GENERAL ASSEMBLY

INTRO TO GIT

Joseph Nelson h/t Haley Boyan

Data Science Immersive, General Assembly DC

LEARNING OBJECTIVES

- What is Git?
- Use/explain git commands like init, add, commit, push, pull, and clone
- Distinguish between local and remote repositories
- Create, copy, and delete repositories locally, or on Github
- Clone remote repositories

AGENDA

- Intro to GIT
- Demo and Guided Practice: Individual Git Usage
- Collaboration with Git

PRE-WORK

PRE-WORK REVIEW

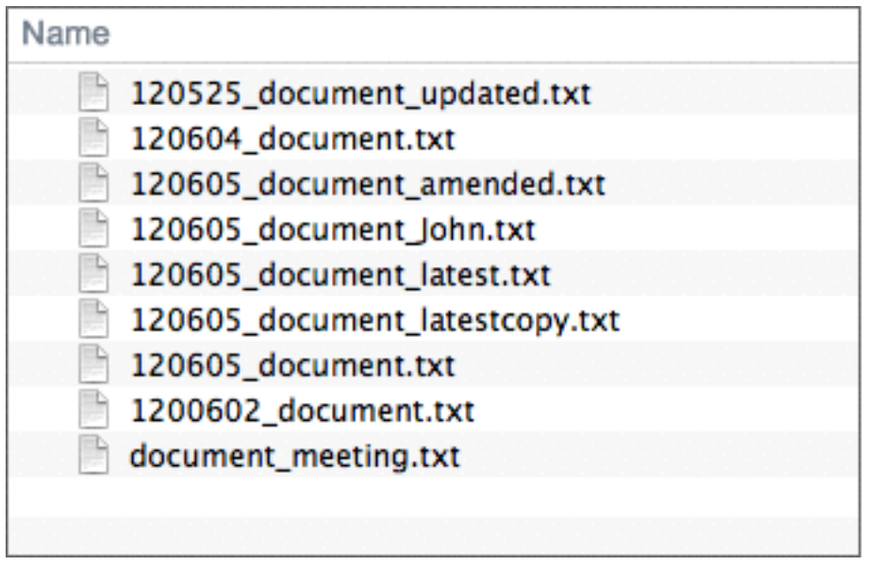
Before this lesson, you should already have done the following:

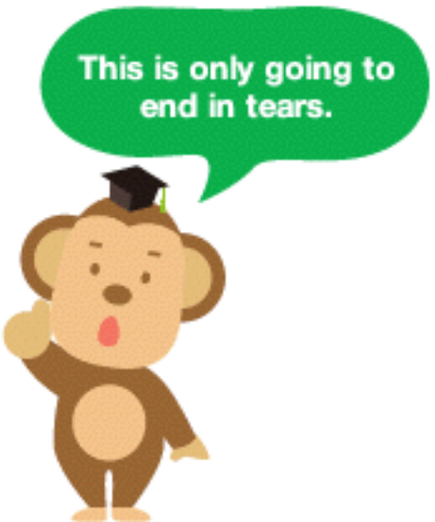
- Completed Code Academy: Learn Git
- Install Homebrew — http://brew.sh/
- Install git — Terminal: type "brew install git”
- Setup a GitHub account

LET'S GIT IT ON

WHY USE GIT?

- Easy file management across teams
- Allows collaboration
- Version tracker, allows for rollbacks
- Shows what has changed at each step
- Manages conflicts
- Dropbox for Code



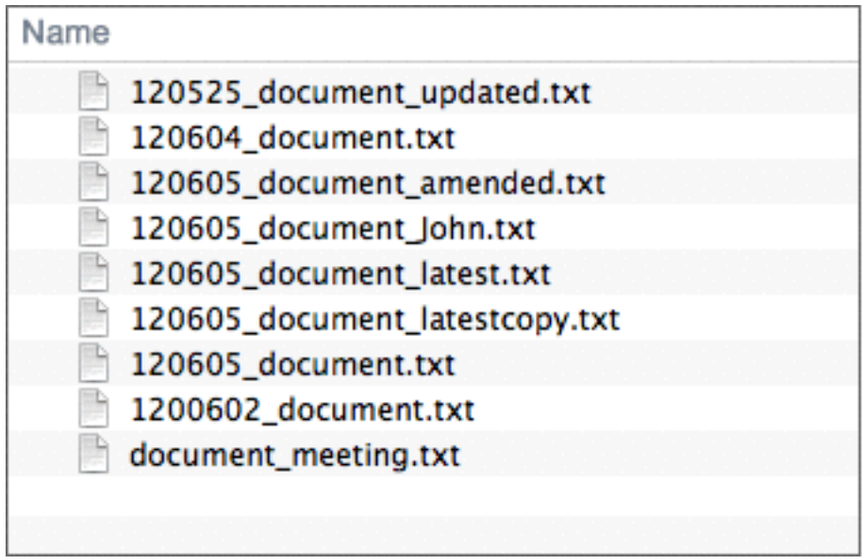


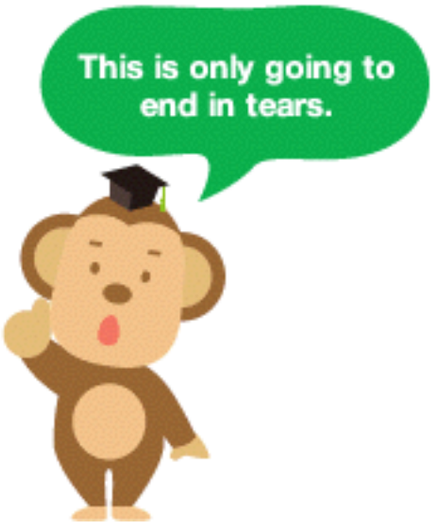
WHAT IS GIT?

