

GIT AND GITHUB

WHAT WE WILL COVER TODAY

- What is version control and why should we care?
- Basics of Git: the essential commands
- GitHub

IF IT SEEMS HARD, IT'S BECAUSE IT IS

A Google engineer, speaking to an audience of Google engineers, once described the newly invented Git as:

"a version control system which is expressly designed to make you feel less intelligent"
([source](#))

... but it gets easier with practice!

COMMAND LINE - QUICK OVERVIEW

COMMON COMMANDS (IN WINDOWS)

- `cd directory_name` : change directory
- `dir` : list all the files
- `mkdir directory_name` : make directory
- `rmdir directory_name` : remove/delete directory
- `type NUL >> filename` : create a file
- `rm filename` : remove a file
- `cd` : find out the file path of current directory you are in, from the root

terminology note: directory === folder

IMPORTANT: You must use 2 arrows (>>) when creating a new file or else this command will erase everything inside the file if it already exists.

CD (CHANGING DIRECTORIES)

A little further explanation...

- `cd folder_name`: moves down into the folder
example: `cd my_sait`
- `cd path_name`: moves down into the last folder listed
example: `cd my_sait/fuzzball/favorite_foods`
- `cd ..`: moves up one folder level
- `cd ../../../../`: moves up three folderlevels

WATCH ME!

Goal: Create a folder named `my_project` on my desktop

ACTIVITY: DO IT ON YOUR OWN!

1. Use the command line to navigate to your desktop folder
2. Create a folder called my_project
3. Move inside of your my_project folder
4. Check to see your current file path!

POTENTIAL SOLUTION

1. Use the command line to navigate to your desktop folder

```
$ cd Desktop
```

2. Create a folder called my_project

```
$ mkdir my_project
```

3. Move inside of your my_project folder

```
$ cd my_project
```

4. Check to see your current file path!

```
$ cd
```

VERSION CONTROL

WHAT IS VERSION CONTROL?

Version control is a tool that allows you to...

COLLABORATE

Create anything with other people, from academic papers to entire websites and applications.

TRACK AND REVERT CHANGES

Mistakes happen. Wouldn't it be nice if you could see the changes that have been made and go back in time to fix something that went wrong?

YOU ALREADY MANAGE VERSIONS OF YOUR WORK!

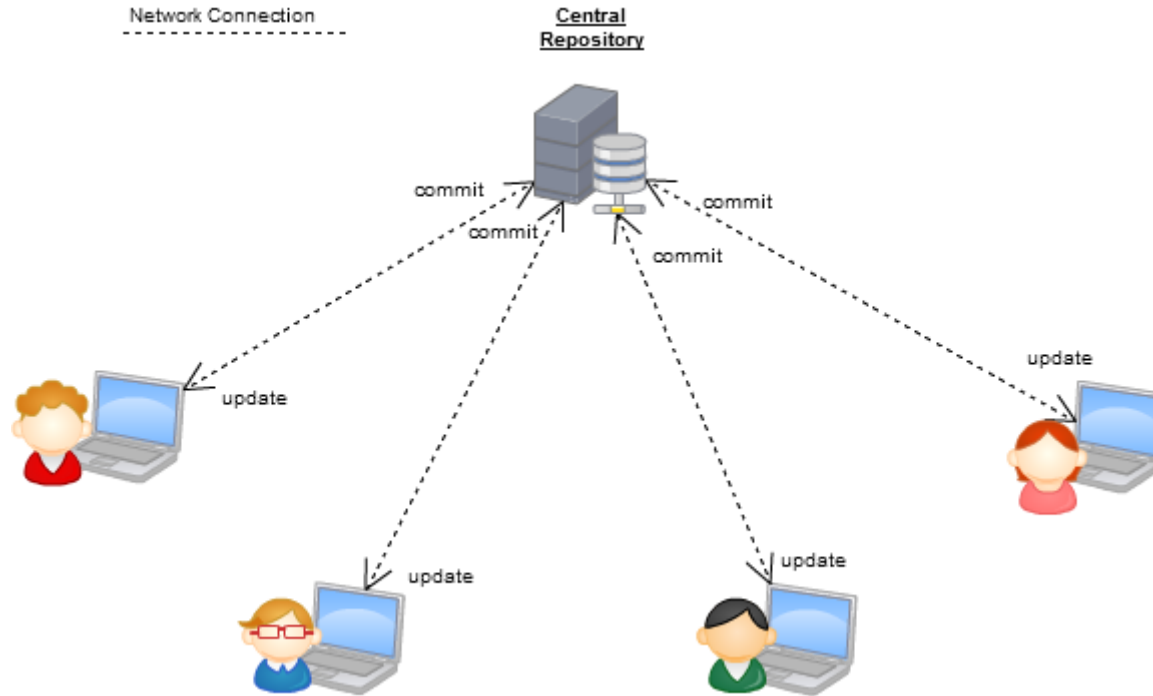
Do you have files somewhere that look like this?

```
Resume-September2016.docx  
Resume-for-Duke-job.docx  
ResumeOLD.docx  
ResumeNEW.docx  
ResumeREALLYREALLYNEW.docx
```

