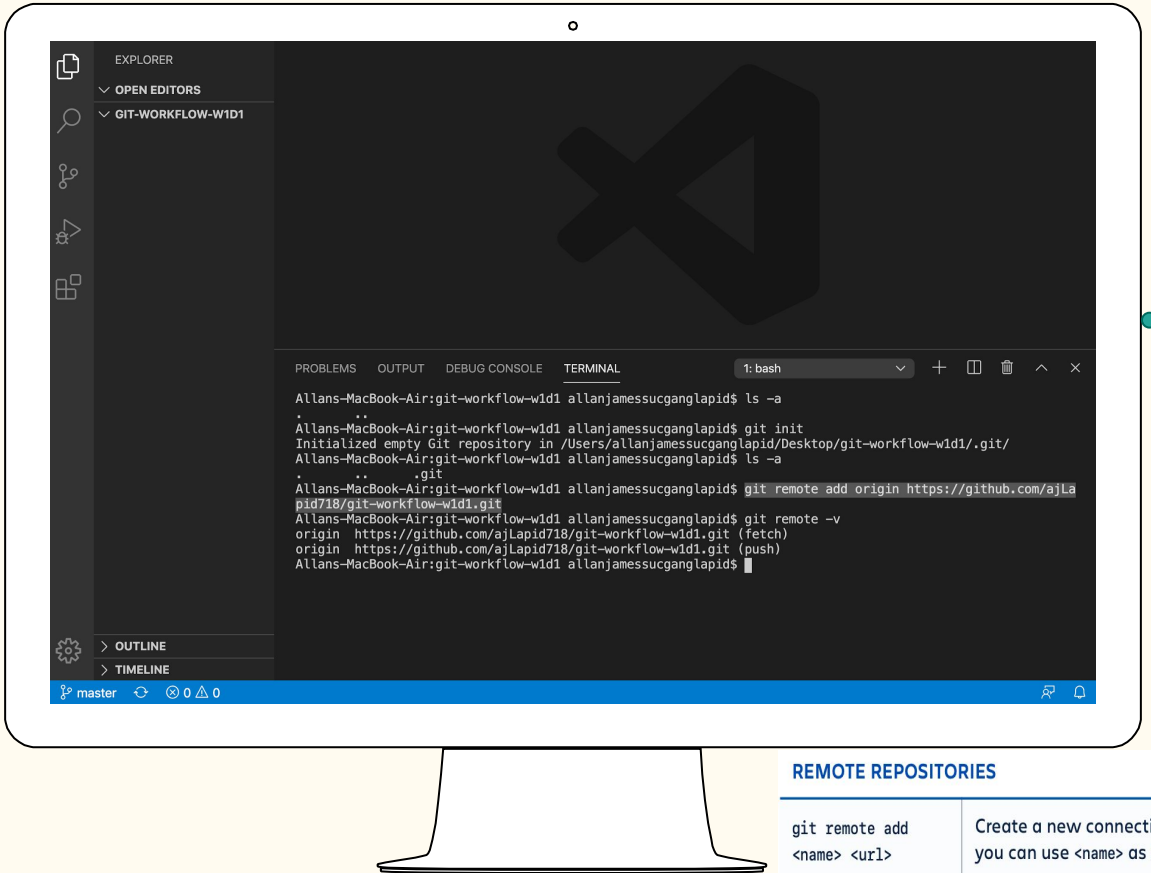
`git init <directory>`	Create empty Git repo in specified directory. Run with no arguments to initialize the current directory as a git repository.

git init	✅
create upstream repository	✅
git remote add	TBD
git add	TBD
git commit	TBD
git push	TBD
create issues	TBD
git branch	TBD
git checkout	TBD
create pull requests (PR)	TBD
sync and close PRs	TBD
deploy	TBD



click on “Create repository” at the very bottom, which will provide helpful instructions afterwards

git init	✅
create upstream repository	✅
git remote add	✅
git add	TBD
git commit	TBD
git push	TBD
create issues	TBD
git branch	TBD
git checkout	TBD
create pull requests (PR)	TBD
sync and close PRs	TBD
deploy	TBD

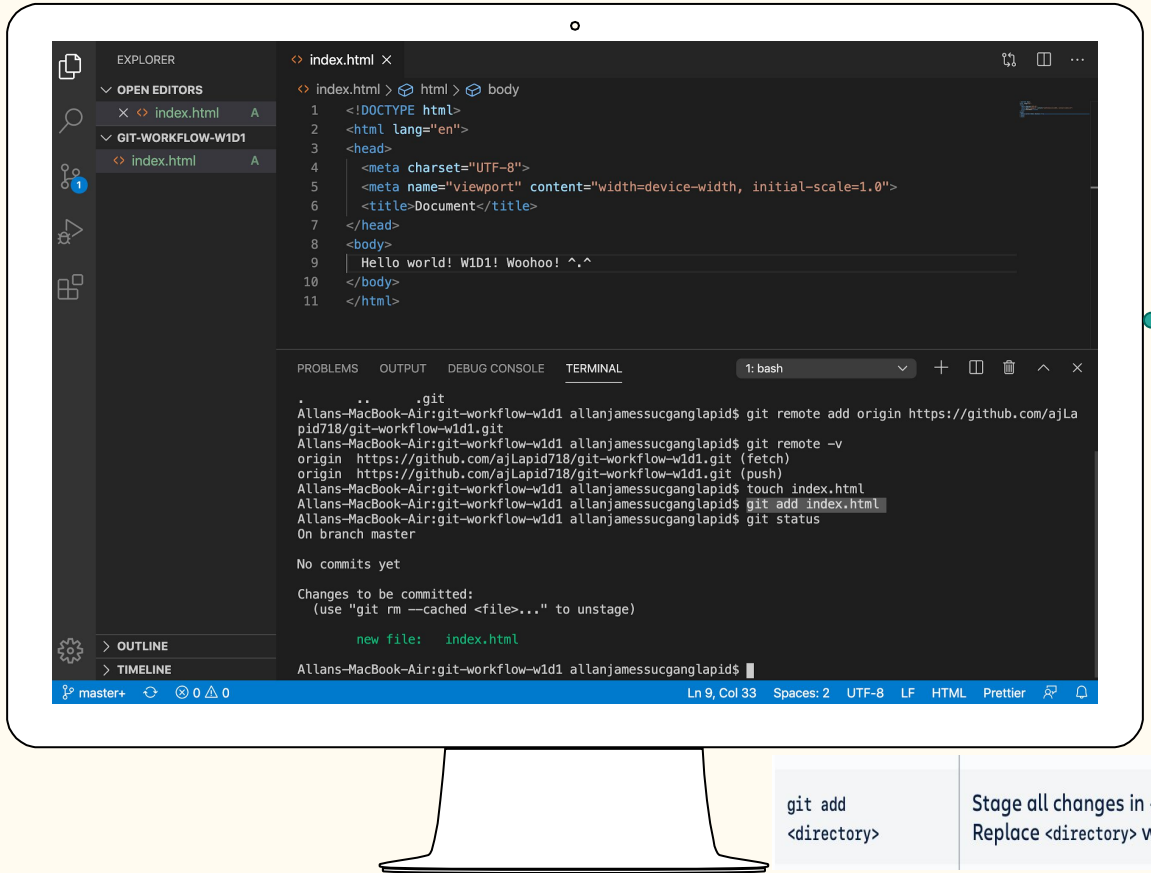


git remote -v will list the URL of the upstream repository (it is the URL to your new remote repository with a .git extension)

REMOTE REPOSITORIES

`git remote add <name> <url>`	Create a new connection to a remote repo. After adding a remote, you can use `<name>` as a shortcut for `<url>` in other commands.

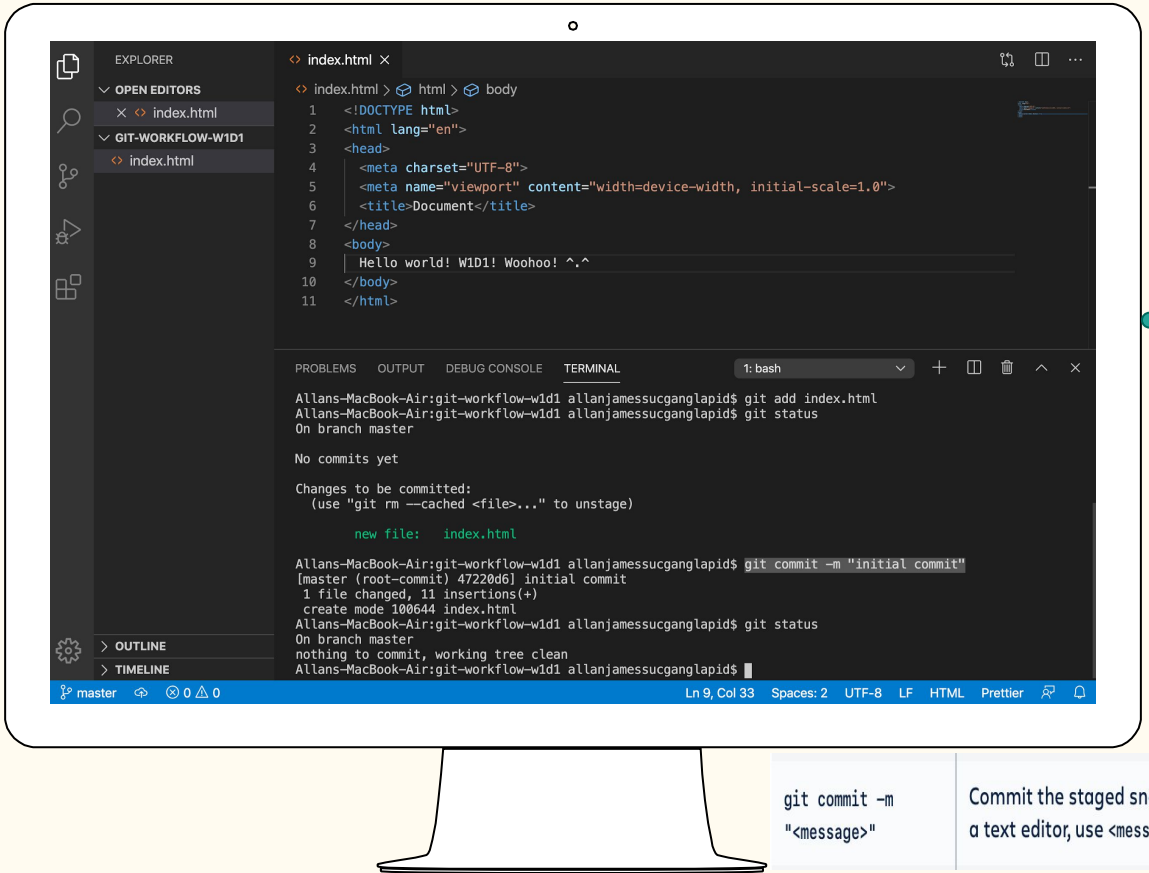
git init	✅
create upstream repository	✅
git remote add	✅
git add	✅
git commit	TBD
git push	TBD
create issues	TBD
git branch	TBD
git checkout	TBD
create pull requests (PR)	TBD
sync and close PRs	TBD
deploy	TBD



git add will move the changes from workspace to index

`git add <directory>`	Stage all changes in `<directory>` for the next commit. Replace `<directory>` with a `<file>` to change a specific file.

git init	✅
create upstream repository	✅
git remote add	✅
git add	✅
git commit	✅
git push	TBD
create issues	TBD
git branch	TBD
git checkout	TBD
create pull requests (PR)	TBD
sync and close PRs	TBD
deploy	TBD



git commit will move the content of the index to the local repository level aka saving

`git commit -m "<message>"`	Commit the staged snapshot, but instead of launching a text editor, use `<message>` as the commit message.

What are some commit conventions?

Commit Conventions for Effective Git Workflow

- Various Commit Conventions Cover:
  - Length of a commit message
  - Handling of the subject versus the body
  - Capitalization rules
  - Punctuation rules
  - Tense (past, present, future, imperative, etc.) of the commit message

- Importance of Team's Agreed Commit Conventions:
  - Maintain a clean and consistent commit history.
  - Improve code readability for others who view your code.
  - Facilitate effective collaboration within the team.

Commit Conventions for Effective Git Workflow (Contd.)

- Other Commit Conventions Include:
  - Emphasizing either the "what" has changed or the "how" something has changed
  - Categorizing commit messages
  - Utilizing emojis to describe commit contents

- Benefits of Following Commit Conventions:
  - Ensures a standardized approach to commit messages.
  - Promotes clarity and understanding of code changes.
  - Helps in tracking and managing project history.
  - Enhances the overall quality and maintainability of the codebase.

Effective Commit Messages

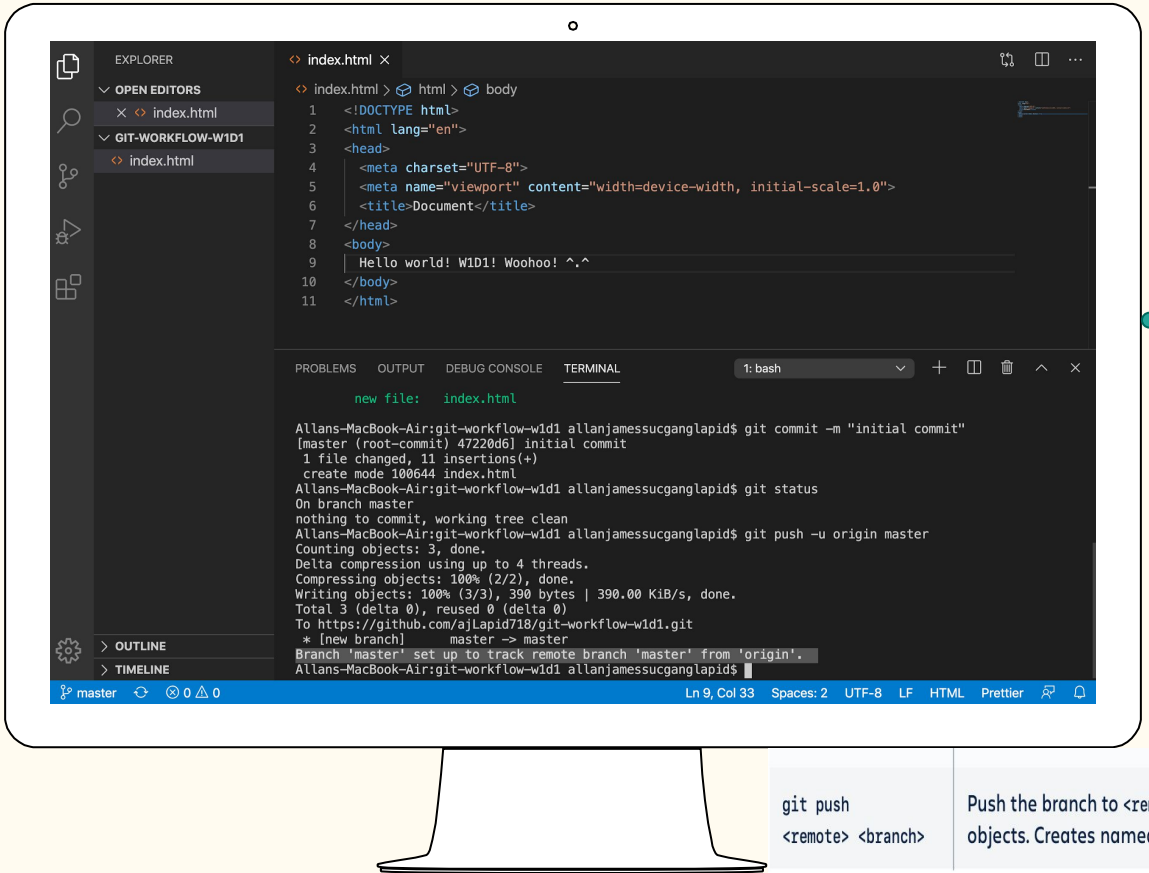
- Visualizing Commit Messages:
  - Focus on "what the commit does" rather than "what the developer performed."
    - Example: If we reset a commit that changed text color to red, the commit message might be "apply red to all text."
  - Resetting from that commit communicates the idea of "we no longer want to apply red to all text."
- Importance of Descriptive Commit Messages:
  - Prioritize clarity and specificity in commit messages.
  - Avoid vague phrases like "fixed the bug" and be more descriptive.
  - Prioritizing "what a commit brings to the table" helps maintain understanding.
- Best Practices for Commit Messages:
  - Be consistent and descriptive in writing commit messages.
  - Consistency ensures a standardized approach.
  - Descriptive commit messages aid in understanding code changes.
- Benefits:
  - Easier navigation of projects and understanding commit history.
  - Facilitates collaboration, code review, and troubleshooting.