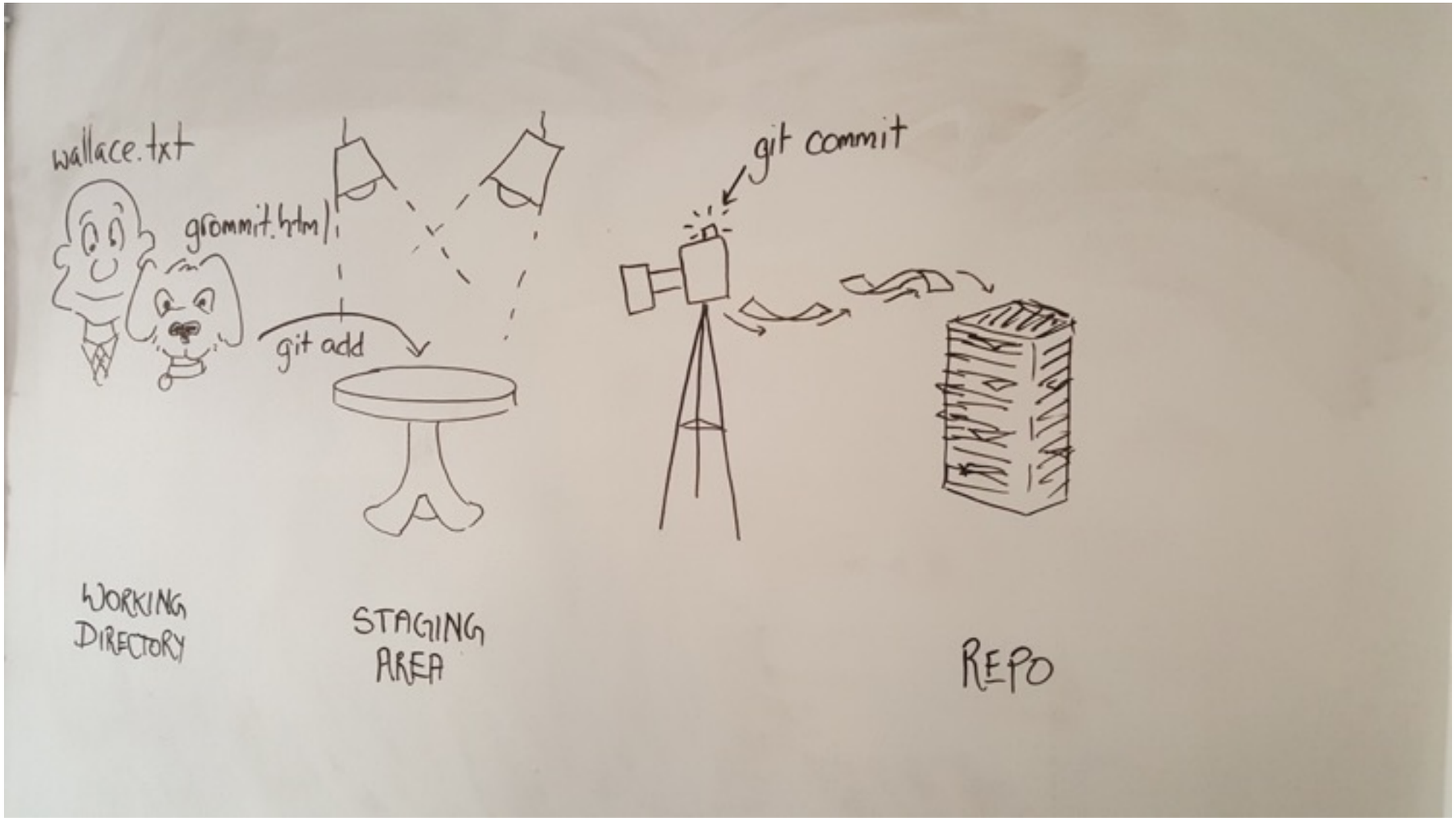
- Distributed version control system
  - Keeps history of all changes to code
  - Allows programmers to rollback changes (switch to older versions) as far back in time as they started using Git on their project
- Run from the command line or a GUI
- Created by Linus Torvalds, the principal developer of Linux.
- A codebase in Git is referred to as a **repository** (**repo** for short)

VERSION CONTROL SYSTEM

- A system that records changes to a file or set of files over time
- Allows you to recall specific versions later
- You can do this with nearly any type of file on a computer






wallace.txt
grommit.html
git add
git commit
WORKING DIRECTORY
STAGING AREA
REPO

WHAT IS GITHUB?

- DIFFERENT FROM GIT
- Hosting service for Git repositories
  - Consists of individual accounts filled with codebases
- Web interface to explore Git repositories
- Social network of programmers
  - You can follow users and star your favorite projects
  - Developers can access codebases on other public accounts
- GitHub **uses** Git

WHAT IS A REPOSITORY?

- A GIT Codebase, holding all versions of a file and the tracked changes
- Like a directory with a history
- Can either create a brand new repo from scratch (git init) or clone an existing remote repo (git clone) onto local machine.

GIT ARCHITECTURE

Repository

A set of files, directories, historical records, commits, and heads. Imagine it as a source code data structure, with the attribute that each source code “element” gives you access to its revision history, among other things.

.git Directory

The .git directory contains all the configurations, logs, branches, HEAD, and more. Detailed List.

Index

A layer that separates your working tree from the Git repository. Gives developers more power over what gets sent to the Git repository. Often referred to as the staging area.

HEAD

HEAD is a pointer that points to the current branch. A repository only has 1 active HEAD.

head

head is a pointer that points to any commit. A repository can have any number of heads.

Working Tree / Working Directory

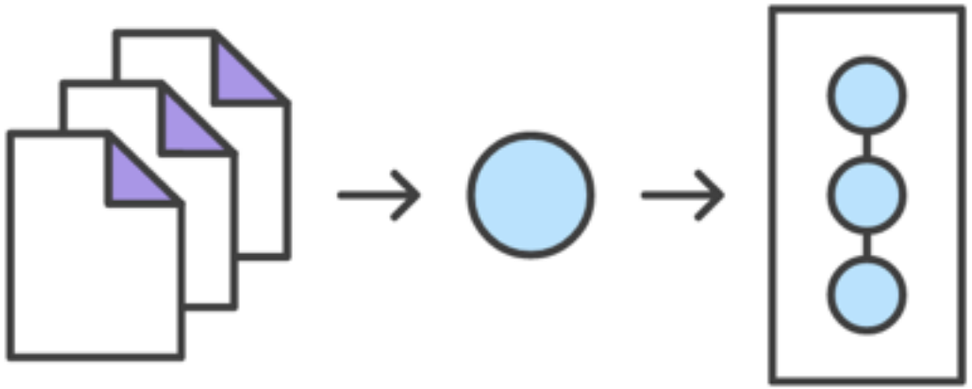
The directories and files in your repository.