You invented your own version control!

TYPES OF VERSION CONTROL SYSTEMS

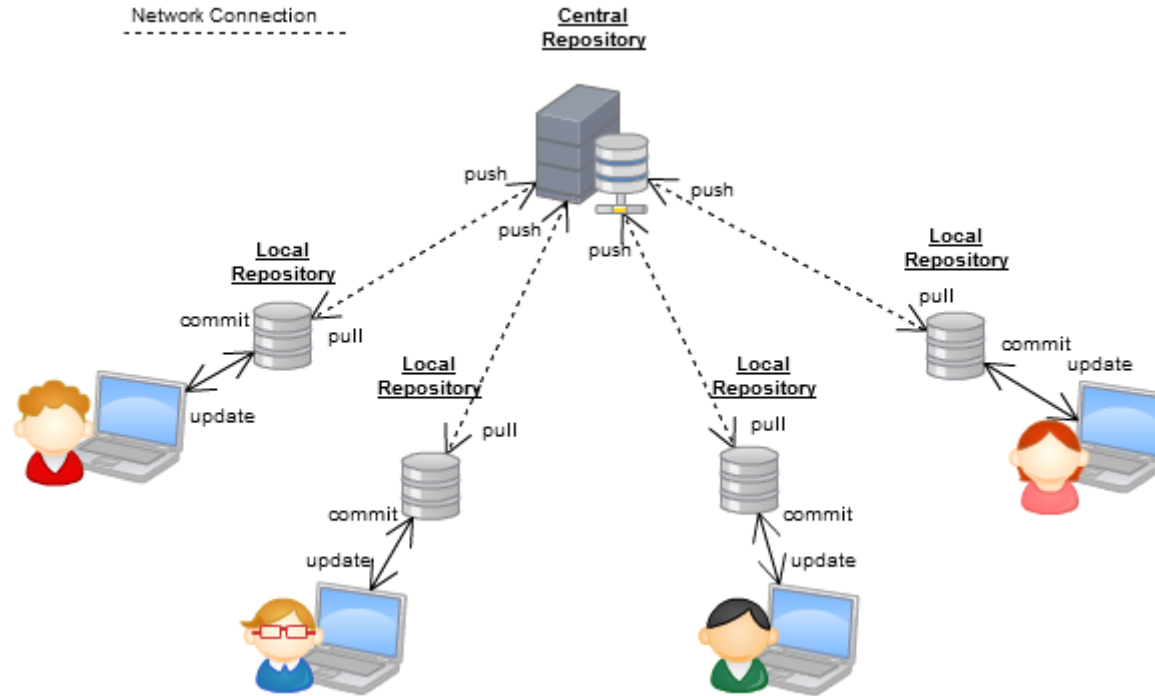
CENTRALIZED VERSION CONTROL



One central server, each client (person) checks out and merges changes to main server

Examples: CVS, Subversion (SVN), Perforce

DISTRIBUTED VERSION CONTROL



Each client (person) has a local repository, which they can then reconcile with the main server.

Examples: Git, Mercurial

WHY USE GIT?