General Information

- **Golden Rule #1: Only commit working code!**
  - Committing only working code is important to maintain a stable codebase and ensure that each commit represents a functional state of the project. This allows for easier collaboration, smoother development workflows, and the ability to revert back to a known working state if necessary.
- **Golden Rule #2: Commit early, and commit often!**
  - This means making frequent, small commits to the version control system. This practice allows for better tracking, collaboration, and easier identification of changes, enabling developers to work incrementally and have a detailed history of their progress.
- **Golden Rule #3: Keep commits specific and maintain scope!**
  - Clear accountability: Specific commits make it easier to identify the purpose and responsibility of each change. This helps in understanding who made a particular modification and why, making collaboration and issue tracking more efficient.
  - Manageable changes: Keeping commits focused on a specific scope allows for better control and management of changes. It enables easier code review, debugging, and the ability to revert or modify specific features or fixes without affecting unrelated parts of the codebase.

…so *that's* what we're supposed to do after `git commit -m " "`

git init	✅
create upstream repository	✅
git remote add	✅
git add	✅
git commit	✅
git push	✅
create issues	TBD
git branch	TBD
git checkout	TBD
create pull requests (PR)	TBD
sync and close PRs	TBD
deploy	TBD



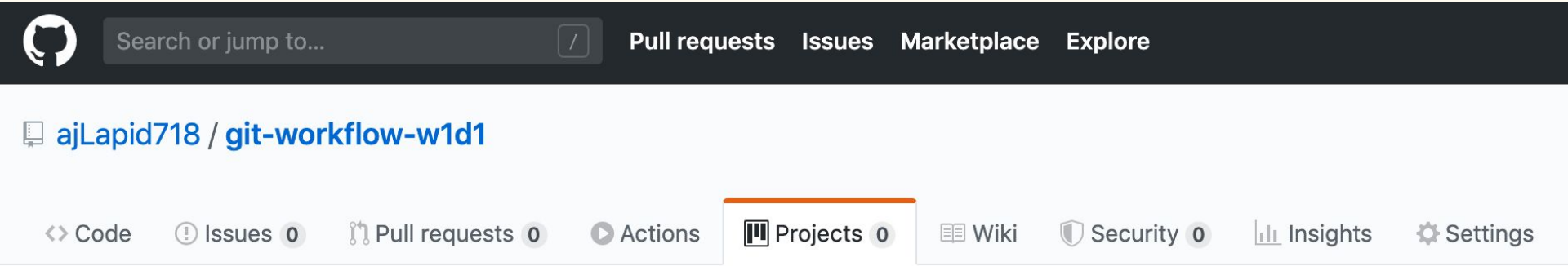
git push -u origin main will set and track the remote upstream branch aka the "counterpart" to your local branch --- anytime you make a change to this branch, you can from then on just do git push

`git push <remote> <branch>`	Push the branch to `<remote>`, along with necessary commits and objects. Creates named branch in the remote repo if it doesn't exist.

Intermission: Setting Up GitHub Projects

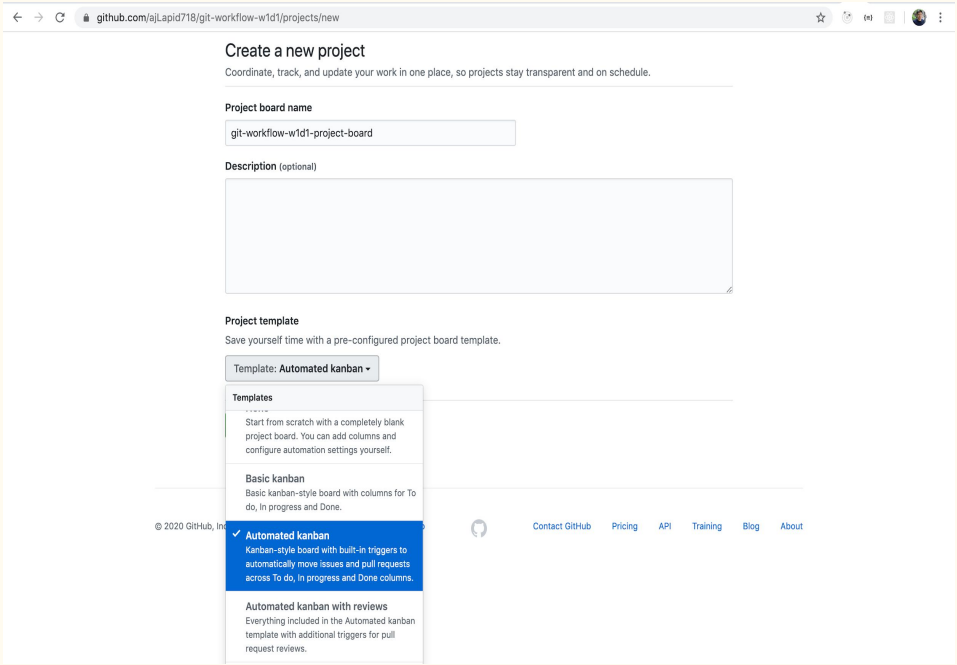
Setting Up GitHub Projects

1) Navigate to the “Projects” tab and click on“New Project”



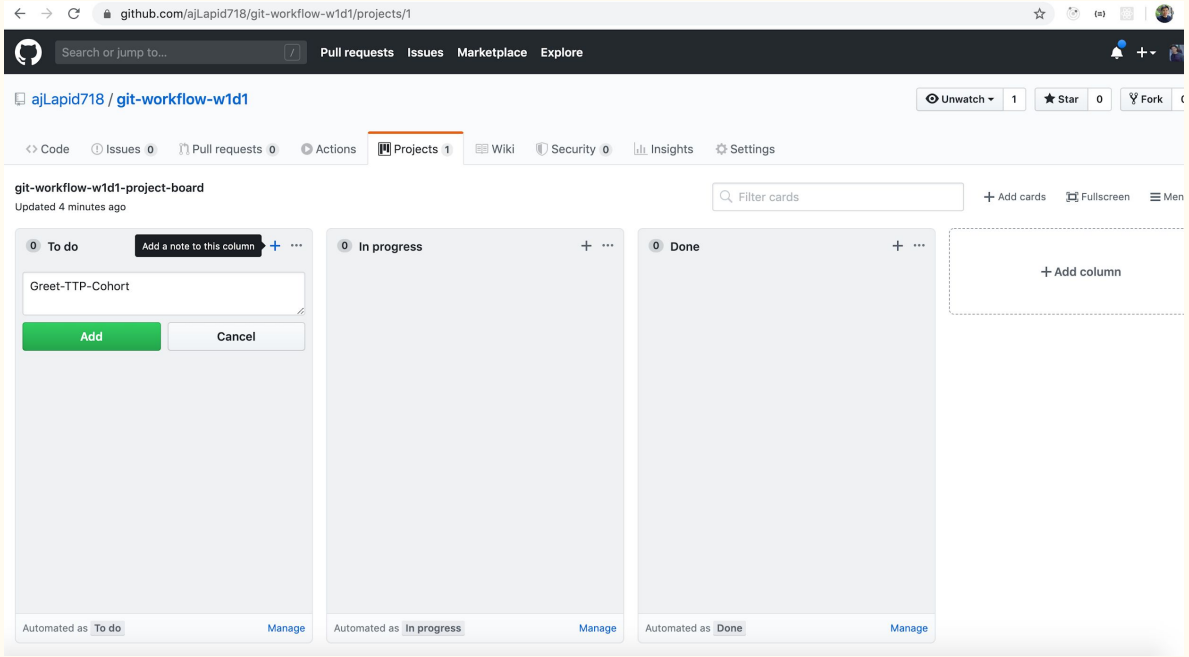
Setting Up GitHub Projects

2) Name the “board”, select the “Automated Kanban” template, and click “Create Project”



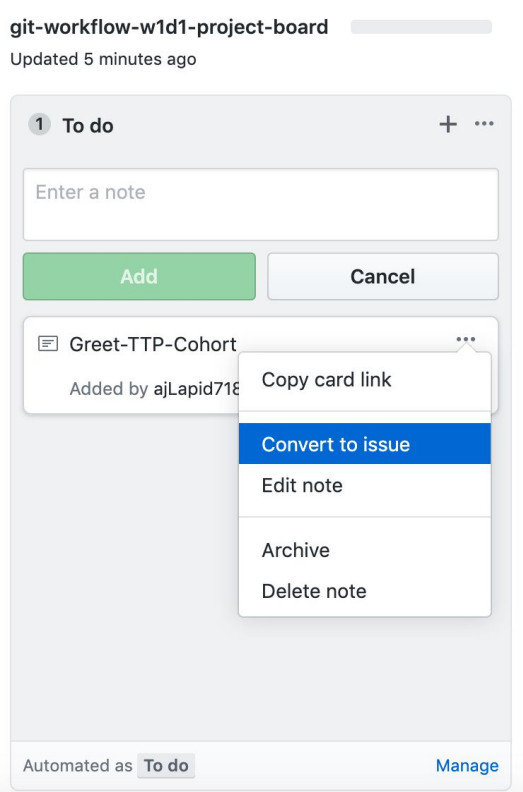
Setting Up GitHub Projects

3) When on the main board, click on the "+" sign in the "To Do" column, and then "Create a note" (this is not a fully-fledged issue yet)



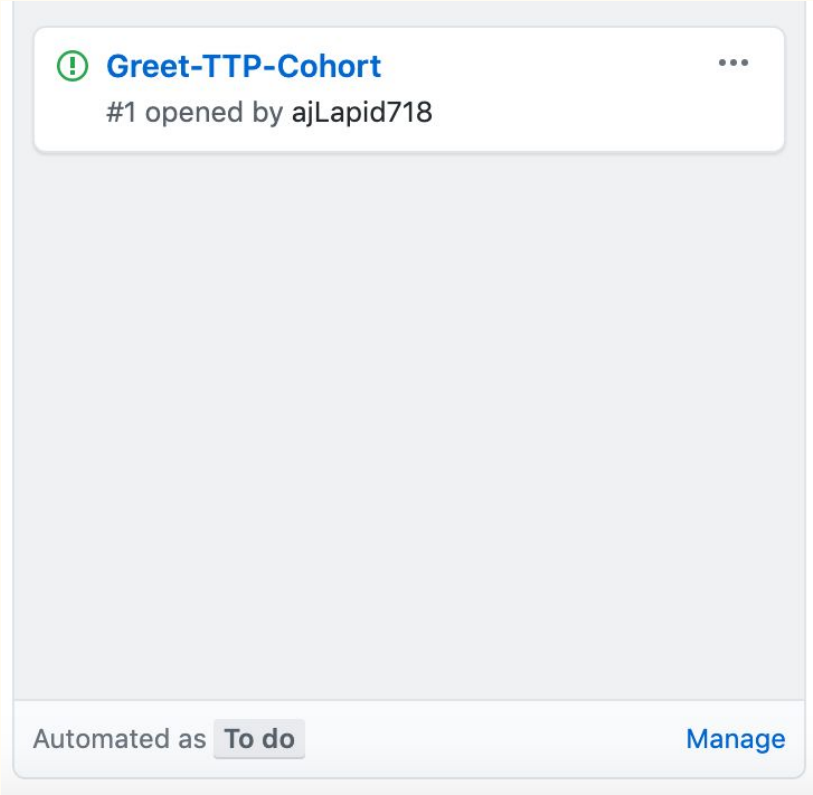
Setting Up GitHub Projects

4) Click on the "..." symbol of the newly created note, and then click on "Convert to issue" in order to have a full-fledged issue



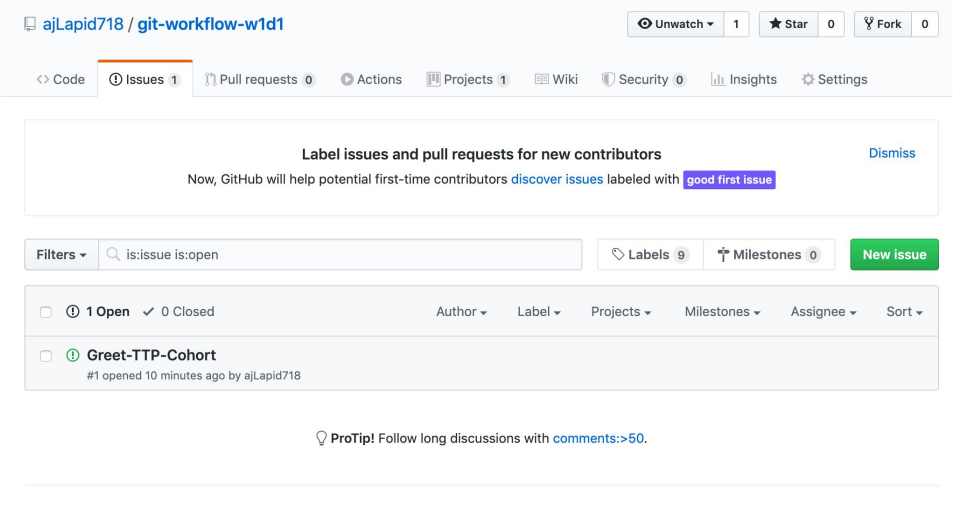
Setting Up GitHub Projects

5) Now you have an issue (with a #1 assigned to it)!
   a) Issues in version control systems are like developer tickets, representing specific tasks or work items
   b) Despite the term "issue," in version control systems, issues are not necessarily problems but actionable items for developers to address and improve the project.
   c) Issues serve as tasks on the agenda, outlining work that needs to be completed in the project.