COMMITS

- A Commit records changes within a file/directory
- Commit at milestones to be able to observe changes chronologically
- Each commit has a 40-character checksum hash as its identifier
- When committing your changes, you must enter a commit message
  - Provides descriptive comments regarding the changes you have made
  - Separating different types of change (bug fixes, new feature, improvements…) into different sets of commits helps understand why and how those changes were made

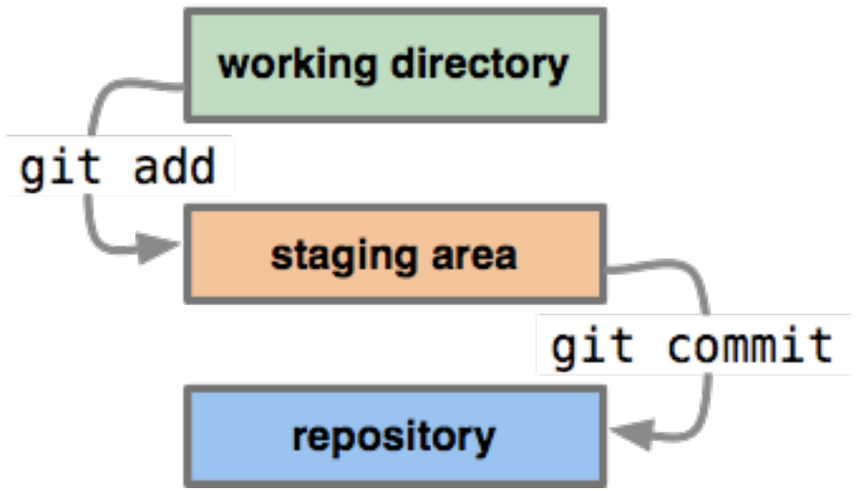
GIT WORKFLOW

- Developing a project revolves around the basic edit/stage/commit pattern.
  - First, you edit your files in the working directory.
  - When you're ready to save a copy of the current state of the project, you stage changes with git add.
  - After you're happy with the staged snapshot, you commit it to the project history with git commit.



STAGES OF GIT

- Modified - Changes have been made to a file but file has not been staged or committed to Git Database yet
- Staged - Marks a modified file to go into your next commit snapshot
- Committed - Files have been committed to the Git Database



GIT COMMANDS

Command	Purpose
init	Creates new, empty Git repo
clone	Copies an existing Git repo
config	Allows you to configure your Git installation or an individual repo from the command line
add	Adds a change in the working directory to the staging area. Tells Git that you want to include updates to a particular file in the next commit. Does NOT actually record changes.
status	View the state of the working directory and the staging area. Lets you see which changes have been staged, which haven't, and which files aren't being tracked.
commit	Commits staged snapshot to project history. Will never be changed unless you explicitly tell it to.
log	Displays committed snapshots. Lets you list, filter, or search project history. ONLY operates on committed history.
checkout	Switch from one branch to another
revert	Undoes a committed snapshot. Does not remove the commit from project history, instead figures out how to undo the changes introduced by the commit.
remote -v	List all currently configured remote repositories
remote add	If you haven't connected your local repository to a remote server, add the server to be able to push to it
fetch	Fetch and merge changes on the remote server to your working directory
merge	To merge a different branch into your active branch
diff	Show all merge conflicts
branch	List all the branches in your repo, and also tell you what branch you're currently in
push <a> <b>	Send changes to a (destination, remote repo) from b (current branch)
pull	Fetch and merge changes on the remote server to your working directory
reset	Like revert, but DOES remove the commit from project history. Basically a permanent undo. Be careful - this is one of the only Git commands that could allow loss of work.
clean	Removes untracked files from your working directory. Like an ordinary rm command, git clean is not undoable, so make sure you really want to delete the untracked files before you run it.

DEMO: INDIVIDUAL GIT USAGE

REPO PROCESSES

- Establish existing folder as repository:
  - Navigate into that folder
  - git init
  - git status (untracked files)
- Add files to staging area:
  - git add .
  - git status (to be committed)
- Make first commit:
  - git commit -m “first commit”
  - git status (working directory clean)
- Make changes to repo and commit them:
  - touch mouse.txt
  - move through workflow (status, add, status, commit, status)
  - git log (shows commit history)
- Revert to previous commit:
  - git revert <first commit id>
  - if stuck in bash screen, type :wq
  - ls (cat and dog should be gone)
  - git log (new commit created, could still go back to older version)
  - git reset would NOT allow this
- Clean repository:
  - touch frog.txt
  - git status (frog is untracked)
  - git clean (does nothing)
  - git clean -f (frog should be removed)
  - CANNOT UNDO CLEAN

GUIDED PRACTICE: INDIVIDUAL GIT

SET UP YOUR GIT

- git config --global user.name <name>
- git config --global user.email <email>
- git config --global —unset core.editor

CREATE NEW REPO AND COMMIT TO IT

Make the folder from this morning into a repository:

- Navigate into that folder (cd animals)
- `git init`
- `git status` (should see: untracked files)

Add files to staging area:

- `git add .`
- `git status` (should see: to be committed)

Make your first commit:

- `git commit -m “first commit”`
- `git status` (should see: working directory clean)

MAKE CHANGES IN A REPO

- `touch mouse.txt`

- move through workflow
  - `git status`
  - `git add .`
  - `git status`
  - `git commit -m “message”`
  - `git status`

- `git log`
  - What do you see in the commit history?
    - identifier, author, date, message
  - How are commits listed?
    - most recent to longest past

REPO REVERT

Revert to previous state:

- `git revert <first commit id>`
  - if stuck in bash screen, type “ : `wq` ”
- `ls`
  - cat.txt and dog.txt should be gone
- `git log`
  - new commit created, could still go back in time
  - git reset would NOT allow this

CLEAN A REPOSITORY

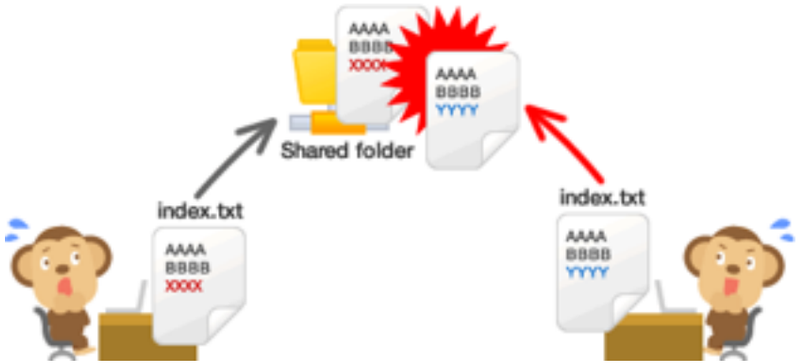
Clean a directory:

- `touch frog.txt`
- `git status` (frog is untracked)
- `git clean` (does nothing)
- `git clean -f` (frog should be removed)
  - CANNOT UNDO CLEAN
  - Have to add “-f” to force a clean
  - Git looks out for you