- **Fast!** Access information quickly and efficiently.
- **Distributed!** Everyone has her own local copy.
- **Scalable!** Enables potentially thousands (millions!) of developers to work on single project.
- **Local!** You don't need a network connection to use it. You only need a remote server if you want to share your code with others (e.g., using GitHub).
- **Branches!** Keep your coding experiments separate from code that is already working.
- Everyone has a local copy of the **shared files** and the **history**.

INSTALLATION AND SETUP

Install Git



Downloads



Mac OS X

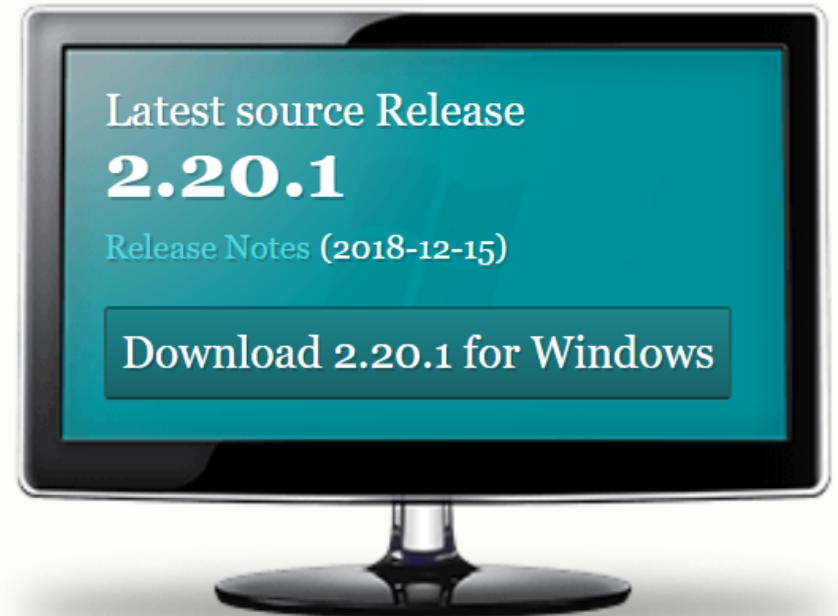


Windows



Linux/Unix

Older releases are available and the [Git source repository](#) is on [GitHub](#).



Setup steps on GitHub

INSTALLATION AND SETUP

Set up name and email in gitconfig

```
$ git config --global user.name "Your Name Here"  
# Sets the default name for Git to use when you commit
```

```
$ git config --global user.email "your_email@example.com"  
# Sets the default email for Git to use when you commit
```

```
$ git config --list
```

SETUP: SETTING THE DEFAULT TEXT EDITOR

By default Git is set up to use Vim as the text editor.

(**esc** + **:q** or **:q!** to get out of Vim)

```
$ git config --global core.editor "code --wait"  
# Sets the default text editor to VS Code
```

You can then open a file with:

```
$ code filename  
# Opens filename in VS Code
```

You can find commands for other text editors in [these instructions](#).

ACTIVITY: SETUP!

Install Git and set your user name and email.

BONUS LEVEL: If you want to, change your default text editor.

GIT HAS ITS OWN VOCABULARY

- A **repository** is where you keep all the files you want to track.
- A **branch** is the name for a separate line of development, with its own history.
- A **commit** is an object that holds information about a particular change.
- **HEAD** refers to the most recent commit on the current branch.

MAKING A REPO

WHAT IS A REPOSITORY (REPO)?

Essentially, a Git version of a project folder.

Git will track any changes inside of a repository.

CREATE A LOCAL REPOSITORY - DO IT WITH ME!

1. Make sure you are inside of your my_project

```
$ cd
```

2. Initialize it as a local Git repository

```
$ git status  
# should show an error because  
# we haven't made it a repository yet!  
$ git init  
$ git status
```


WHAT DID WE JUST DO?

- `git init` will transform any folder into a Git repository.
- You can think of it as giving Git super powers to a folder so that Git starts tracking any changes in that folder.
- If the command `git status` returns no errors, it means your folder has successfully been Git-ified!