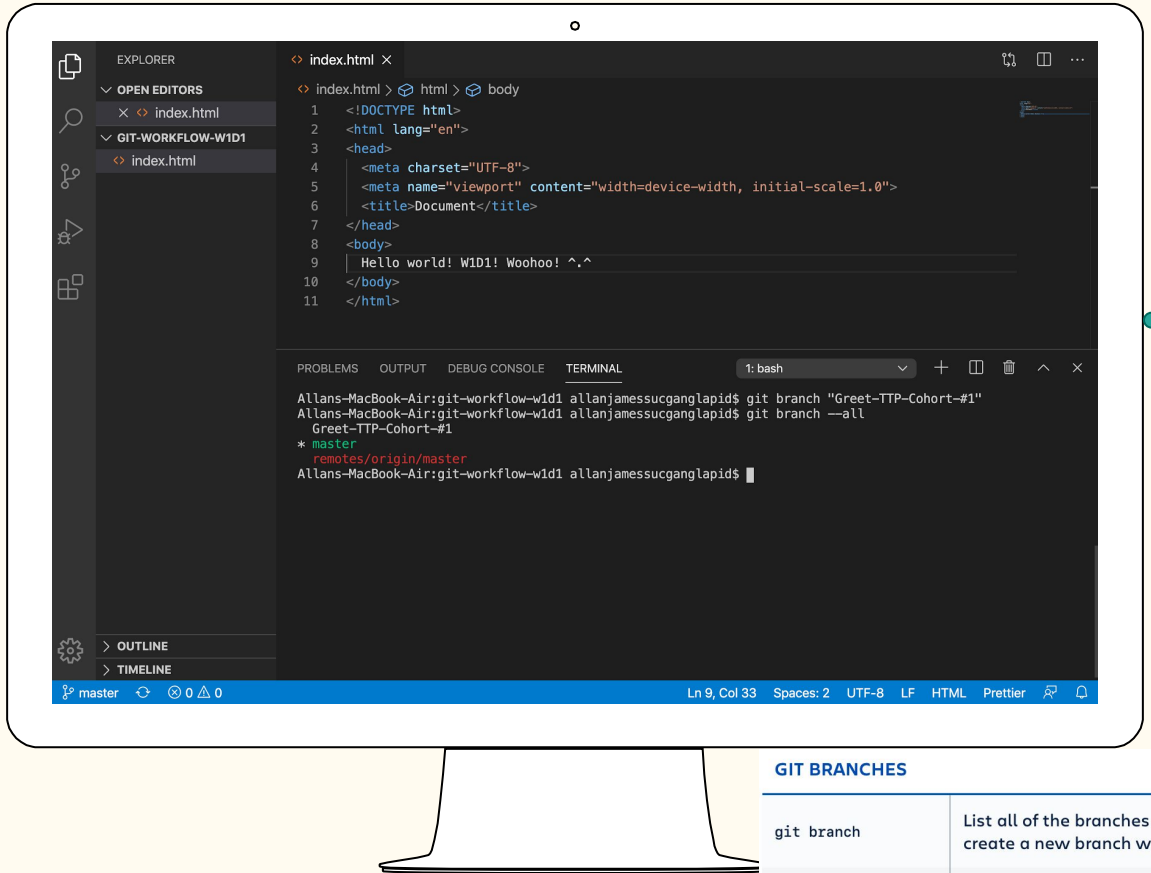
Setting Up GitHub Projects

6) The issue you created on the Projects Board will also be automatically reflected/populated in the “Issues” tab as demonstrated by the image on the right-hand side



Back To Regularly Scheduled Programming

git init	✅
create upstream repository	✅
git remote add	✅
git add	✅
git commit	✅
git push	✅
create issues	✅
git branch	✅
git checkout	TBD
create pull requests (PR)	TBD
sync and close PRs	TBD
deploy	TBD

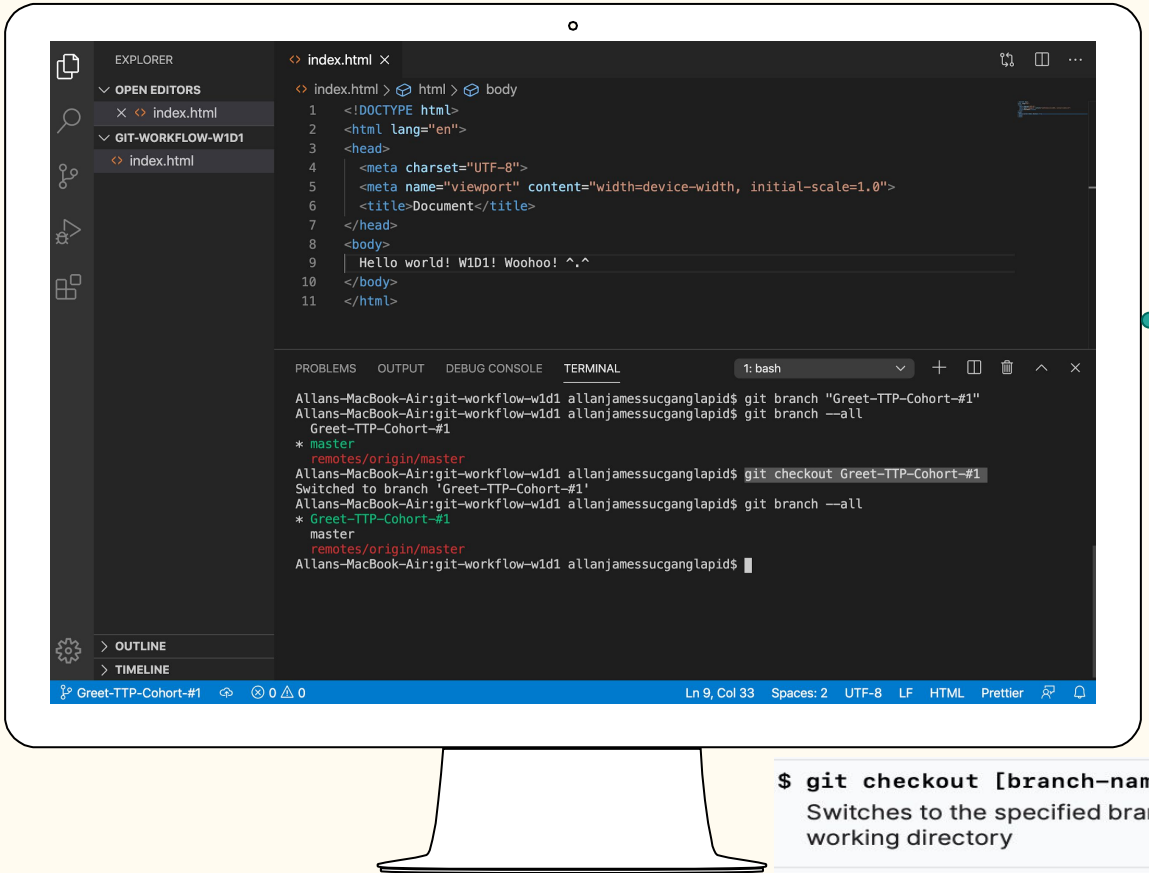


git branch --all will list both local branches and remote-tracking branches as well as the current branch you are on (via asterisk)

GIT BRANCHES

`git branch`	List all of the branches in your repo. Add a <branch> argument to create a new branch with the name <branch>.

git init	✅
create upstream repository	✅
git remote add	✅
git add	✅
git commit	✅
git push	✅
create issues	✅
git branch	✅
git checkout	✅
create pull requests (PR)	TBD
sync and close PRs	TBD
deploy	TBD



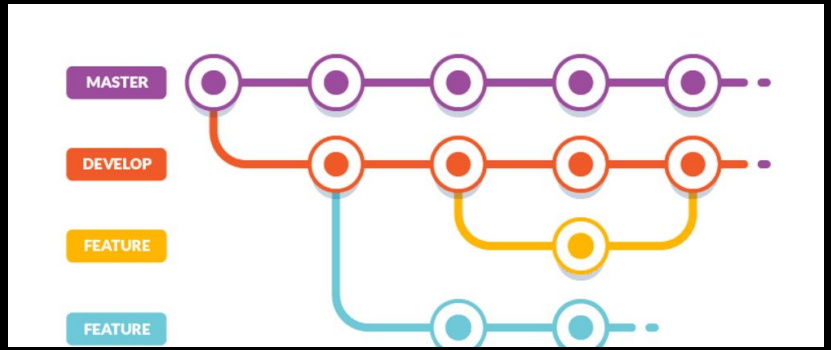
git branch --all will list both local branches and remote-tracking branches as well as the current branch you are on (via asterisk)

`$ git checkout [branch-name]`

Switches to the specified branch and updates the working directory

Wait, what's a branch and why are we switching to it?

Whenever we want to isolate and specify work on a feature branch, one that is separate from the mainline (main branch), we can rely on a branch



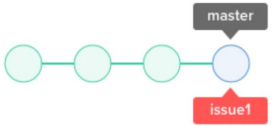
Branching can be done multiple times at multiple points (if necessary), and each branch can be an inexpensive, parallel way to experiment across different solutions independent of the mainline

Technical: A branch in Git is actually a simple file (41 bytes) that contains a 40-character string (plus a newline) referring to the commit (snapshot) it points to

```
git branch issue1
```

The illustration below provides a visual on what happens when the branch is created. The repository is the same, but a new pointer is added to the current commit.

Before new branch is created



After new branch issue1 is created

Switching between branches in Git changes the code in your workspace to reflect the exact state of the project at the time of each commit on that branch, providing you with different snapshots of the codebase as you move between branches.

This is *also* the reason why we actually do not have a **main branch** established immediately after `git init` until the “initial commit” takes place

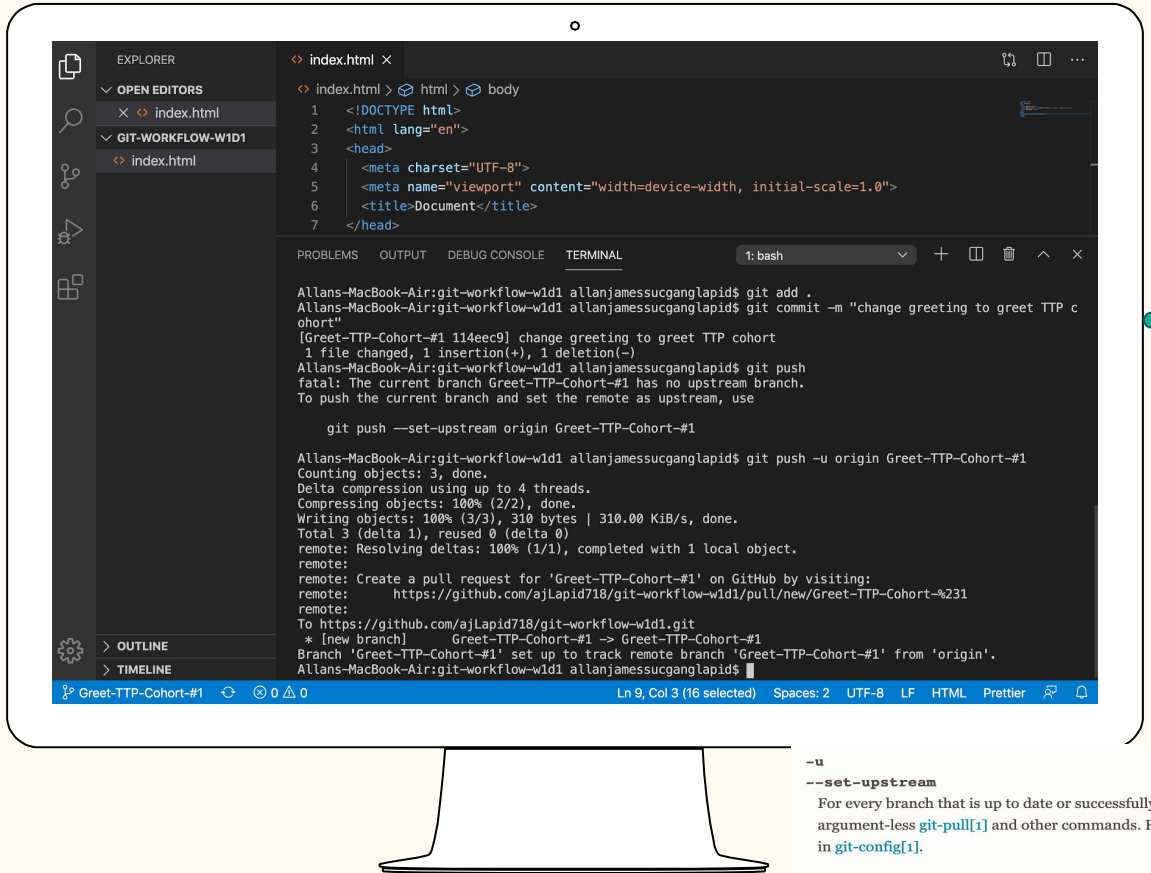
You *also* cannot make a **feature branch** until the “initial commit” (first commit in the repository) exists…because a branch is designed to hold a reference to a commit and it typically cannot exist without a preexisting one

Consider: In practical terms, branches in Git can be seen as a way to create a separate working directory, staging area, and project history that are not connected to the mainline. This allows you to work on new features or experiments independently, with the freedom to later merge them back into the mainline or discard them without affecting the main project.