GIT TOGETHER

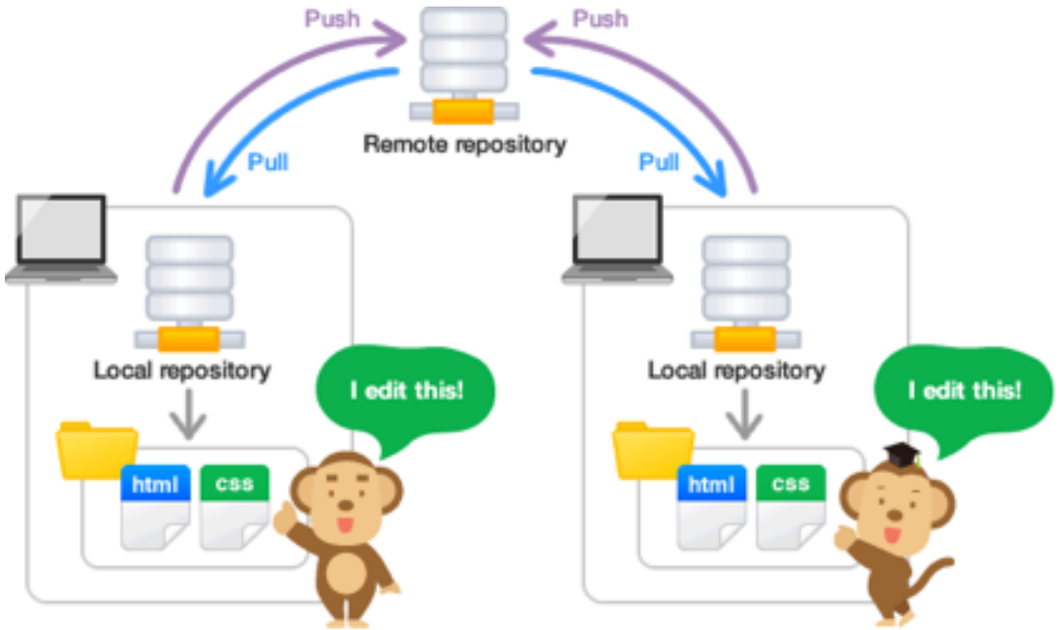
COLLABORATING WITH GIT

- Each developer gets their own copy of the repo
- Usually want to share a series of commits (rather than every one)
- Commits are collected along a branch, then shared to the common repository
- Publish local history by “pushing” branches to other repositories
- See what others have contributed by “pulling” branches into your local repository



REMOTE VS LOCAL REPOSITORIES

- Local repository: on local machine of individual user
  - Can use all of Git's version control features (reverting changes, tracking changes, etc.)
  - Can be new (init) or a copy (clone)
- Remote repository: on a remote server, often shared by team members
  - Used for sharing your changes or pulling changes from your team

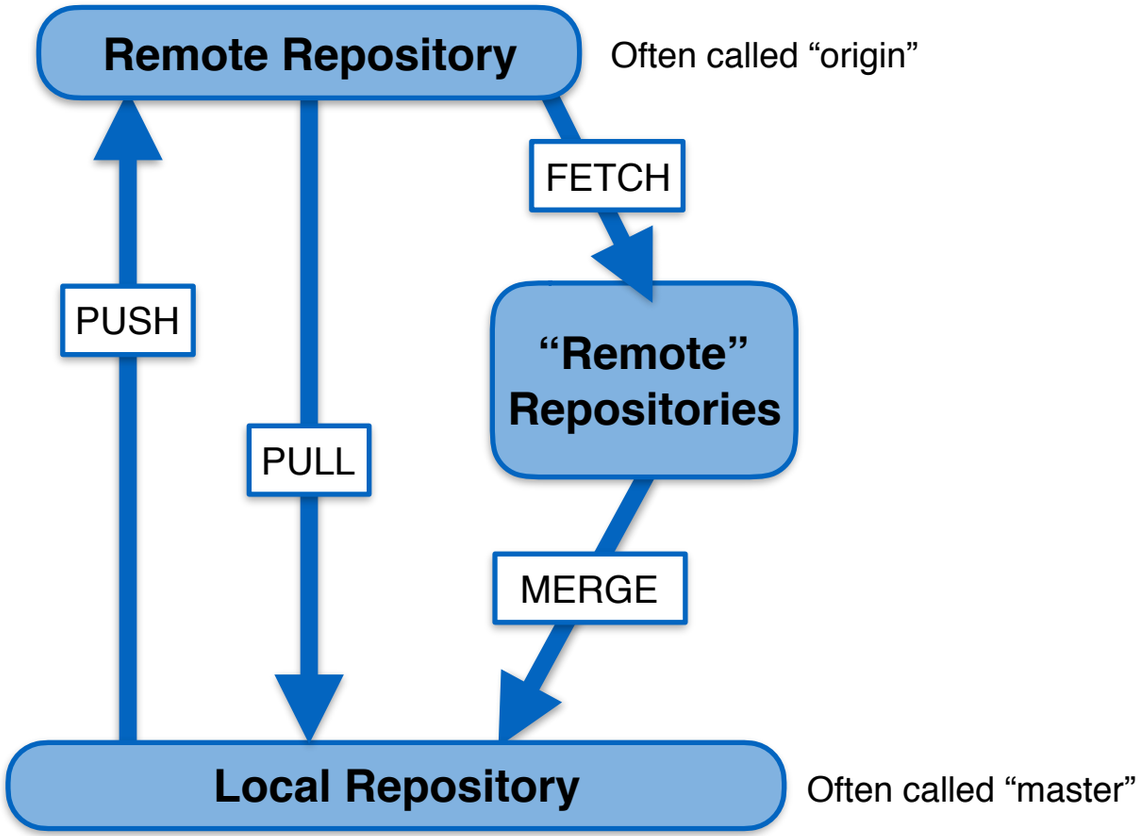


SETTING UP CONNECTION

- Local repo: on local machine, every team member has their own
- Remote repo: on remote server, can have individual branches/forks but usually members share to a central repo
  - List current remote connections from within a repository using "git remote -v"

Repo Setup	Creation	Connection
Fresh/New	git init (from within directory)	git remote add <name> <url>
Copy	git clone	automatic from original repo

TRANSFERRING INFORMATION



BRANCHES

A branch is:

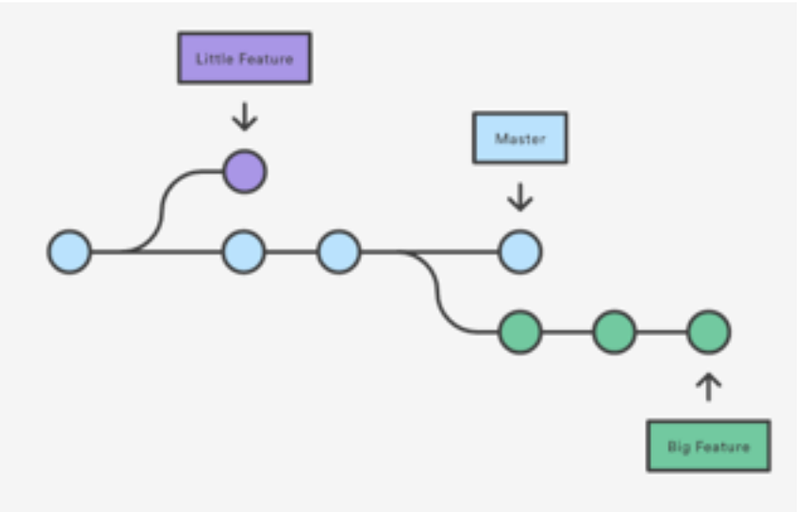
- An independent line of development
- A fork in the history of the project
- A way to request a brand new working directory, staging area, and project history
- A reference to a commit at the tip of a series of commits, all recorded in the history for the current branch

