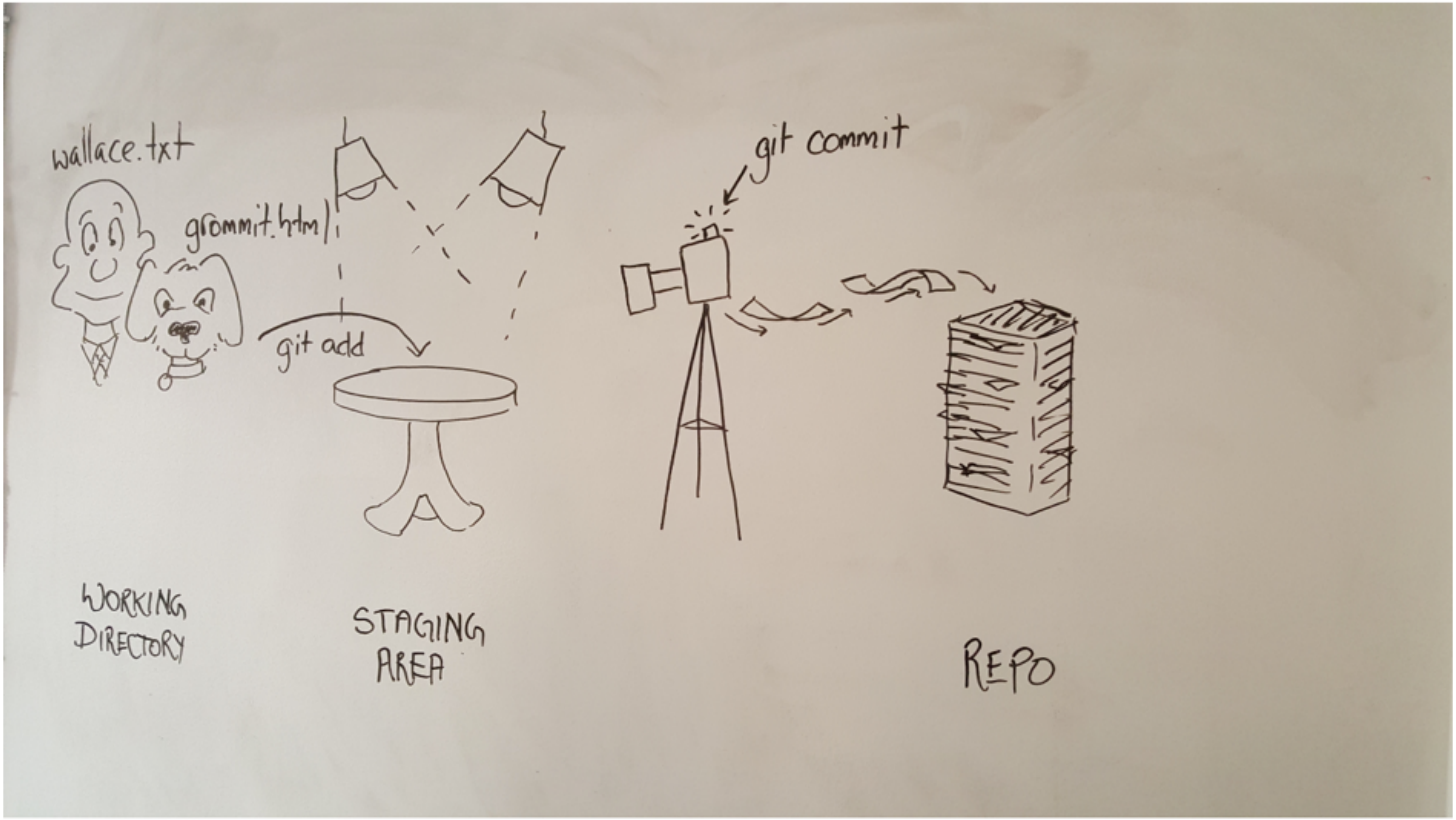
COLLABORATING WITH GITHUB

Lesson objectives:

- Compare different git collaboration workflows
- Contribute your edits to a shared repo using pull requests
- Use branches to isolate changes tied to specific features
- Efficiently and correctly resolve merge conflicts
- Fetch changes from a remote without merging them into your own
- Explain how rebase combines two branches