In this course: A feature branch could follow the convention of: “Ticket-Name-On-Board-#1” where “Ticket-Name-On-Board” is the title of the issue on GitHub Projects and “-#1” is the auto-incremented issue number provided by GitHub (keep branches specific and manageable, similar to best practices with commits)

Now that we know why we created a branch, what it affords us, and why we switched to it, let's continue.

git init	✅
create upstream repository	✅
git remote add	✅
git add	✅
git commit	✅
git push	✅
create issues	✅
git branch	✅
git checkout	✅
create pull requests (PR)	TBD
sync and close PRs	TBD
deploy	TBD

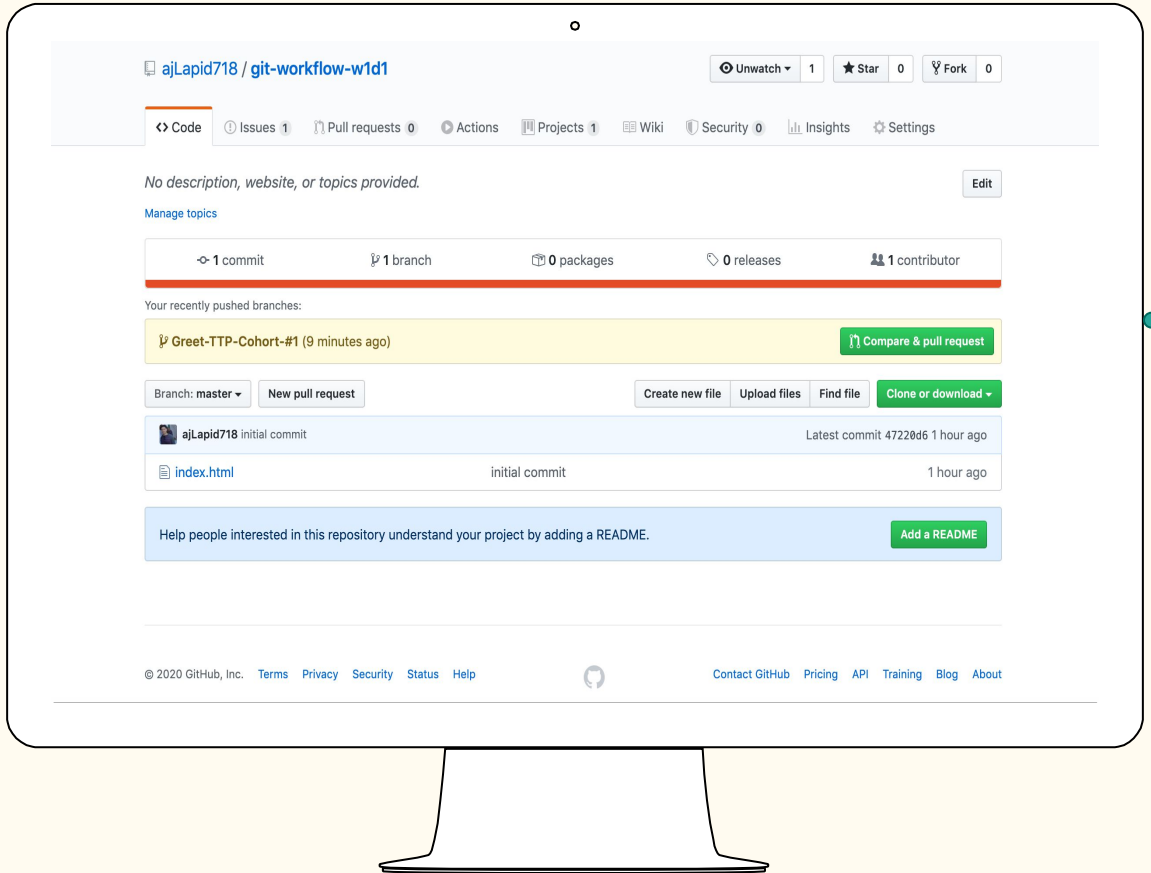


make, stage, and commit changes just like on any other branch, click on git push for help, and set remote upstream for this particular local branch

-u is shorthand (see below)

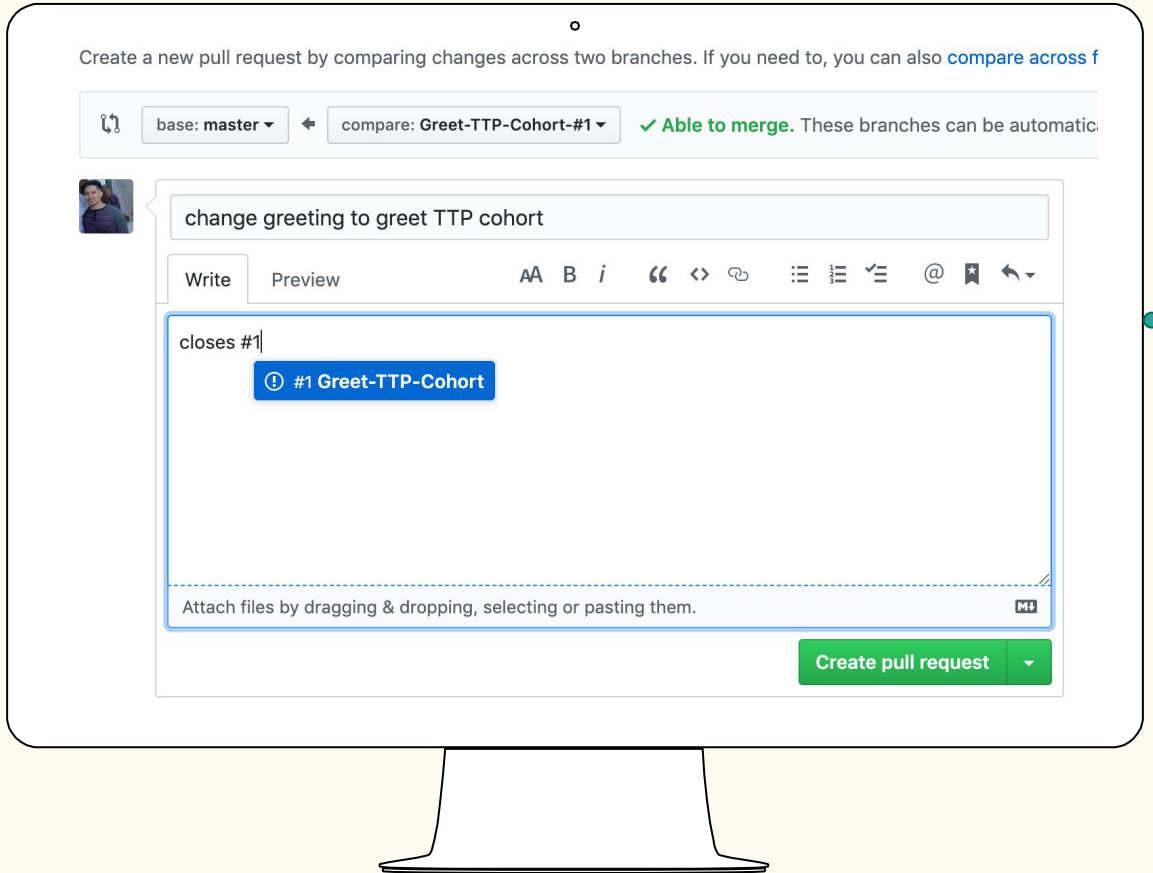
-u
--set-upstream
For every branch that is up to date or successfully pushed, add upstream (tracking) reference, used by argument-less git-pull[1] and other commands. For more information, see `branch.<name>.merge` in git-config[1].

git init	✅
create upstream repository	✅
git remote add	✅
git add	✅
git commit	✅
git push	✅
create issues	✅
git branch	✅
git checkout	✅
create pull requests (PR)	TBD
sync and close PRs	TBD
deploy	TBD



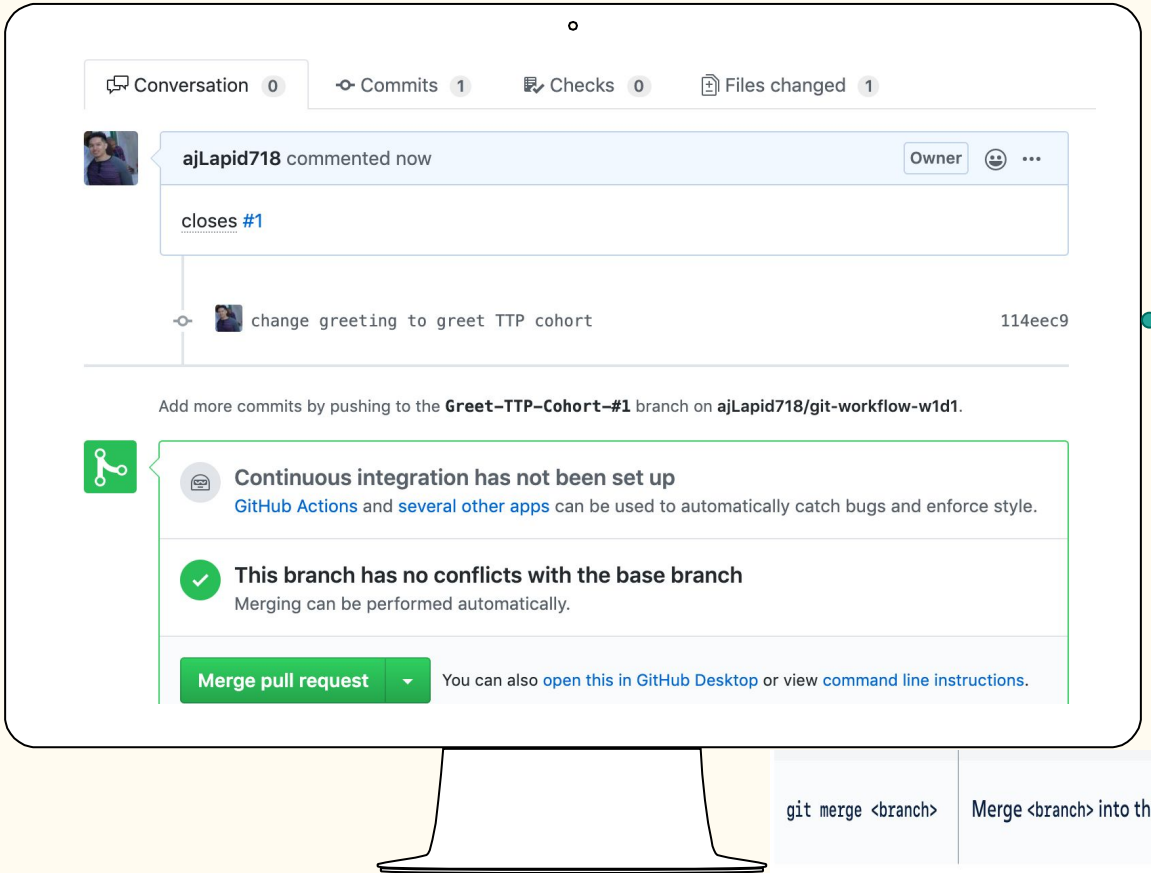
after pushing the branch, go to the main repository and click on “compare & pull request”

git init	✅
create upstream repository	✅
git remote add	✅
git add	✅
git commit	✅
git push	✅
create issues	✅
git branch	✅
git checkout	✅
create pull requests (PR)	✅
sync and close PRs	TBD
deploy	TBD



then, in the body of the PR, write “closes” followed by the “#” with the issue number, and select the appropriate option from the dropdown --- this connects the PR to the issue on the board we made earlier --- then click on “Create pull request”

git init	✅
create upstream repository	✅
git remote add	✅
git add	✅
git commit	✅
git push	✅
create issues	✅
git branch	✅
git checkout	✅
create pull requests (PR)	✅
sync and close PRs	✅
deploy	TBD



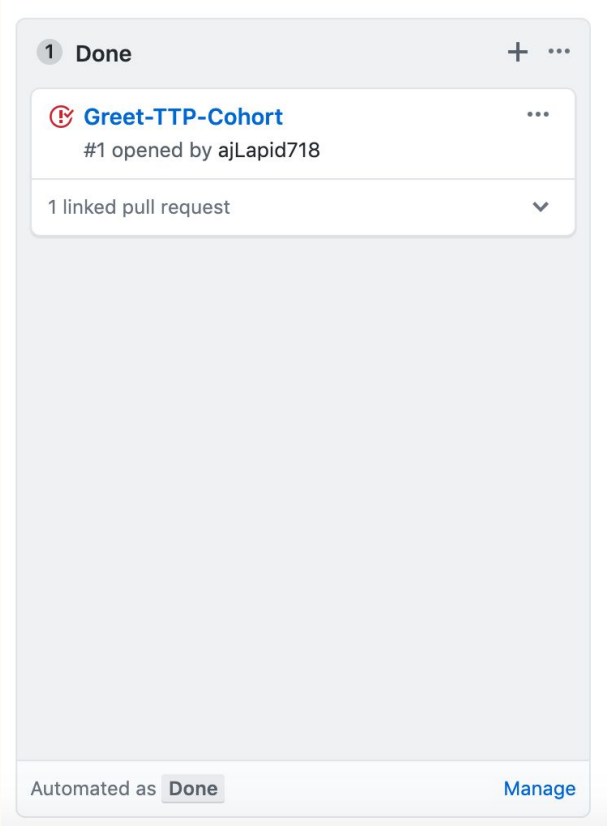
click on “Merge pull request” in order to take the changes from the remote branch and merge them with the remote main branch

`git merge <branch>`	Merge <branch> into the current branch.

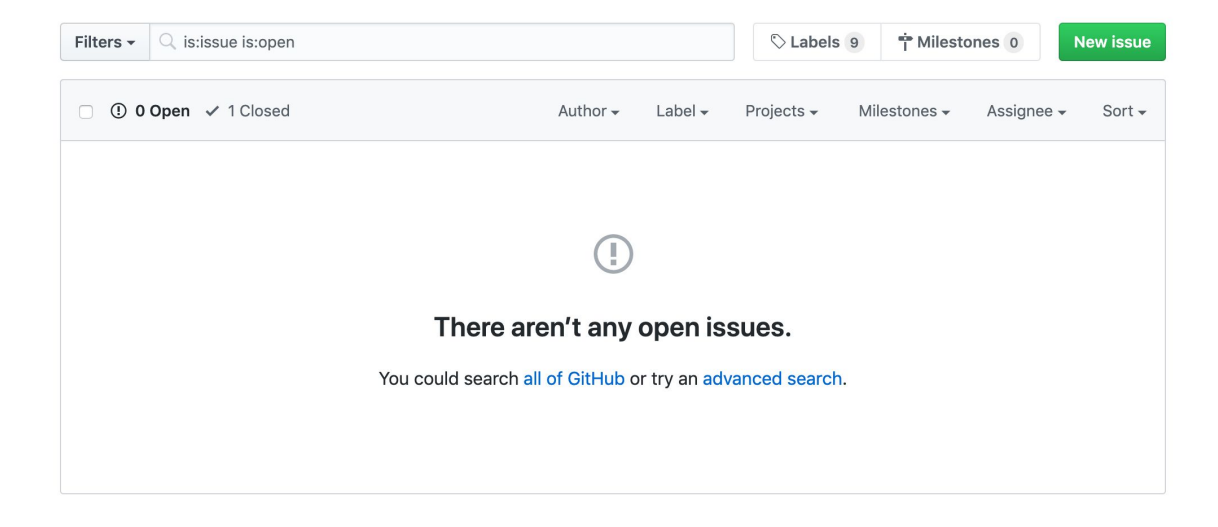
git init	✅
create upstream repository	✅
git remote add	✅
git add	✅
git commit	✅
git push	✅
create issues	✅
git branch	✅
git checkout	✅
create pull requests (PR)	✅
sync and close PRs	✅
deploy	TBD