A branch is not:

- Copying files from directory to directory
- A container for commits

A branch allows:

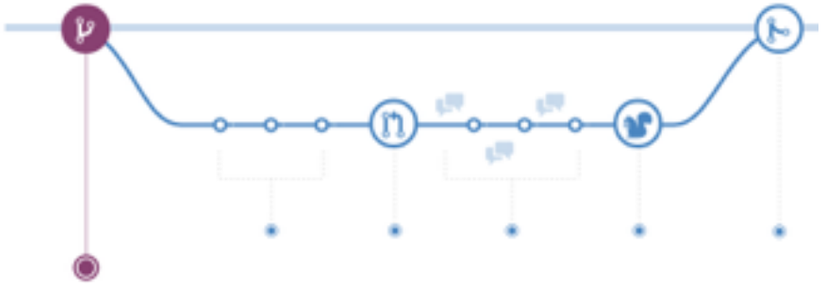
- work on multiple parts of a project in parallel
- keeps the main master branch free from questionable code



BRANCH COMMANDS

git branch	List all of the branches in your repository.
git branch <branch>	Create a new branch called <branch>. This does not check out the new branch. The repository history remains unchanged. All you get is a new pointer to the current commit. To start adding commits to it, you need to select it with git checkout, and then use the standard git add and git commit commands.
git branch -m <branch>	Rename the current branch to <branch>.
git branch -d <branch>	Delete the specified branch. This is a “safe” operation in that Git prevents you from deleting the branch if it has unmerged changes.
git branch -D <branch>	Force delete the specified branch, even if it has unmerged changes. Use to permanently throw away all commits from a particular line of development.
git checkout	Lets you navigate between the branches created by git branch. Checking out a branch updates the files in the working directory to match the version stored in that branch, and tells Git to record all new commits on that branch. A way to select which line of development you’re working on.
git checkout <existing-branch>	Check out the specified branch, which should have already been created with git branch. This makes <existing-branch> the current branch, and updates the working directory to match.
git checkout -b <new-branch>	Create and check out <new-branch>. The -b option is a convenience flag that tells Git to run git branch <new-branch> before running git checkout <new- branch>. git checkout -b <new-branch> <existing-branch>

BRANCHING WORKFLOW



Create a branch

Add commits

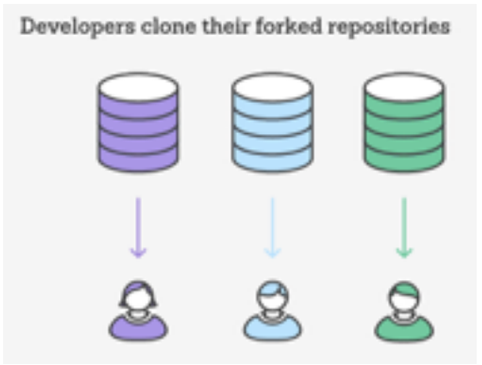
Open a Pull Request

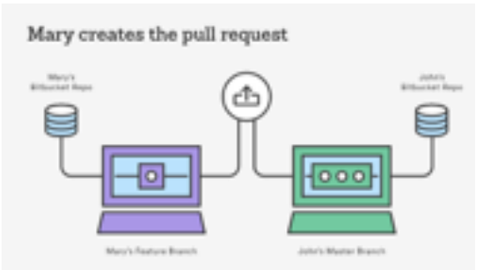
Discuss and review your code

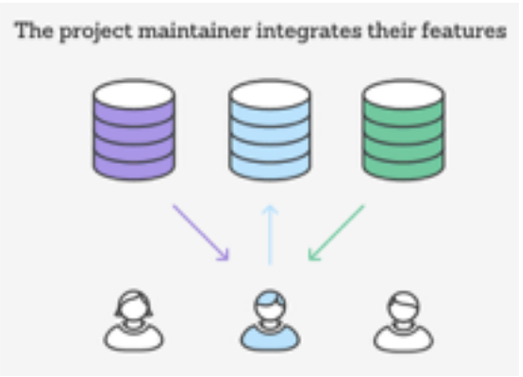
Deploy

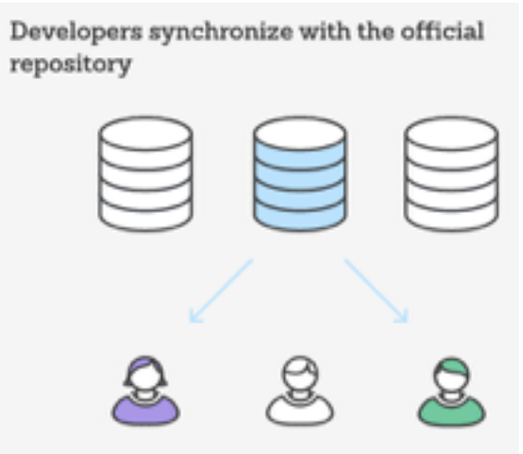
Merge

FORKING WORKFLOW









The Forking Workflow is fundamentally different than other workflows.

Instead of using a single server-side repository to act as the “central” codebase, it gives every developer a server-side repository.

This means that each contributor has not one, but two Git repositories: a private local one and a public server-side one.