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

REVIEW TERMS

remote - another repository that can be synchronized with a remote

github - a service that hosts git remote repositories, and provides a web app to interact / collaborate on them

clone - download an entire remote repository, to be used as a local repository

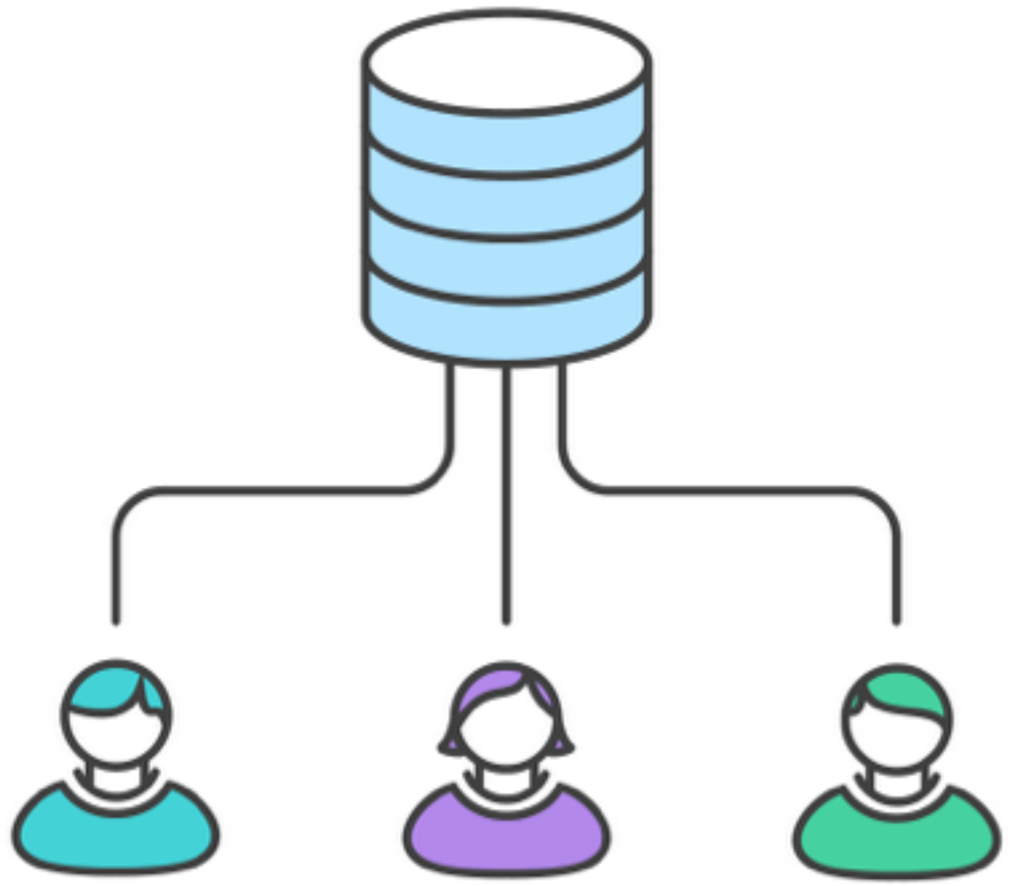
fork - copy a repo from another person's account

pull - fetching* changes and merging them into the current branch

push - sending changes to a remote repository and merging them into the specified branch

merge conflict - when two commits conflict, and thus can't be merged automatically

**fetch - downloading the set of changes (commits) from a remote repository*



“We need to collaborate!”

ONE WORKFLOW: FORKS

COLLABORATING ACROSS FORKS

- Initialize a repo
- Each person forks and clones a copy to their local machine. Set the original as an upstream remote.
- Make edits to the code, commit to your local. *git push origin master*
- Create a *pull request* to the base fork
- One person manages the base fork and checks for merge conflicts
- To sync with the base fork, run git fetch instead of git pull