click on “Confirm merge” --- all done

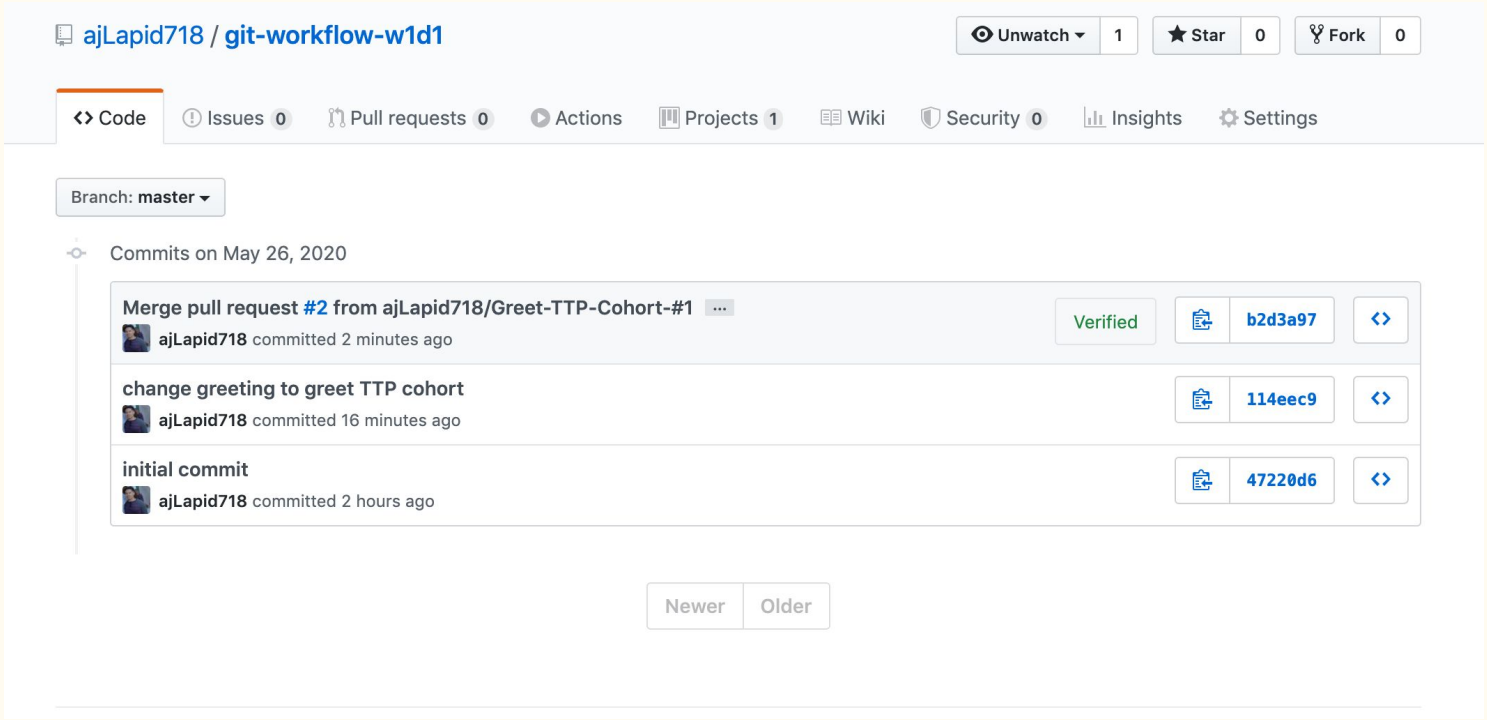
`git merge <branch>`	Merge <branch> into the current branch.



You may notice that the GitHub Projects Board automatically moved your issue associated with your Pull Request (PR) from the “To Do” column to the “Done” column

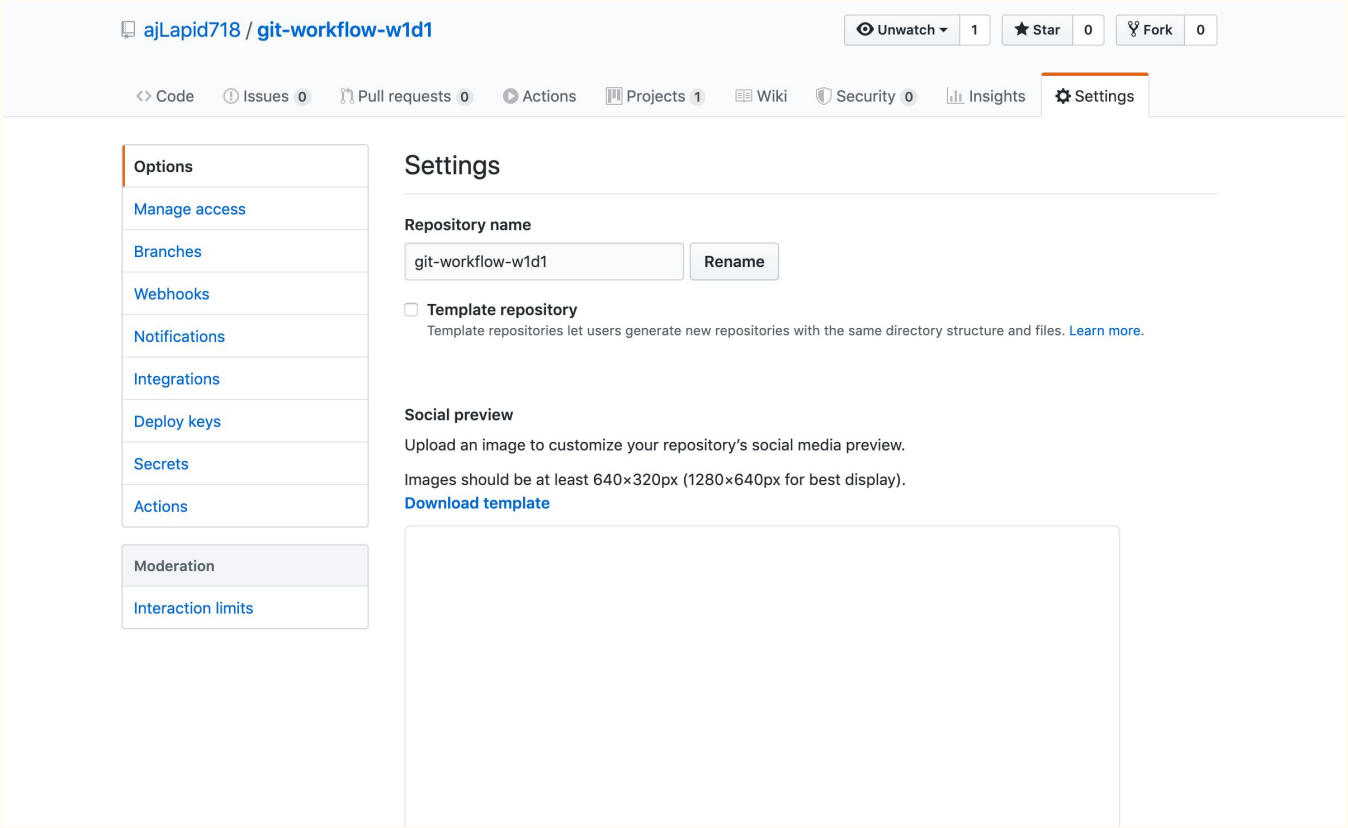


You may also notice that there are no longer any “open” issues (instead, there is now 1 “closed” issue) in the “Issues” tab, mirroring the GitHub Projects Board

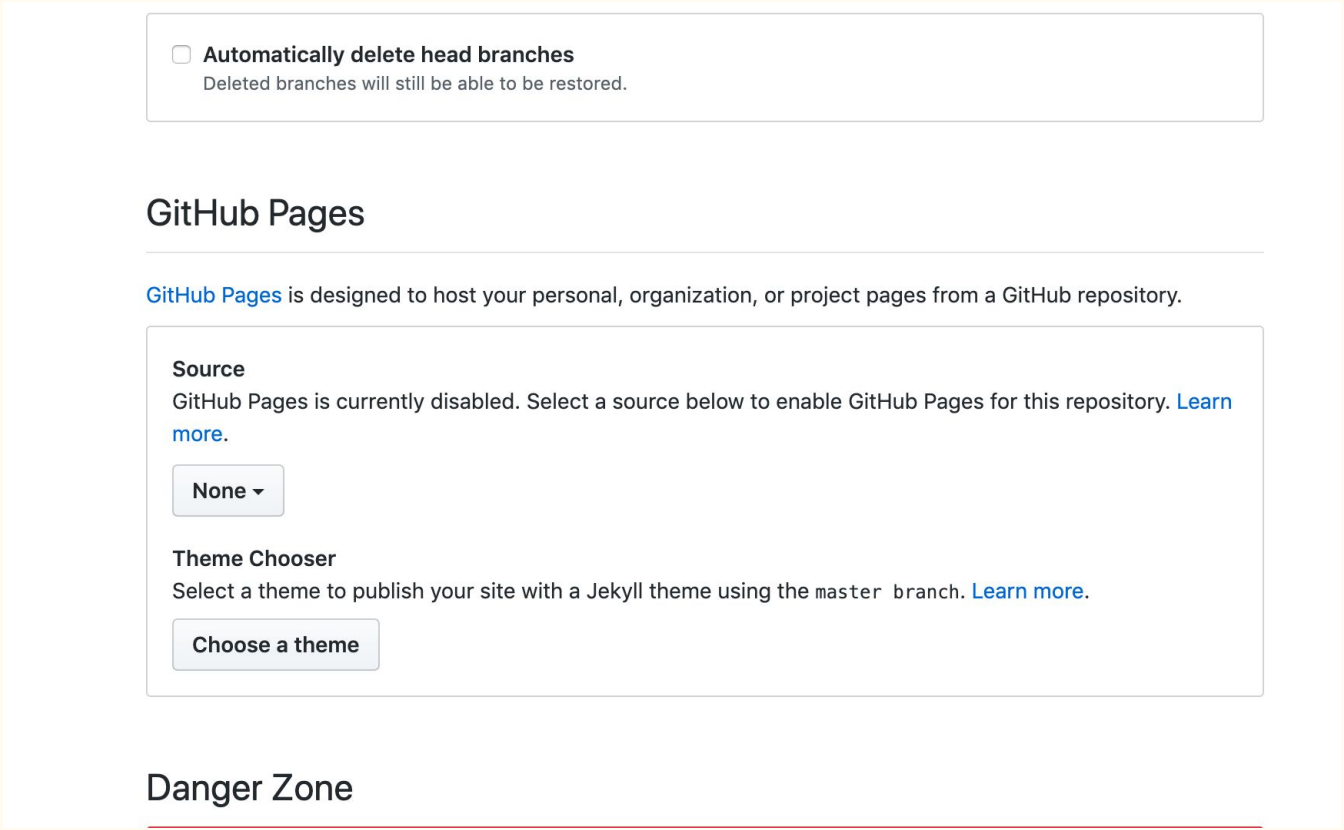


In addition, you may notice the “commit history” on the main branch has changed, confirming a successful merge. The extraneous “Merge Commit” with the “Verified” label has two parents: “initial commit” and “change greeting to greet TTP cohort”. To unify the work you just did with what was previously on there, the Merge Commit will have parents such that the last commit of each branch involved (main and Greet-TTP-Cohort-#1, in this case) are the parents of the Merge Commit node.

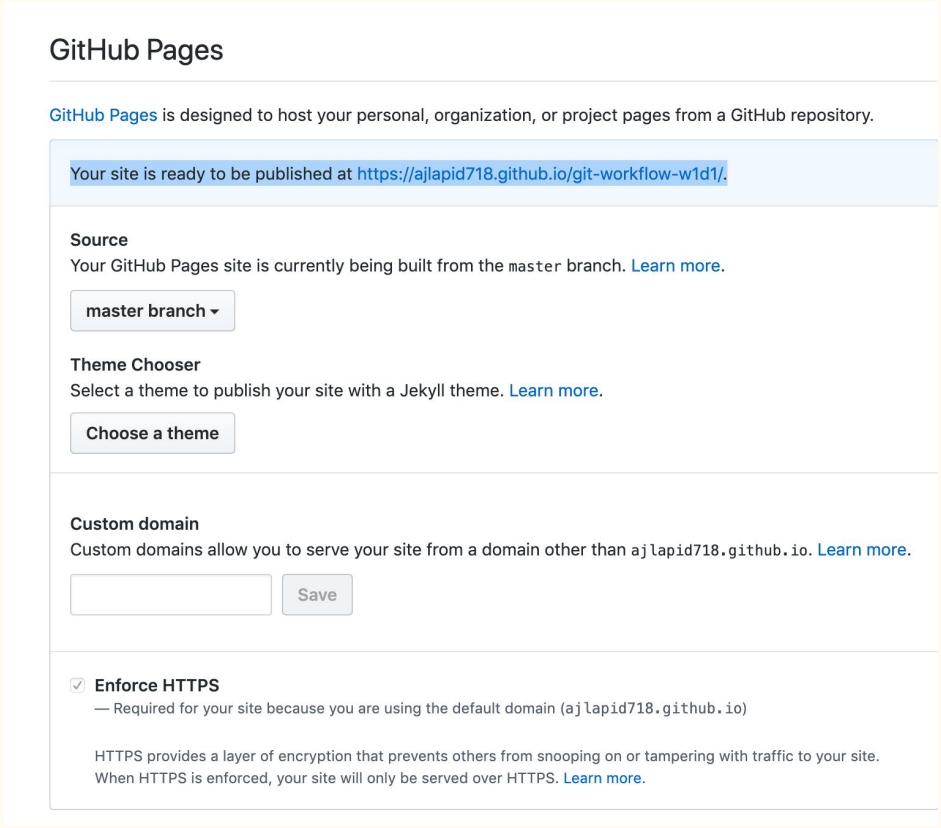
Time To Deploy To GitHub Pages



1) Navigate to the “Settings” tab

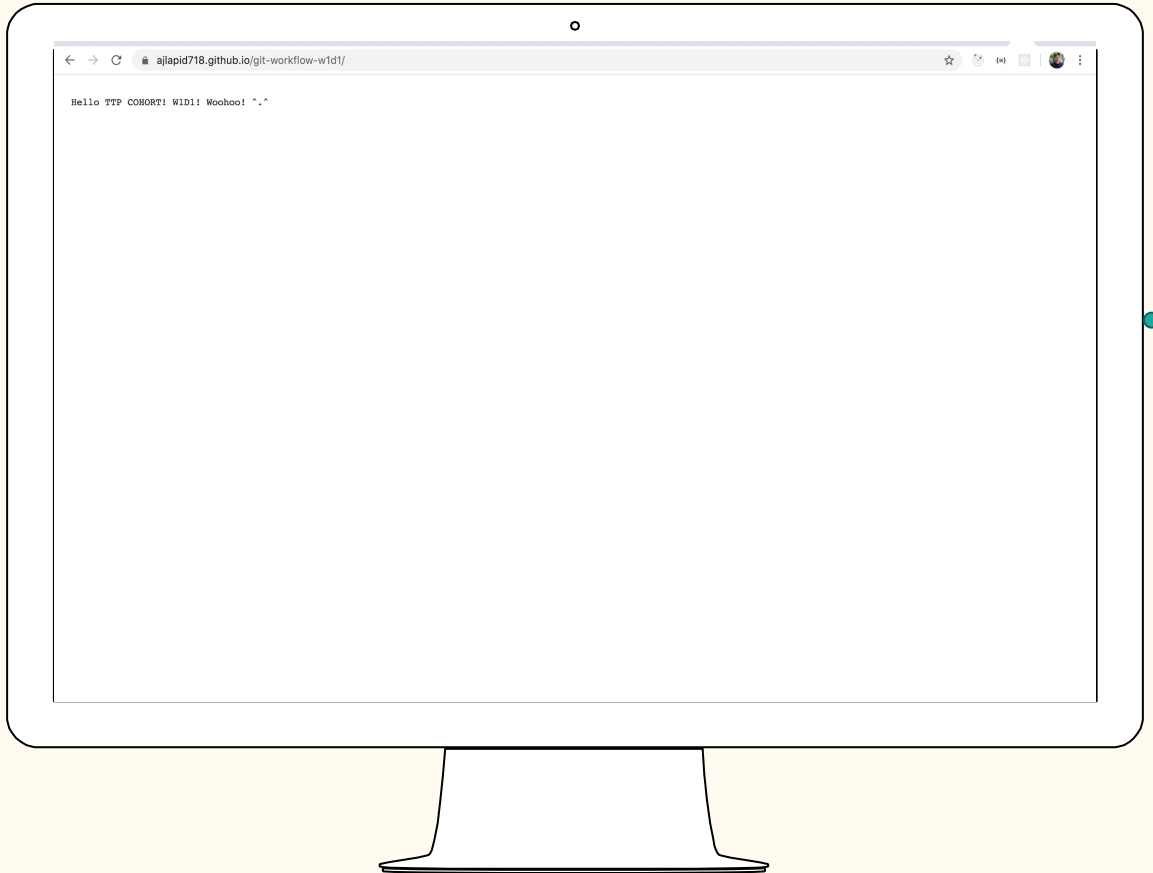


2) Scroll down to the “GitHub Pages” section and select “main branch” for the “Source” dropdown (not shown)