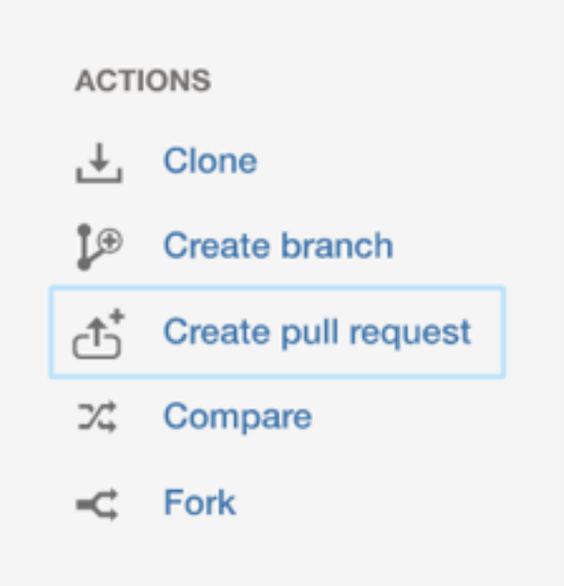
CHECK FOR UNDERSTANDING

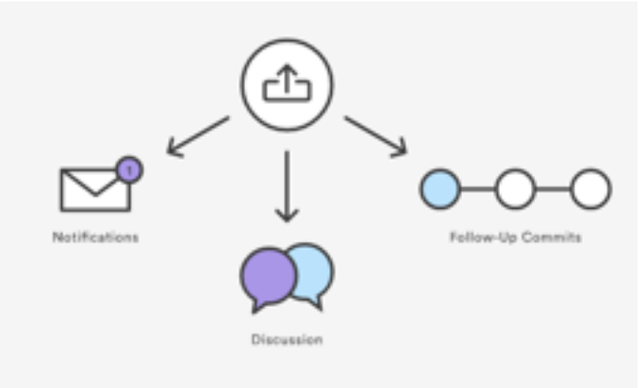
When might it be better to use a fork? A branch?

BRANCHING VS FORKING

Branch	Fork
Like a branch of a tree: • Remains part of the original repository • The code that is branched (main trunk) and the branch know and rely on each other • Knows about the trunk (original code base) it originated from.	Clone or copy: • Independent from original repository (but can always see which repo the fork came from) • If original repository deleted, the fork remains • If you fork a repository, you get that repository and all of its branches
Branches are useful when: • You have a small group of programers who trust each other and are in close communication. • You are willing to give the development team write access to a repository. • You have a rapid iteration cycle.	Forks work well in situations where: • You don't want to manage user access on your repository. • You want fine-grain control over merging. • You expressly want to support independent branches. • You want to discard experiments and changes easily.

PULL REQUESTS



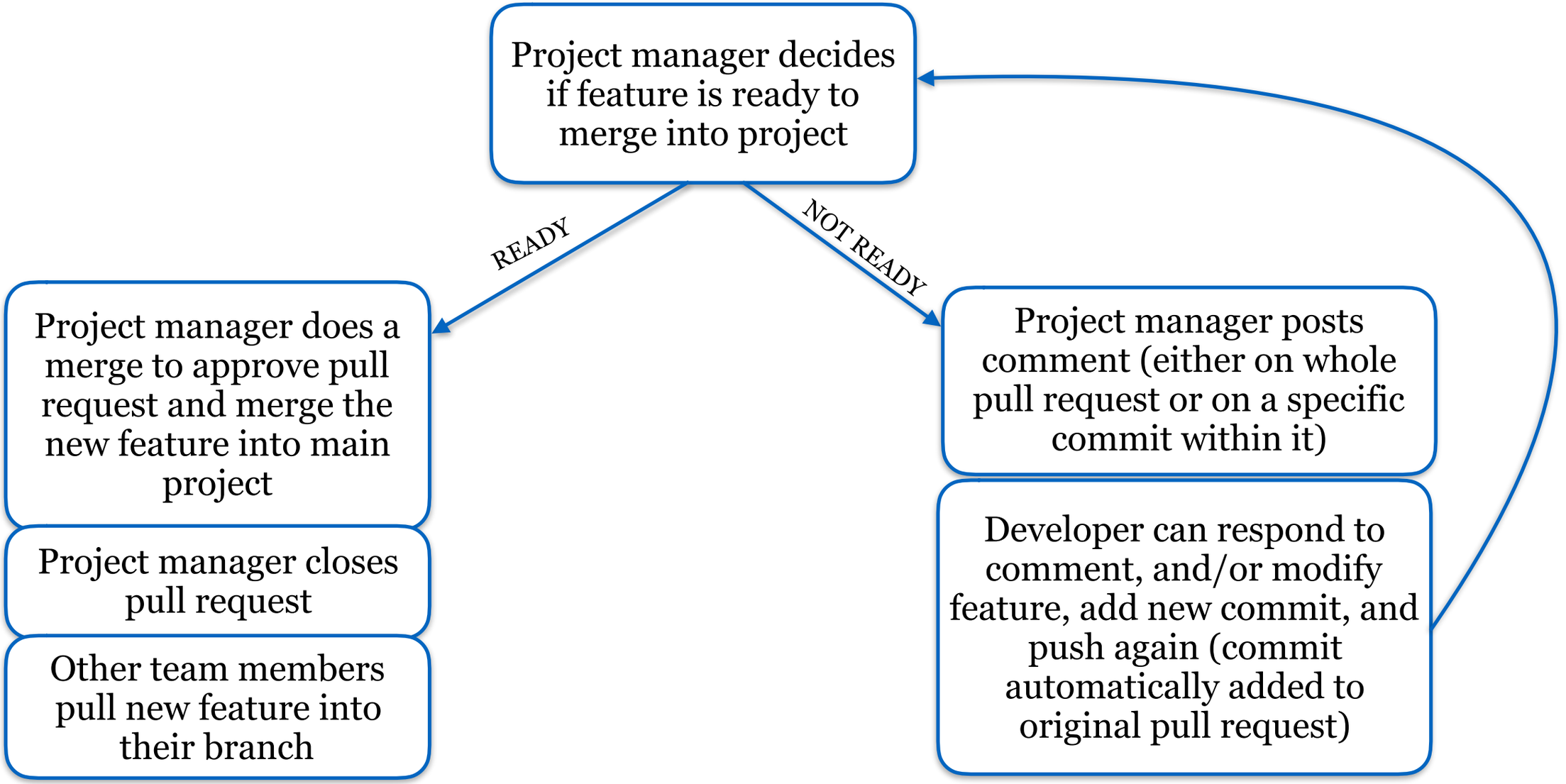


- Alerts team to view proposed code and merge it into main project
- Forum to discuss proposed changes/features before integrating them
- Teammates can post feedback in the pull request
- Teammates can tweak the feature by pushing follow-up commits
- All of this activity is tracked directly inside of the pull request

In the pull request process, the developer:

- *(Only if branching) Creates the feature in a dedicated branch in local repo*
- Pushes the branch to a public repository
- Files a pull request
- The rest of the team reviews the code, discusses it, and alters it.
- The project maintainer merges the feature into the official repository and closes the pull request

REVIEWING PULL REQUESTS



MERGE CONFLICTS

If the two branches you're trying to merge contain conflicts in the same part of the same file, Git won't know which to use and will stop so that you can manually resolve the conflict.