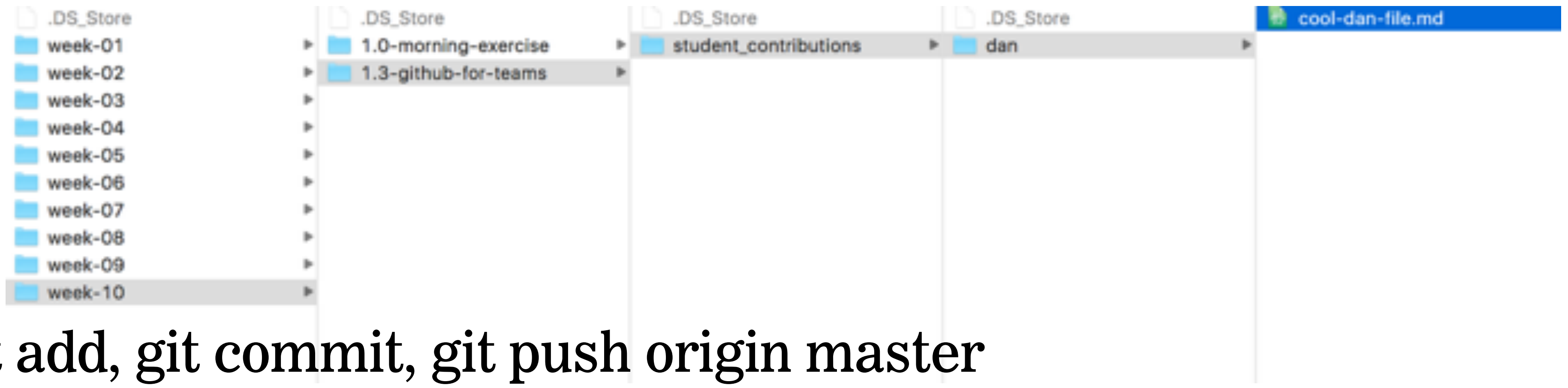
SIDE NOTE: FETCH VS PULL?

- git fetch <origin> <branch> downloads all the changes from a repo
- git merge <branch> incorporates those changes into your local branch, so you can check for conflicts one by one

- git pull <origin> <branch> does both at once

COLLABORATING ACROSS FORKS

- Create a file in your working directory under student_contribtuions/your_name:



.DS_Store
week-01
week-02
week-03
week-04
week-05
week-06
week-07
week-08
week-09
week-10

.DS_Store
1.0-morning-exercise
1.3-github-for-teams

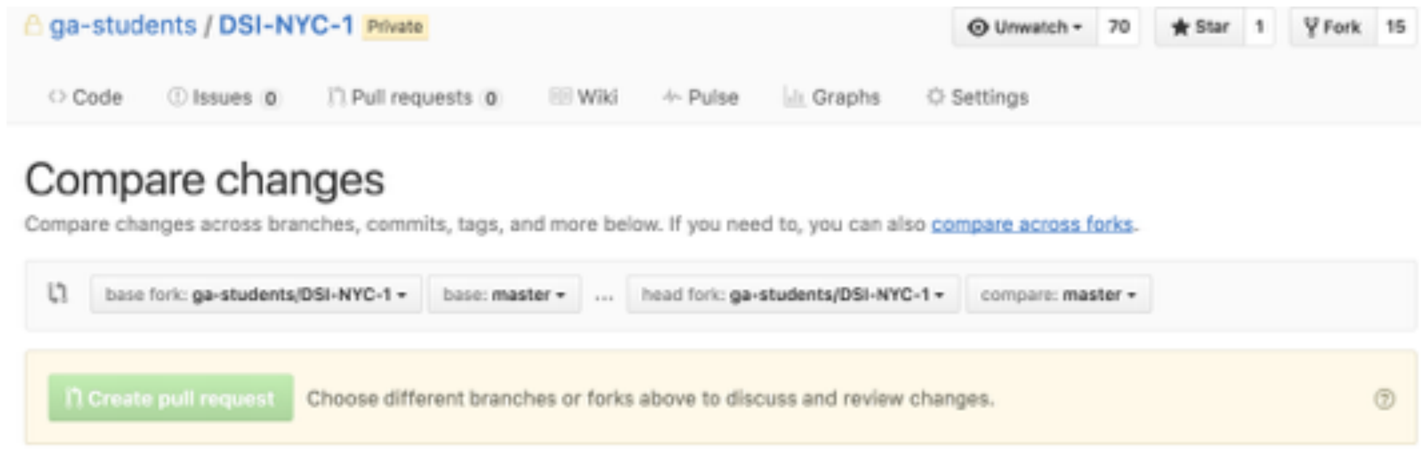
.DS_Store
student_contributions

.DS_Store
dan

cool-dan-file.md

- git add, git commit, git push origin master

- Create a pull request on the Github GUI. Click “New Pull Request” then “compare across forks”



Compare changes

Compare changes across branches, commits, tags, and more below. If you need to, you can also compare across forks.

base fork: ga-students/DSI-NYC-1 · base: master ... head fork: ga-students/DSI-NYC-1 · compare: master

Create pull request · Choose different branches or forks above to discuss and review changes.

COLLABORATING ACROSS FORKS

(+) One person integrates all changes and checks all commits, so there's consistency.

(-) Could get overwhelming for large projects.