3) Your site is ready to be published at `https://<username>.github.io/<name-of-github-repository>`

git init	✅
create upstream repository	✅
git remote add	✅
git add	✅
git commit	✅
git push	✅
create issues	✅
git branch	✅
git checkout	✅
create pull requests (PR)	✅
sync and close PRs	✅
deploy	✅

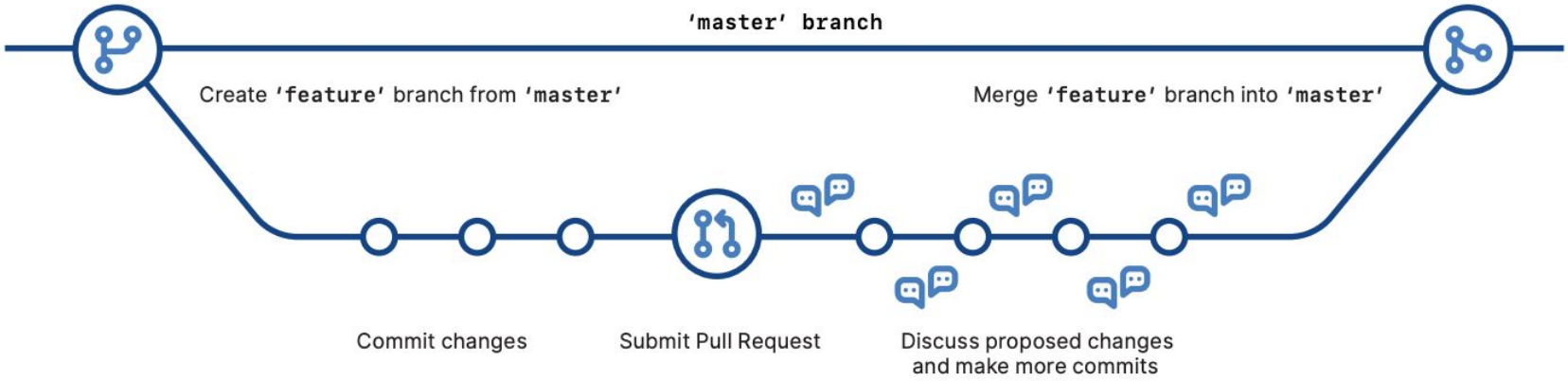


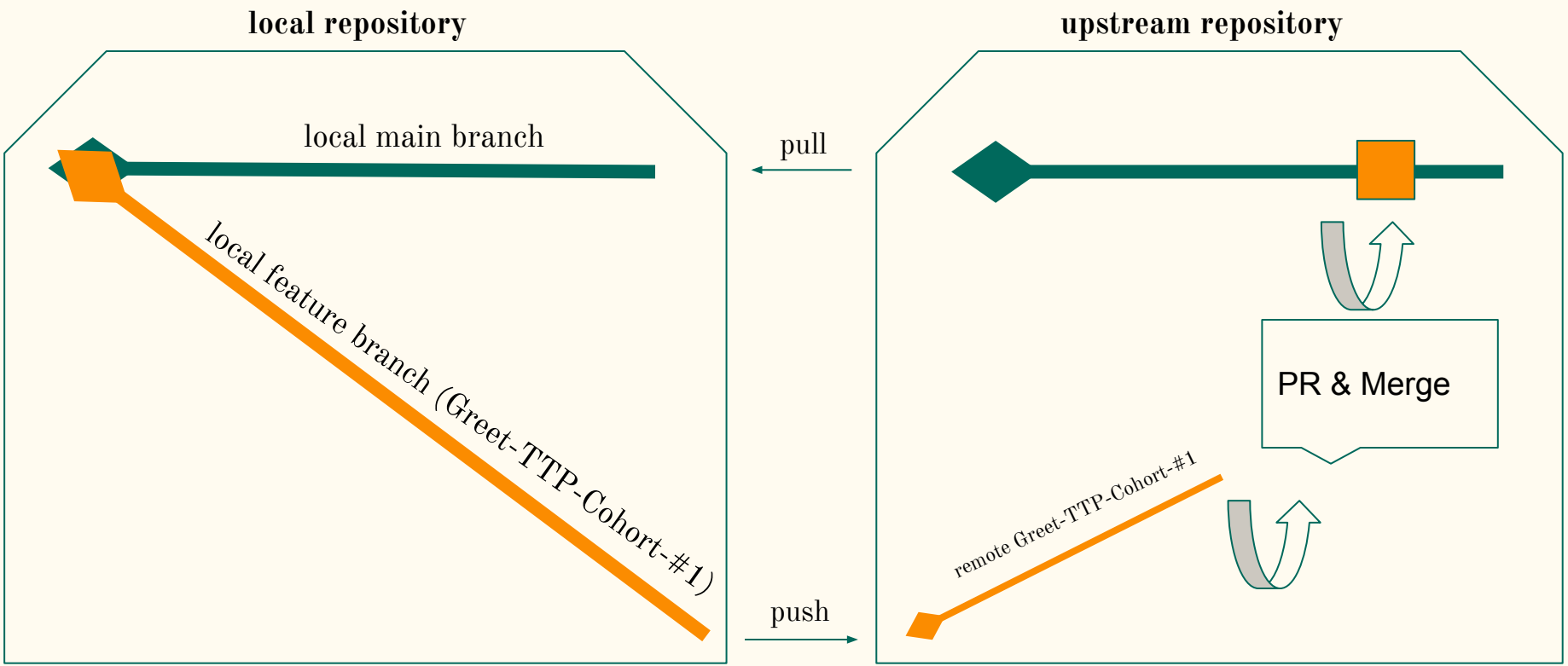
Let’s take a five minute break

2) The Golden “Git” Workflow

The “Golden” Git Team-Based Workflow with an Established Project on GitHub (Continuing With What We Just Pushed Up and Deployed)

GitHub Flow


'master' branch
Create 'feature' branch from 'master'
Merge 'feature' branch into 'master'
Commit changes
Submit Pull Request
Discuss proposed changes and make more commits



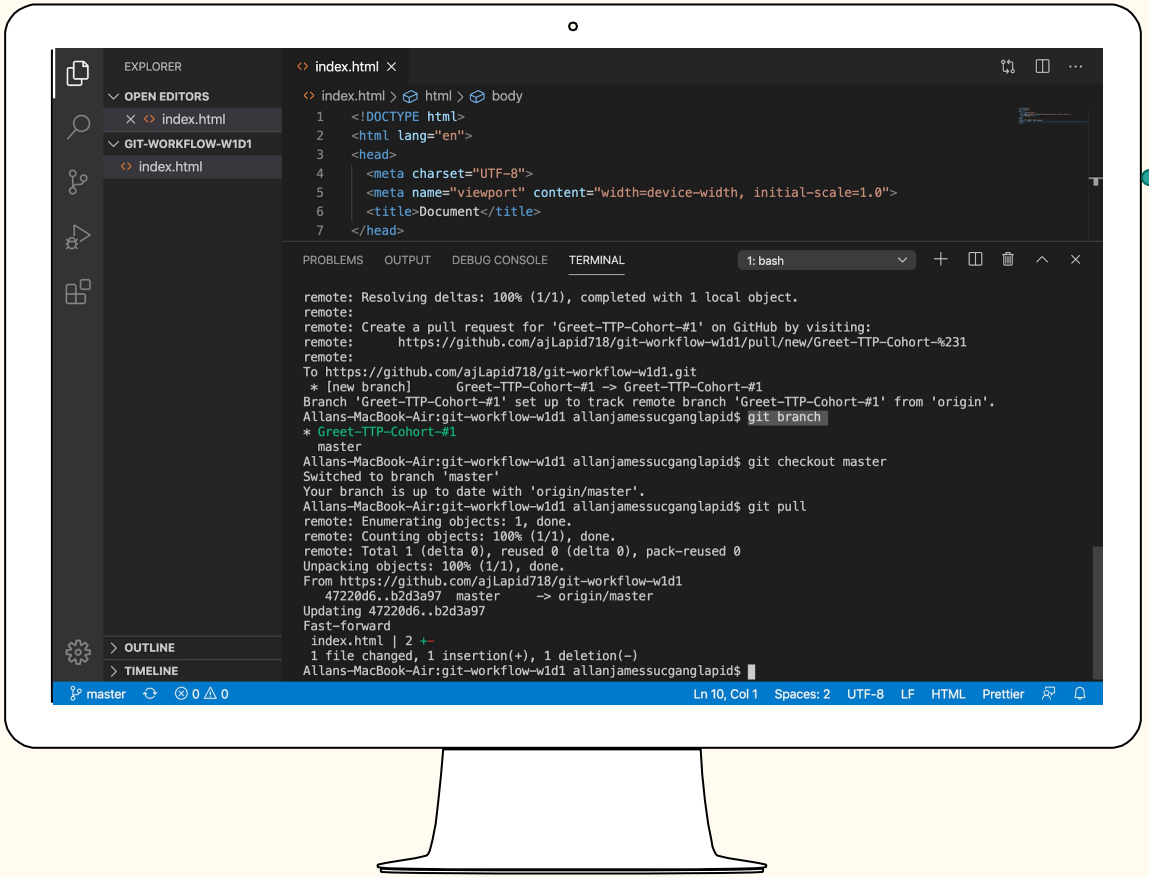
Note: This is a loose heuristic to visualize. Technically, the PR & Merge will create a Merge Commit that gets tacked onto the project's history, which is not displayed here. This extraneous Merge Commit will have two parents: the last commit on Greet-TTP-Cohort-#1 and the last commit on the main branch, both of which are determined based on the time the `merge` occurred. This visual is primarily supposed to show the "full circle" of events, in general, and why we need to checkout to our local main branch (post-push-pr-and-merge) and run a `git pull` command. Also, when we push and pull, the remote tracking references are facilitating the transportation of information in either direction.

The Golden “Git” Workflow

1. Pull from the upstream repository to update your local main branch. IMPORTANT
2. Create a dedicated branch for the feature with a descriptive name with semantic naming from the Github issue.
3. Switch to the feature branch.
4. Make changes, stage them, and commit.
5. Push the feature branch to the upstream repository.
6. Create and accept a pull request for review.
7. Merge the feature branch into the upstream main branch.
8. Switch back to the local main branch and use git pull to update it

Scenario Part I

- I want to start a new feature regarding the greeting we deployed, which centers around making the font size appear larger for better readability
  - I should confirm which branch I am currently on (via the asterisk provided by git branch) and then switch FROM the local feature branch (Greet-TTP-Cohort-#1) to the local main branch (main)
  - Then, I should execute the command `git pull` (since the upstream was configured previously, it will implicitly do `git pull origin main` for us) in order to have the latest updates FROM the remote main branch (side note: you can push to and pull from any branch, but in this case, we are using the main branch)
  - This ensures that I have the latest working version of the server-side repository and that it is, at this point in time, the best available (most updated) version of main to start to work on my assigned feature/ticket/issue



The remote repository has the latest merged changes, but we need to update our local repository's main branch since the remote main branch changed

[Verbose] Side Note (We will discuss further later on)

On the previous slide, you may have noticed (after the command: `git checkout main`) that it stated: Your branch is up to date with 'origin/main'

This might seem strange if we, as some people do at first, think of `origin/main` as the remote repository we just updated on the GitHub platform, but let's quickly recap and see why that is not the case.

We started off on the local main branch (main). We then created a local feature branch (Greet-TTP-Cohort-#1). We created, staged, committed, and pushed those changes from the local feature branch to the remote feature branch. We then merged the remote feature branch (origin Greet-TTP-Cohort-#1) into the remote main branch (origin main). At this point, the **remote main branch** is up-to-date, but then the **local main branch** needs those changes, allowing things to come back full circle. The local main branch tends to be the last one to "receive the news" that a feature made its way from development to production.

How could my local main branch be up to date with 'origin/main'? This is true because 'origin/main' is different from origin main. This, 'origin/main', is a **remote tracking reference** to a branch known as "main" that belongs to a project that goes by the value of the **origin** variable. This gets updated when information is pulled from the corresponding branch on the remote repository (aka: git fetch/git pull). Keep this in mind --- "origin/main" is a local copy of its remote counterpart and "origin main" is a remote copy on the GitHub server. This might require re-reading, but more on this later.