In a merge conflict, running `git status` shows you which files need to be resolved, like this:

```
# On branch master
# Unmerged paths:
# (use "git add/rm ..." as appropriate to mark resolution)
#
# both modified: hello.py
#
```

The developer needs to:

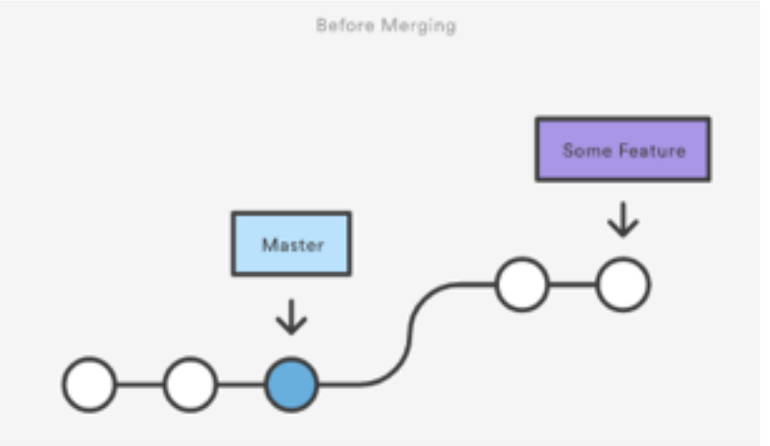
- Fix the conflict by editing the files
- Run `git add` on the file(s) to signal that the issue is resolved
- Run `git commit` to generate the merge commit

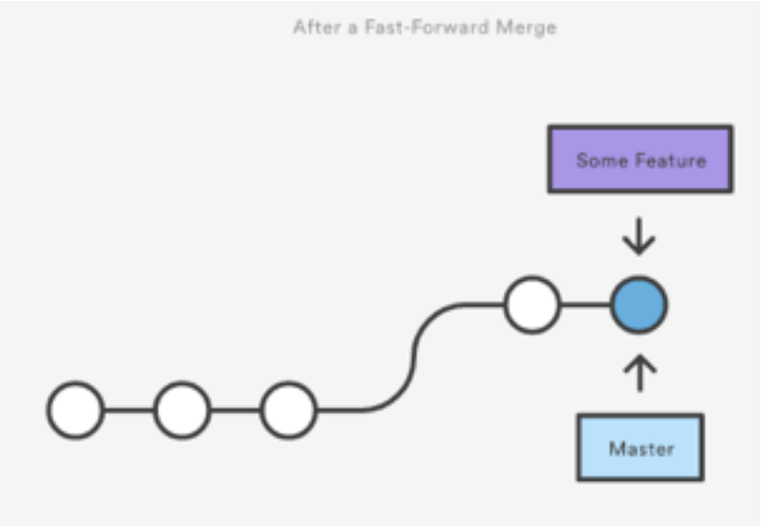
Look familiar?

FAST FORWARD MERGES

- Occur when there is a linear path from current branch tip to target branch
- Git can just “fast forward” the branch rather than do an actual, complex merge

- Git will refuse push requests if they result in a non-fast-forward merge to prevent overwriting the main history
- If the branches have diverged, Git 3-way merges use a dedicated commit to tie together the two histories
- You will need to pull the remote branch first, merge it into your local one (resolve conflicts if needed), then push again





GUIDED PRACTICE: GIT TOGETHER

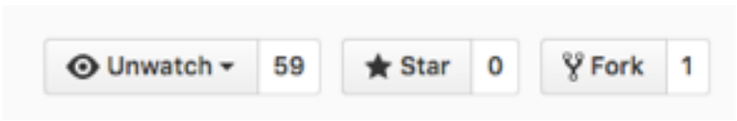
SET UP SSH

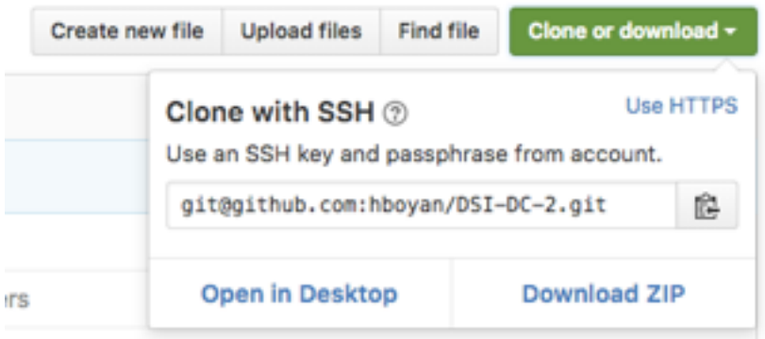
- https://help.github.com/articles/generating-a-new-ssh-key-and-adding-it-to-the-ssh-agent/

GET READY TO USE OUR CLASS REPO

EXERCISE

- Go to our class repo:
  - https://github.com/ga-students/DSI-DC-2
- Fork the repository into your repo
- Click “Clone or download” and copy the URL
- In Terminal:
  - `cd ~`
  - `cd desktop`
  - `git clone <URL>`
  - You should be able to see a folder called DSI-DC-2 on your desktop





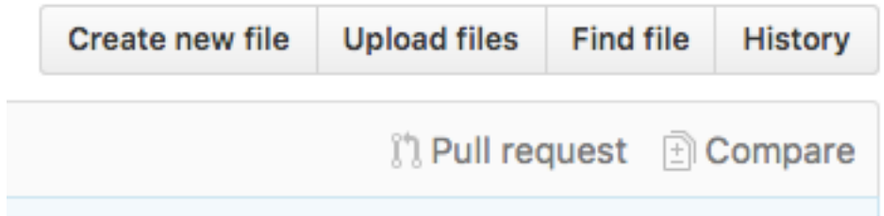
MAKE A CHANGE

- In Terminal:
  - `cd DSI-DC-3/curriculum/week-01/1.02-Intro-to-Git`
  - `touch <yourname>.txt`
  - `ls` to check if file was created
- `cd ..` until you're in the DSI-DC-2 main folder
- `git remote -v` to check that remote repo is <you>/DSI-DC-2
- `git add .`
- `git commit -m "<Your Name> test submission"`
- `git push origin master`
  - origin = remote repo, master = current directory

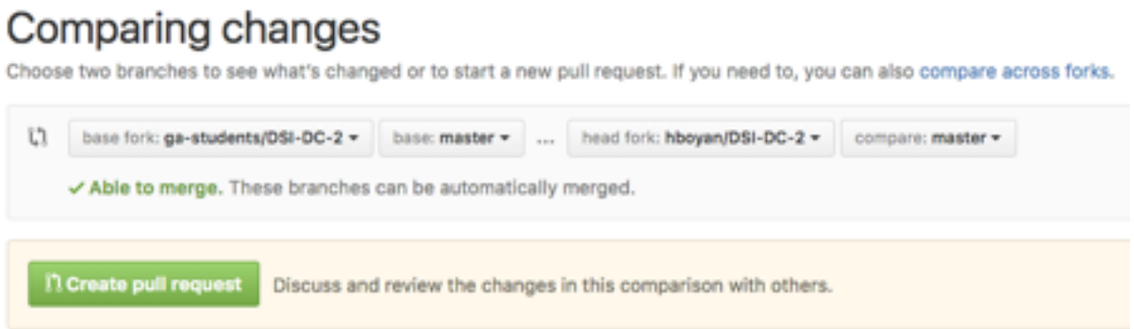
CREATE PULL REQUEST

EXERCISE

- In your github repo, you should be able to find the file you just made in “DSI-DC-2/curriculum/week-01/1.02-Intro-to-Git”
- Click “Pull request”