(-) One person becomes the point of failure.

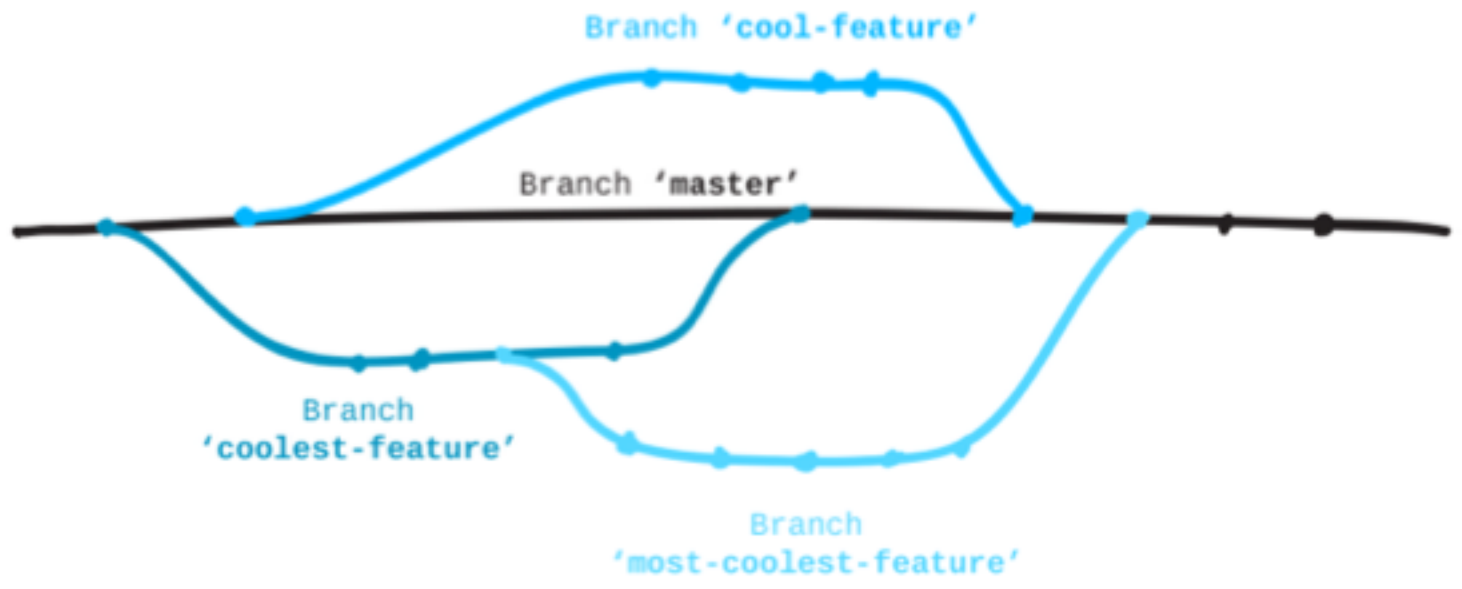
(-) Not memory lightweight. Each fork creates a whole copy of all the files

NEW COMMAND ALERT

GIT BRANCH

BRANCHING

When you want to add a new feature or fix a bug—no matter how big or how small—you spawn a new branch to encapsulate your changes.



WHY BRANCHING?

To allow experimentation.

To allow easy bug fixes on a stable version while features are being developed.

To allow work to proceed on multiple features (or by multiple people) without interfering. When a feature is complete, it can be merged back into master.

"Branch Early, Branch Often": Branches are lightweight, there is no additional overhead associated with branches, so it can be a great way to organize our workflow

HOW TO BRANCH

Start a new branch

- git branch new-feature #create a new branch
- git checkout new-feature #switch to that branch

Edit some files

- git add <file>
- git commit -m “Started, finished a feature"

Merge in the new feature branch

- git checkout master #switch to the master branch
- git merge new-feature #merge in the new-feature branch
- git branch -d new-feature #delete the branch

BRANCHING PRACTICE 1 (10 MINUTES)

Do Levels 1-3: http://learngitbranching.js.org/

Take your time. Read all the dialogs. They are part of the tutorial.

Stop before 4: "Rebase Introduction”.

BRANCHING PRACTICE 2 (10 MINUTES)

- Create a new blank repo “just_for_practice” and clone it locally
- Create a new file, *hello.txt*. Push it up to Github.
- Edit the file on *new-cool-branch*
- *git add hello.txt, git commit -m “some edits”*
- *git push origin new-cool-branch* to push up the branch
- Navigate into Github to manage and the merge with branch *master*.
- In terminal, switch back to your *master* branch.
- *git fetch* and *git merge* to sync with your remote

NEW COMMAND ALERT

GIT REBASE

WHY REBASE?