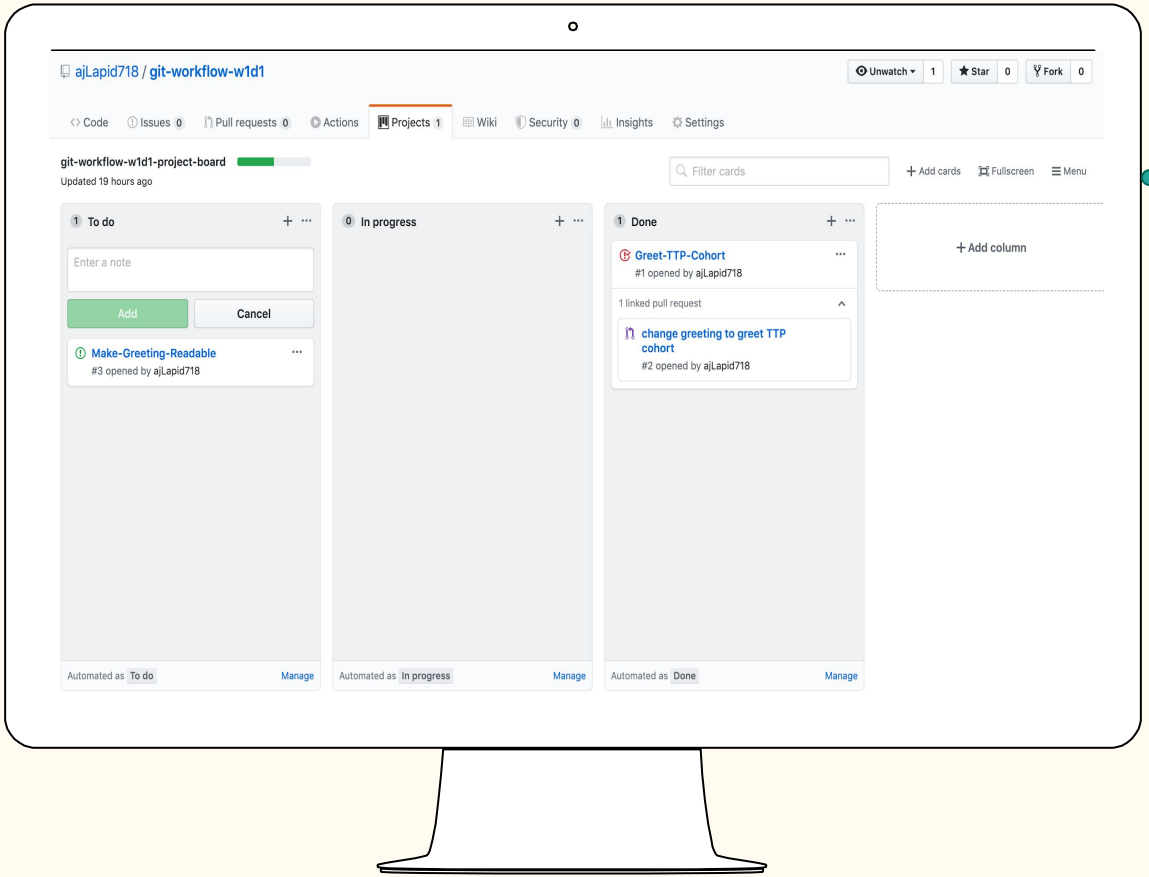
`git remote show origin` will allow you to see those local reference tracking branches mentioned

Scenario Part II

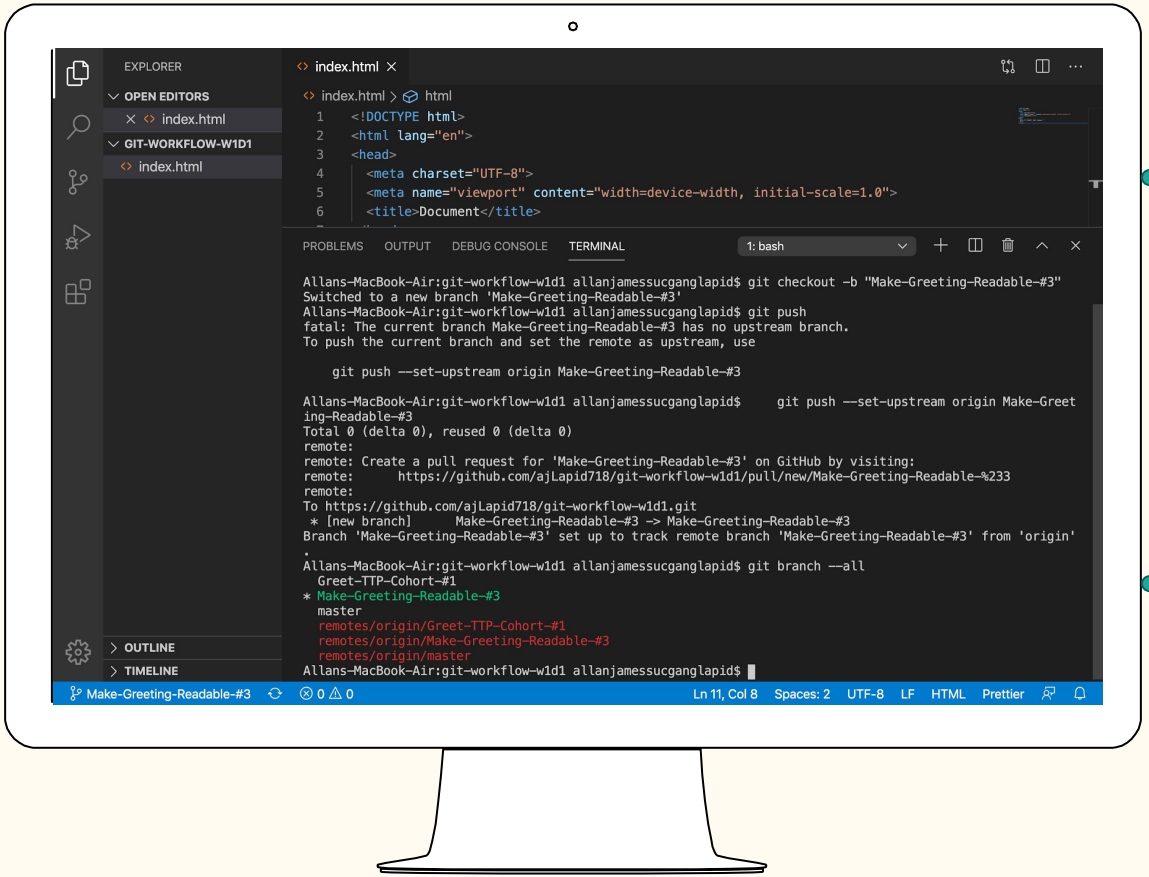
- We just completed Step 1. That is, we pulled down information from the main branch on the remote repository to the main branch on the local repository. Before every new feature, this is a highly recommended step!

- Now, here are steps 2 and 3
  - create a branch dedicated solely for that feature (with semantic naming convention based on an issue created on GitHub)
  - checkout to that feature branch

- The following slide(s) will use the command `git checkout -b <branch-name>` to create a new branch and immediately checkout to it, thereby fulfilling two steps with one unifying command



We made a note and converted it to an issue (see earlier slides for the step-by-step instructions)



`git checkout -b <branch-name>` will allow us to create a branch and immediately switch to it

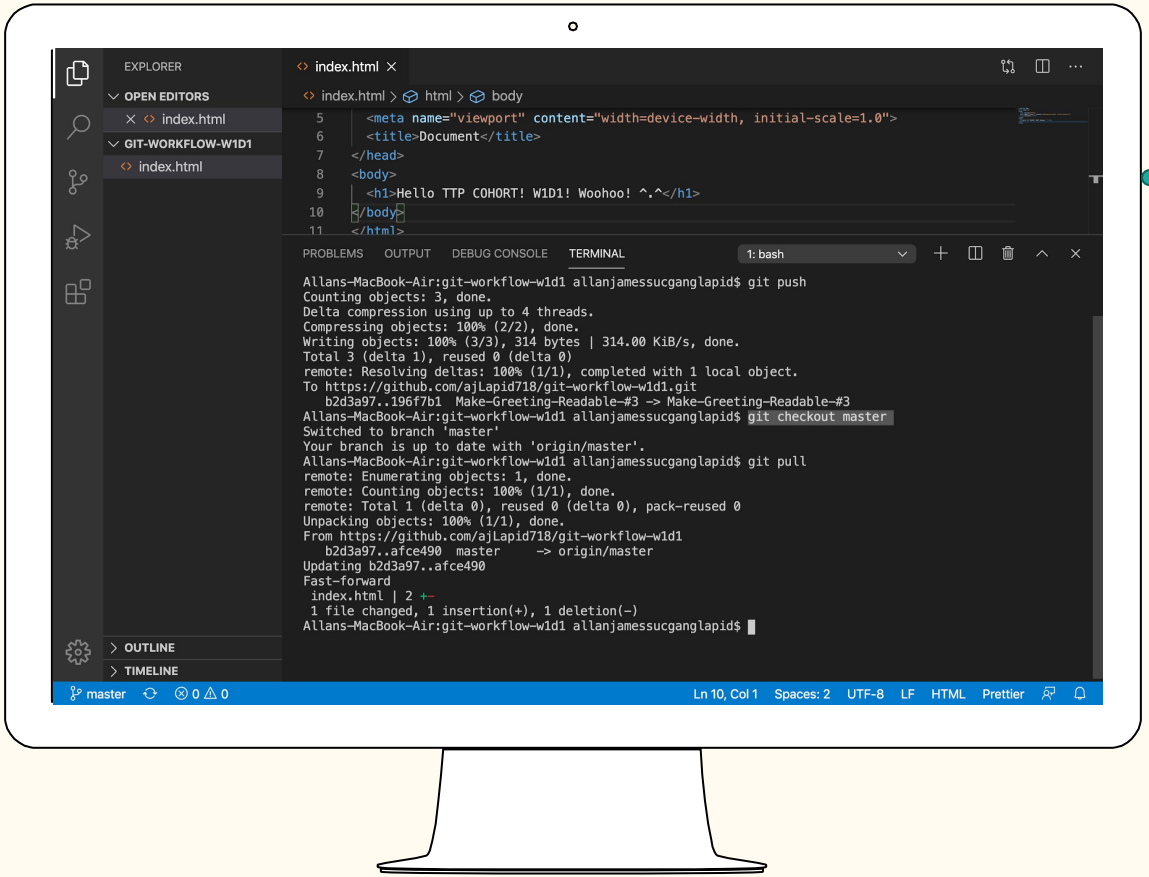
Immediately after you make a branch, you can push it up (and set the upstream, too) in case someone would like to see your latest changes or eventually work with you on it (they would pull it) --- this also sets up a local remote tracking reference

Scenario Part III

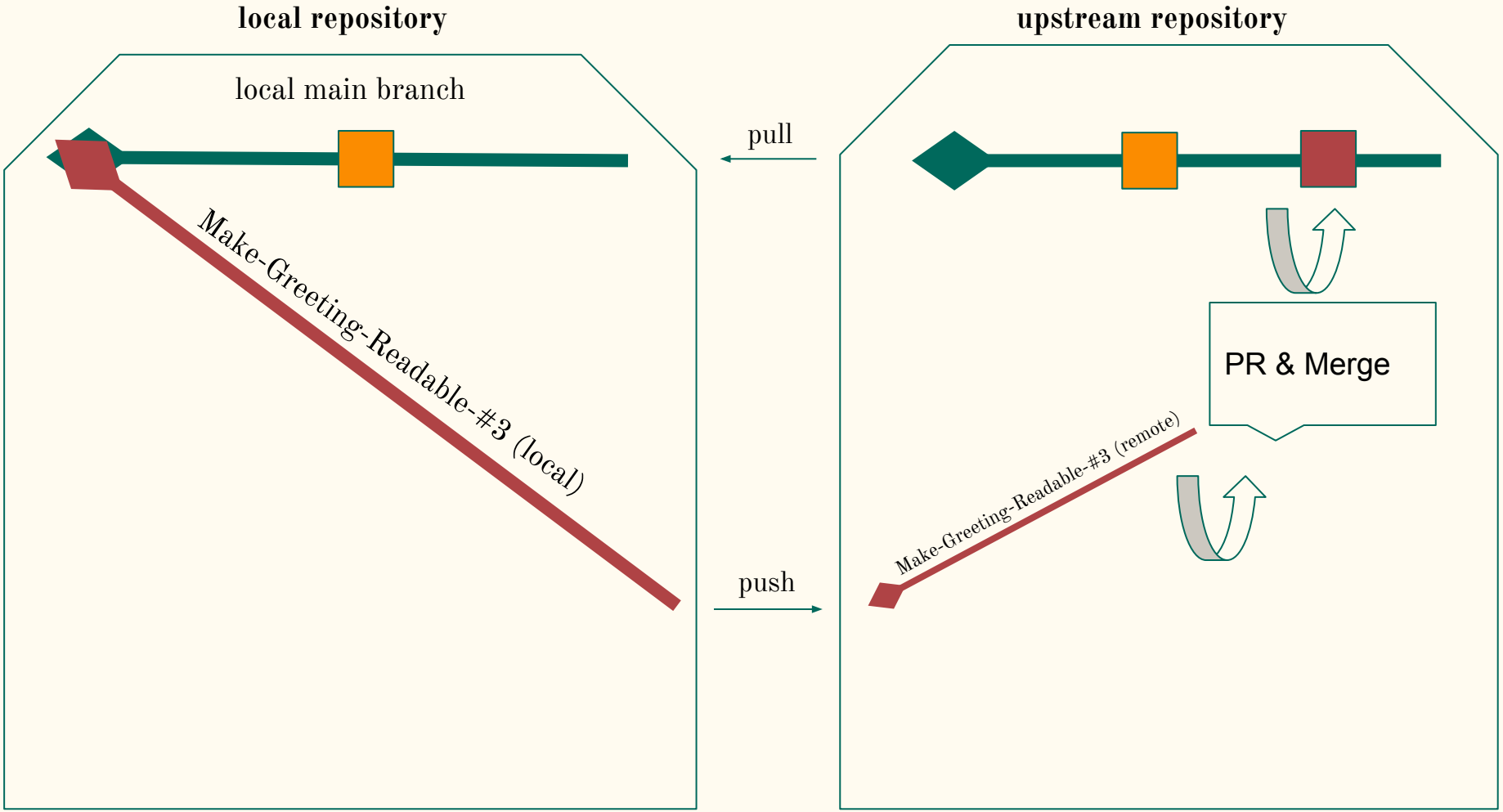
Here are steps 4, 5, 6, and 7 (these are similar to the git workflows we've worked with so far)