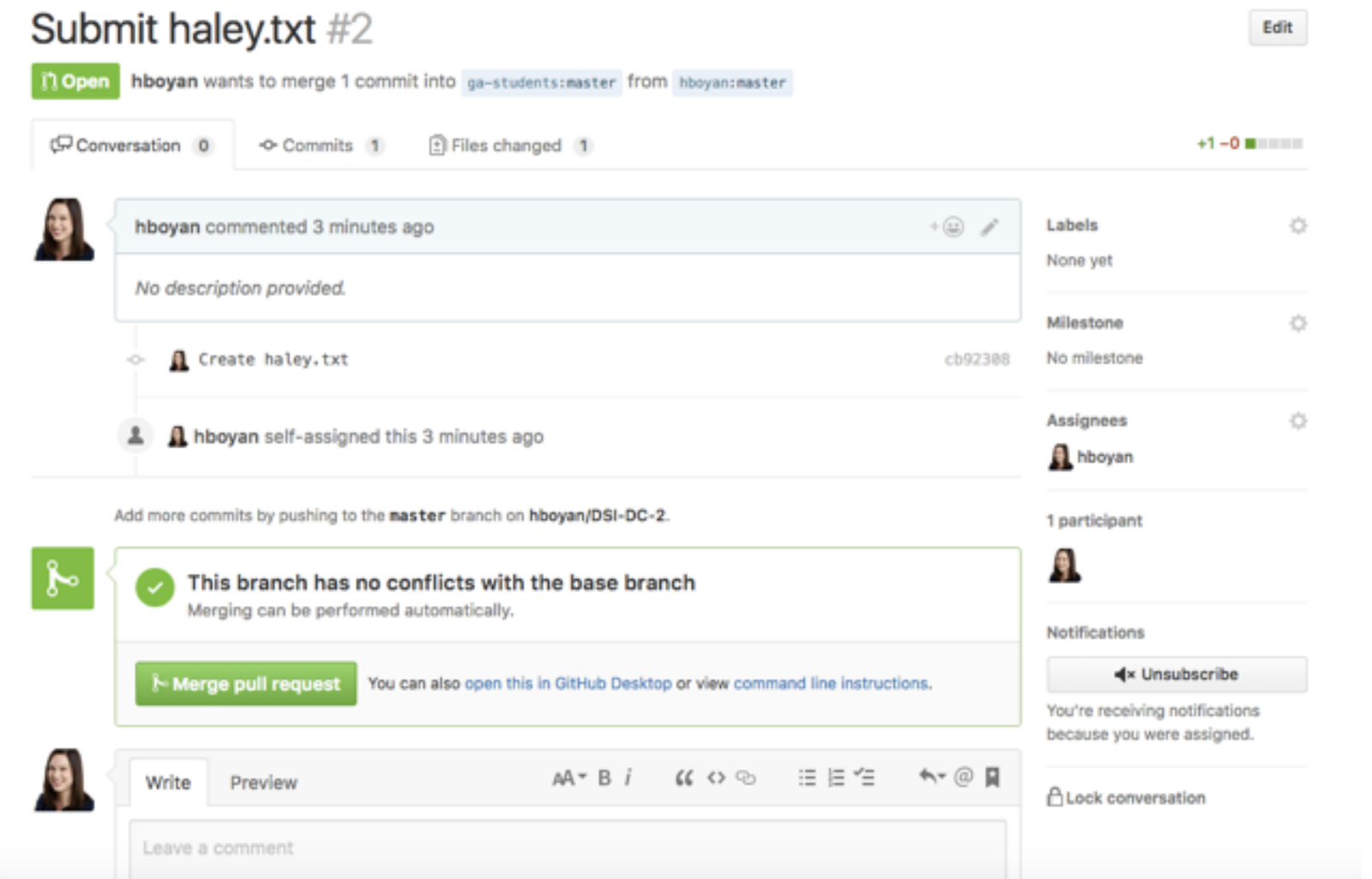


- Click “Create pull request”
- Add a title, a short description
- Again, click “Create pull request”



WHAT PROJECT MANAGER SEES

EXERCISE



MERGE CONFLICT

- In Terminal, open the conflicttest.txt file in the StudentSubmissions folder
- Add some text on the second line, save, and close the file
- Now go through the flow again:
  - `git add`
  - `git commit -m “conflict testing”`
  - `git push`
  - In your repo, find the file and create a pull request for it
  - What’s different?

REVIEW

YOU SHOULD BE ABLE TO ANSWER:

- What is Git? What are some benefits of it?
- How is Git different than Github?
- What is a repository? What are the types of repos?
- What is the process for marking milestones in Git?
- What is the difference between forking and branching? Pros/cons of each?
- What is a fast forward merge? A 3-way merge?
- How are merge conflicts resolved?
- What is the pull request process?

BEFORE NEXT CLASS

BEFORE NEXT CLASS

DUE TUESDAY 9AM

- Homework: Submit Via Pull Request to Homework Folder
- GitPracticeA:
    - http://learngitbranching.js.org/
    - **Screen shot** of completed steps 1-25
- GitPracticeB:
    - https://try.github.io/levels/1/challenges/1
    - **Screen shot** of completed levels, at minimum:
        - Main: Introduction Sequence 1-3
        - Remote: Complete

- Name files as: FirstLast-Title.jpg (or .png, etc.)
    - e.g. HaleyBoyan-GitPracticeA.jpg

CREDITS

TITLE

CITATIONS

- https://backlogtool.com/git-guide/en/intro/intro1_1.html
- https://learnxinyminutes.com/docs/git/
- https://www.atlassian.com/git/tutorials/
- https://confluence.atlassian.com/bitbucketserver/basic-git-commands-776639767.html
- http://rogerdudler.github.io/git-guide/
- https://guides.github.com/
- http://ndpsoftware.com/git-cheatsheet.html
- https://confluence.atlassian.com/bitbucket/branch-or-fork-your-repository-221450630.html

TITLE

Q & A

TITLE

EXIT TICKET

DON’T FORGET TO FILL OUT YOUR EXIT TICKET