- Imagine you're working on your *new-cool-branch* when someone makes an important update to the *master* branch.
- Oh no! Now those edits will lead to a merge conflict if you try to merge *new-cool-branch* into *master* right away.
- Run a *git rebase* to retroactively put those update commits into *new-cool-branch* before you merge

WHY REBASE?

- Rebase is extremely useful for cleaning up your commit history, but it also carries risk; when you rebase, you are discarding your old commits and replacing them with new (though admittedly, similar) commits.

- This can seriously screw up a fellow collaborator if you're working in a shared repo. The golden rule for git rebase is "only rebase before sharing your code, never after."

REBASE PRACTICE 1 (10 MINUTES)

Do Level 4: http://learngitbranching.js.org/

Experiment creating branches, adding several commits to them, then rebasing and merging with master.

ALTERNATE GIT WORKFLOWS

COMPARING WORKFLOWS (20 MINUTES)

https://www.atlassian.com/git/tutorials/comparing-workflows

1. Centralized Workflow
2. Feature Branch Workflow
3. Gitflow Workflow
4. Forking Workflow

Skim the article to get a sense of all 4 workflows. With your table, discuss how one workflow works, diagram it out, describe how to contribute to it, and prepare to present it to the class.

Let’s recap

RECAP

- pull requests — a way to suggest changes (from a branch or fork) to a shared repo
- branch — allows you to edit part of a shared repo before merging your code back in
- rebase — incorporates other peoples commits to your branch, before you merge with master
- fetch — downloads changes to the repo (commits, branches) without merging with your own

APPENDIX: ALTERNATE GIT WORKFLOWS

SINGLE REMOTE WORKFLOWS

One thing all of these approaches have in common is the necessity of staying on top of changes within a single shared repository.

This is usually accomplished by running git fetch, which pulls updates from origin, and merging those updates; alternatively, you could use git pull to do both at once.

CENTRALIZED WORKFLOW

How It Works: The remote repo has one single branch on it, master. All collaborators have separate clones of this repo. They can each work independently on separate things.

However, before they push, collaborators need to run git fetch/git pull (with the --rebase flag) to make sure that their master branch isn't out of date.

(+) Very simple

(-) Collaboration is kind of clunky.

FEATURE BRANCH WORKFLOW

How It Works: This workflow is very similar to the centralized workflow. The biggest difference is that there are branches (which helps to keep feature-related commits isolated), and that instead of pushing changes up directly, collaborators:

(a) push up changes to a new remote branch rather than master

(b) submit a pull request to ask for them to be added to the remote repo's master branch.

(+) Better isolation than Centralized model, but sharing is still easy. Very flexible.

(-) Sometimes it's too flexible - it doesn't meaningfully distinguish between different branches, and that lack of structure can cause problems on larger projects.

GITFLOW WORKFLOW

How It Works: Similar to the Feature Branch workflows, but with more rigidly-defined branches. For example:

Historical Branches : master stores official releases, while develop serves as a living 'integration branch' that ties together all the standalone features.

Release Branches : 'release' branches might exist for any given release, to keep all of those materials together.

Feature Branches : pretty much the same as in the prior model.

Maintenance/'Hotfix' Branches : branches used to quickly patch issues with production code.

(+) Highly structured - works well for large projects.

(-) Sometimes overkill for something small.

DISTRIBUTED WORKFLOWS

These approaches all use multiple remote repos; typically, everyone has their own fork of the 'original' project (the version of the repo that's publicly visible and is managed by the project maintainer), and changes are submitted via pull request.

INTEGRATION MANAGER WORKFLOW

How It Works: One collaborator plays the role of 'Integration Manager'. This means that they are responsible for managing the official repository and either accepting or rejecting pull requests as they come in.

(+) One person integrates all changes, so there's consistency.

(-) Could get overwhelming for large projects.

DICTATOR-LIEUTENANTS WORKFLOW

How It Works: This workflow is very similar to the Integration Manager Workflow. The biggest difference is that rather than submitting all pull requests to a single integration manager, pull requests are funneled through 'Lieutenants', who all report to the 'Dictator'. Only the Dictator has write access to the official repo.

(+) One person integrates all changes, so there's consistency.

(-) Only one person has final write access, so there's consistency but also a single point of failure.