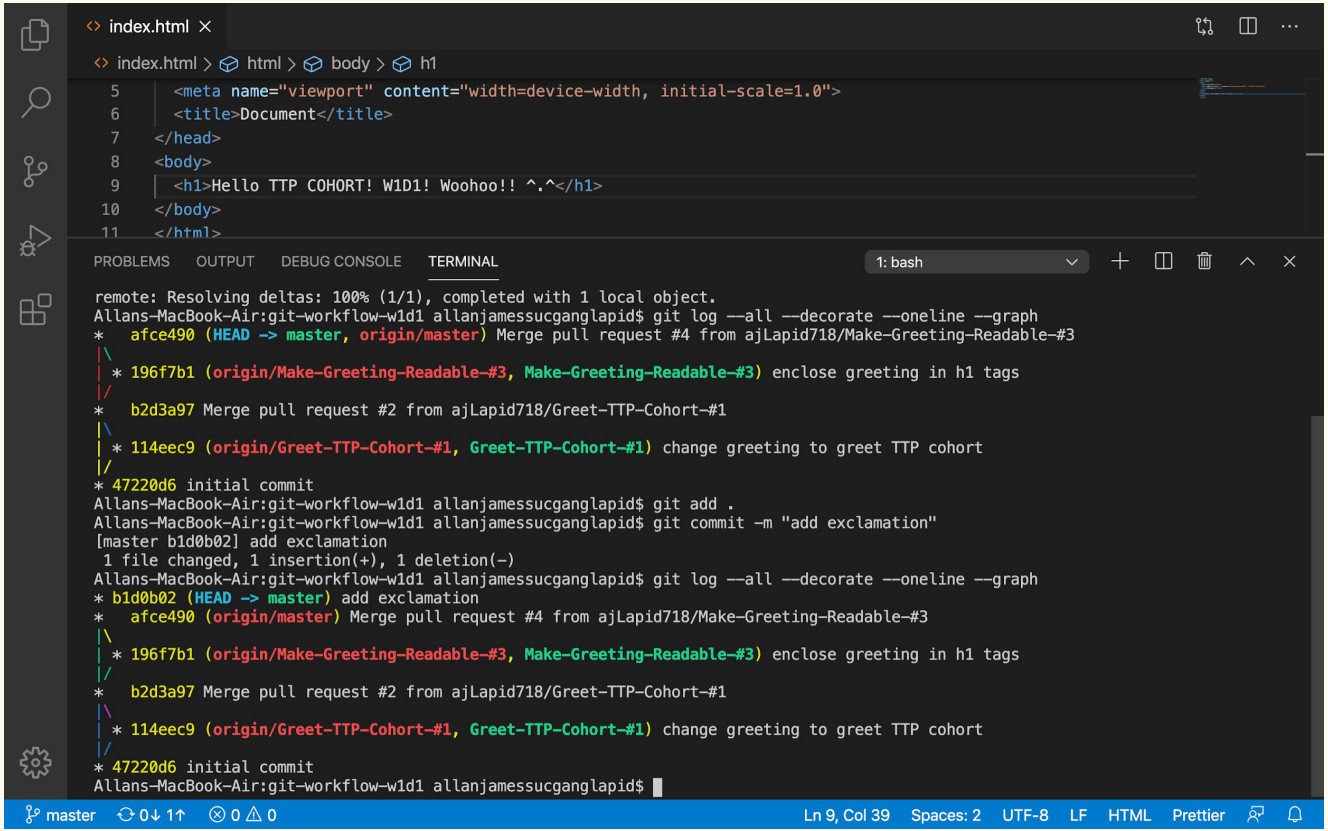
- make changes in workspace, stage those changes to the index, commit those changes
  - enclosed the greeting in <h1></h1> tags, in this specific case
- push the branch up from the local repository to the upstream repository
- create and accept a pull request
- merge changes in upstream feature branch to upstream main branch

For the sake of time, we will skip these steps (not shown), and cover Step 8, which is more of a reminder than anything else: after you merge your changes remotely, go full circle by **checking out from local feature branch to local main branch and `git pull` to receive the latest version.** Now would also be a (optional) good time to delete the branches you no longer need (locally and remotely) if necessary (please research this declarative command on your own).



`git checkout -b <branch-name>` will allow us to create a branch and immediately switch to it


local repository
local main branch
Make-Greeting-Readable-#3 (local)
pull
push
upstream repository
PR & Merge
Make-Greeting-Readable-#3 (remote)



If you want to see your local repository's commit history in the form of a readable graph, you can execute `git log --all --decorate --oneline --graph` in the terminal

3) More Git!

Brief Glossary

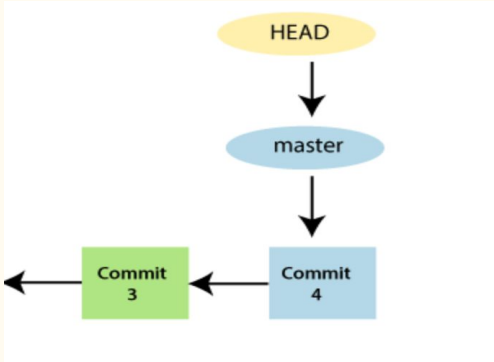
3. More Git

a. Glossary
   i. HEAD
   ii. Detached HEAD
   iii. Origin
   iv. Main vs Origin/Main vs Origin Main

b. Some Brief Gotchas
   i. Pushing and Pulling
   ii. Nested Submodules
   iii. Conflicts
   iv. .gitginore

Glossary: HEAD

- HEAD is like a pointer in Git that indicates the current state of your project.
- More specifically, it points to the branch you are currently working on. In the context of an image, imagine HEAD pointing to the "main" branch.
- The branch you're on is also a pointer, pointing to the most recent commit of that branch
- Therefore, you can think of HEAD as effectively pointing to the snapshot of your most recent commit on the current branch.



Glossary: Detached HEAD

- Picking up from the last slide, you can only be working in one place at any single point in time, and **the HEAD pointer helps us keep track of that current branch we are working with**

- However, when HEAD does not point to a branch, it means you have checked out a specific commit (`git checkout 144b0002d46e0cc47a8ad187eff77ba5ca0af93b`)
  - HEAD, in this case, will point to that particular commit in history and NOT a branch

- When HEAD points to a specific commit, this is referred to as **a detached HEAD state**

- You may never find use for this in this course, but it's important to know that when in a detached HEAD state, any commits made in this state are orphaned and do not belong to any particular branch (think of how HEAD typically points to a branch and a branch points to a commit and a commit points to its parent commits --- we don't have that here) and so you would have to make a feature branch to save commits (a prompt from Git will suggest this, too)

Glossary: Origin (this is a typical convention)

- Whether from the instructions given to you on a new GitHub repository or through your own knowledge, to add a new remote (link to an upstream repository), you use the `git remote add` command on the terminal, in the local directory your local repository is stored at, in order to add a remote repository.

- Let's say my repository link on GitHub is: https://github.com/ttp-residency-2020/Assignment-0

- The remote link is that same link, but with a git extension like so: https://github.com/ttp-residency-2020/Assignment-0.git

- In my terminal, at an appropriate directory where I wanted to point that local Git repository to that remote Git repository, I would do `git remote add origin https://github.com/ttp-residency-2020/Assignment-0.git`
    - This will allow you to push and pull from that remote repository
    - This will assign the variable of "origin" to the value of that long remote link
    - If we did not specifically assign this value to "origin", then for commands like `git push origin main`
        - We would have to write it out manually every time like so instead:
            - `git push https://github.com/ttp-residency-2020/Assignment-0.git main`

Glossary: Main vs Origin/Main vs Origin Main

At first glance, here we have three similarly-named entities: main, origin/main, and origin main. What are ways to describe and distinguish the three?

- The conventional “starter” branch for new repositories goes by the name of the main branch. It is a branch like any other, but it conventionally and by default goes by the name of main. There are services out there that specifically look for a branch called main, and there are numerous guides out there that refer to the mainline as the main branch --- so it’s a way to make things uniform.
    - This refers to something more so involved with the workspace and index

- So, origin/main is the local copy of a remote tracking reference (this allows you to effectively fetch, merge, and pull) --- it’s format is self-descriptive in that it points to a branch called “main” on the remote repository associated with the origin variable
    - This refers to something in the local repository

- Lastly, origin main, as described by the previous slide, refers to the actual remote main branch on the remote URL associated with the origin variable
    - This refers to something in the upstream repository

Gotchas: Miscellaneous Pushing and Pulling

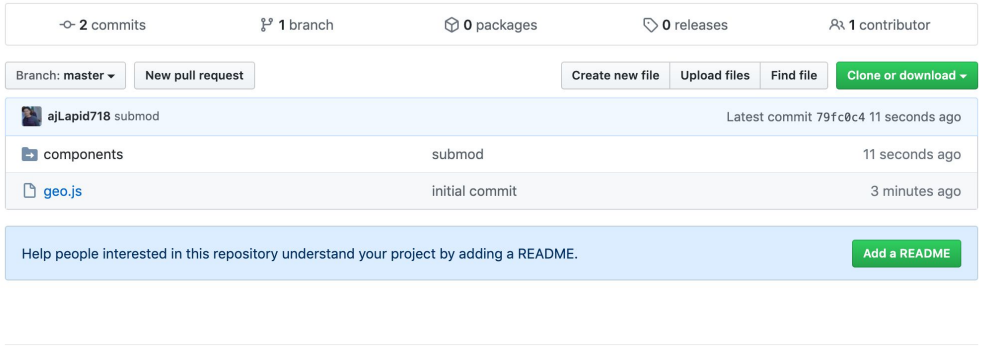
- Did you know that you can push branches up to GitHub, pull down branches from GitHub, and effectively “pass around” a branch from your local machine to your teammate’s local machine through this?
  - Keep in mind that your teammate might have the local copy of the branch as well as their own local remote-tracking reference of that branch
  - In addition, keep in mind that the upstream repository might have the remote copy of that branch
  - Keep in mind that `git branch --all` or `git branch -av` will only show the local and remote-tracking references within your local machine
  - This means you will need to `fetch` or `pull` the first time in order to “catch” the branch being passed from your teammate

- Did you know that if you start off a project and populate it with only an empty folder, then GitHub will render (aka display) that folder as greyed-out/empty on the GitHub platform?

Gotchas: Nested Submodules

Unless you are following a particular approach, your projects (at least in this course) should rely entirely on one ".git" directory existing. In other words, from the machine's perspective, `git init` (or, `git clone`, which implicitly runs `git init` for you) should only be executed once. In this way, you would avoid having somewhat difficult to deal (and unreachable on GitHub and other clones) with nested submodules like in the image below.

Consider that one `git init` creates enough to set up an entire project, and so running this command (unless purposefully creating submodules --- please research this independently if necessary and/or drop a specific question in the Slack channel) will create an entirely separate project within your main project. If you change into the submodule directory, it won't know anything about the outer/parent directory since it is the start of its own project.



https://longair.net/blog/2010/06/02/git-submodules-explained

https://gist.github.com/gitaarik/8735255/

Gotchas: Conflicts

- You can run into conflicts from the following:
  - `git stash apply`
  - git merge (locally) (to learn more about this, please run git merge --help in your terminal)
  - git merge (remotely) (same idea as local, but with the help of the GitHub platform)
- While they can be avoidable with planning and isolating features into piecemeal sizes, conflicts are natural and a part of the development process
- A conflict in a team (this can happen in solo work, too) commonly occurs when two people work on the same file and edit the same line
- This is a good thing --- Git is notifying us that it needs a manual, human decision to accept change A, change B, neither, or both --- this is Git's default behavior as the alternative is much scarier because then every time a conflicting modification was being processed, Git would make baseless assumptions on which to prioritize
  - Visual Studio Code helps streamline this process so you can handle changes locally
  - A common convention for commit messages of conflicts is: "Resolve merge conflict"
  - Run `git merge <branch> -m "Resolve merge conflict"` so the commit message happens in the terminal environment, follow git suggestions, and/or look up insertion and writing/saving/quitting in VIM/Nano (if they appear)

Gotchas: The .gitignore file

Please use the .gitignore file, especially at the very beginning of a project, to list out what will not be tracked by git and therefore not involved with not just the upstream repository but the local repository as well.

What happens if you track a file, and then you want to .gitignore it?

https://stackoverflow.com/questions/1274057/how-to-make-git-forget-about-a-file-that-was-tracked-but-is-now-in-gitignore

Miscellaneous Git I

- Be just a little bit more careful when deleting files
  - This isn't to say that there is necessarily a different technique to handle the deletion of files
  - It's still a modification
  - https://stackabuse.com/git-remove-a-file/
- Be just a little bit more careful when renaming files
  - Git keeps track of changes to files in the working directory of a repository by their name.
  - When you move or rename a file, Git doesn't see that a file was moved; it sees that there's a file with a new filename, and the file with the old filename was deleted (even if the contents remain the same)
- We understand that `git rebase` exists, as well as other commands (`squash`) and workflows --- in these four weeks, let's concentrate on purely `git merge` as it has a more favorable learning curve and is less "mutative" but we do encourage you to drop a specific question in the Slack channel and/or research independently
- Probably the **most important link** in this slide deck, a very helpful "cheatsheet" (please bookmark or save this)
  - https://www.git-tower.com/learn/cheat-sheets/vcs-workflow

Miscellaneous Git II

- `git checkout -b <Branch-Name-Here>` is equivalent to executing `git branch <Branch-Name-Here>` followed by `git checkout <Branch-Name-Here>`
  - apply whichever approach (one-step vs two-step) you feel more comfortable with

- `git pull` is equivalent to executing `git fetch` and then `git merge`
  - while there are some advantages to retrieving the information from the upstream repository and housing it, but not merging it, in you local repository via the remote tracking reference, `git pull` will cover most of the scenarios throughout this course aka “I want to use my workspace to work on the content that’s on GitHub on the moment”

- `git clone` will:
  - internally, git clone first calls `git init` to create a new repository
  - it then copies the data from the existing repository
  - it then checks out a new set of working files

Summary-ish

Summary I

- You learned a lot about Git, but the most learning will come from constantly using it, and especially through trial and error

- Learning Git, as well as other topics in this course, will be developed and retained through spaced repetition (also, we encourage research and knowing these topics will help you develop but will also help you find resources and read others code to achieve a better, wide-ranging, holistic understanding)

- Git has “declarative” commands so to some extent, while you should have an idea of what’s going on under-the-hood, you can still produce git-based projects that employ git workflows if you feel you should learn the commands first and then the “theory and architecture” second (it is a viable approach)
    - This naturally comes with the idea of technical debt
    - With constraints, certain tradeoffs (breadth vs depth) have to be made sometimes

Summary II

- Just like a lot of technologies, Git is a tool that will help you and your team accomplish your goals

- Because Git commands tend to be highly situational and contingent upon previous Git commands, it is important to know the fundamentals first and then you can look things up on case-by-case basis afterwards

- If, at this point, you have some mental model of the data flow, an idea of how (moreso where) information is transported, and even just a little bit more intuition regarding what commands to run, then you are in good shape --- a hands-on, learn-by-doing experience might assist with this

Summary III

- With Git, it's important to center yourself and ask (mention) you and your teammates some guiding questions/statements as you develop:
  - "Hey. could you walk me through the steps that you took?"
  - "Everyone should switch to the main branch and pull down the latest changes!"
  - "I realized that I had to edit a file you might be working on, so if you encounter a merge conflict, please let me know and we can work on it together"
  - "Is the main-branch password-protected on GitHub?"
  - "I'm getting permission errors, am I a collaborator for this project?"
  - "What was the commit convention we agreed to again?"
  - "What was the branch-naming convention we agreed to again?"
  - "I understand Git in my own way, but would you mind explaining it to me from your perspective?"
  - "Oh, this is similar to Google Docs and Microsoft Word saving/versioning"
- There is still a lot more to the world of Git, but this will serve as a base

Conclusion