GOOD REPOSITORY PRACTICES

- Repos are meant to be self-contained project folders. 'Project' can be how you define it - one html page or a whole app.
- Name folders with all lowercase letters and with no spaces - use dashes or underscores instead.
- **WARNING NOTE:** Do not put a repo inside of a repo. Git will get confused and have no idea what changes to track.

TRACKING CHANGES

TRACKING STATES IN GIT

As you make changes in your repo, you can tell Git how to treat those changes.



MODIFY A FILE - DO IT WITH ME!

1. Create a new file in your new folder named `about-me.txt`

```
$ type NUL >> about-me.txt
```

2. Check the status of your repo with `git status`

```
$ git status
```

MODIFIED/UNTRACKED

```
D:\Documents\Repositories\slide-decks>git status
On branch master
Your branch is up to date with 'origin/master'.

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        modified:   css/theme/heather.css
        modified:   decks/web-application-development/index.html

Untracked files:
  (use "git add <file>..." to include in what will be committed)

        decks/WAB-javascript/
        decks/web-application-development/img/80x15.png
        decks/web-application-development/img/Liz-Lemon-giving-herself-high-five.gif
        decks/web-application-development/img/branches.jpg
        decks/web-application-development/img/branching-octocats.png
        decks/web-application-development/img/branching.png
        decks/web-application-development/img/three_stages_commit.png

no changes added to commit (use "git add" and/or "git commit -a")
```

When you make changes to a file or add a new file but haven't added or committed yet

ADD A FILE TO STAGING - DO IT WITH ME!

1. Tell Git to track our new file with **git add**

```
$ git add about-me.txt
```

2. Check the status of your repo with **git status**

```
$ git status
```

OTHER COMMANDS

- **git add .** - add all files under current directory
- **git add --all** - add all files from all directories/current project

STAGED

```
D:\Documents\test-sait>git status
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)

    new file:   about-me.txt
```

Here's an example of what you see when you use `git add` to let Git know that these are the files you want to 'stage' or prepare for committing.

COMMITTING CHANGES - DO IT WITH ME!

1. Check the status of your repo with `git status`. Make sure that the changes listed represent exactly what you want to commit.

```
$ git status
```

2. Commit the change with a message that explains and describes what changes you made.

```
$ git commit -m "First commit. Added about-me.txt to repository."
```

COMMITTED!

```
D:\Documents\test-sait>git commit -m "First commit. Added about-me.text to repository."  
[master (root-commit) e27d8b6] First commit. Added about-me.text to repository.  
1 file changed, 0 insertions(+), 0 deletions(-)  
create mode 100644 about-me.txt
```

Success!

**CONGRATULATIONS.
YOU ARE NOW USING GIT.**



WHAT DID WE JUST DO??

HOW IS THIS DIFFERENT FROM JUST SAVING A FILE?

- When we **add** a new file, we tell Git to add the file to the repository to be tracked.
- This is also called **staging** a file. We can see our changes in the **staging area** (aka the **index**, aka the **cache**), ready to be saved.
- A **commit** saves the changes made to a file, not the file as a whole. The commit will have a unique ID so we can track which changes were committed when and by whom.

IN OTHER WORDS...

...a commit is like a snapshot of your project at a current time



LOOK AT YOUR PROGRESS

```
$ git log
```

```
D:\Documents\test-sait>git log
commit e27d8b6f20f531700794dccf93df5298e605ce4a (HEAD -> master)
Author: Heather Tovey <heather@htovey.com>
Date:   Sat Jan 19 14:07:01 2019 -0700

    First commit. Added about-me.text to repository.
```

Try using:

```
$ git log --oneline
```

for a nicer looking git log.

WHEN TO COMMIT?

- Commit early and often!
- When you have completed a mini 'idea' or 'task':
 - You got a function to work!
 - You corrected a few misspellings
 - You added some images
- **IMPORTANT NOTE:** Commit when your code works! Try not to commit broken code

GOOD COMMIT MESSAGES

Include a descriptive but succinct message of the changes you have made, in the present tense

```
$ git commit -m "Add capitalization function for header text"
```

Main point is: other people need to be able to read your commit history and understand what you were accomplishing at each step of the way

[Article: Art of the commit](#)

QUICK REVIEW

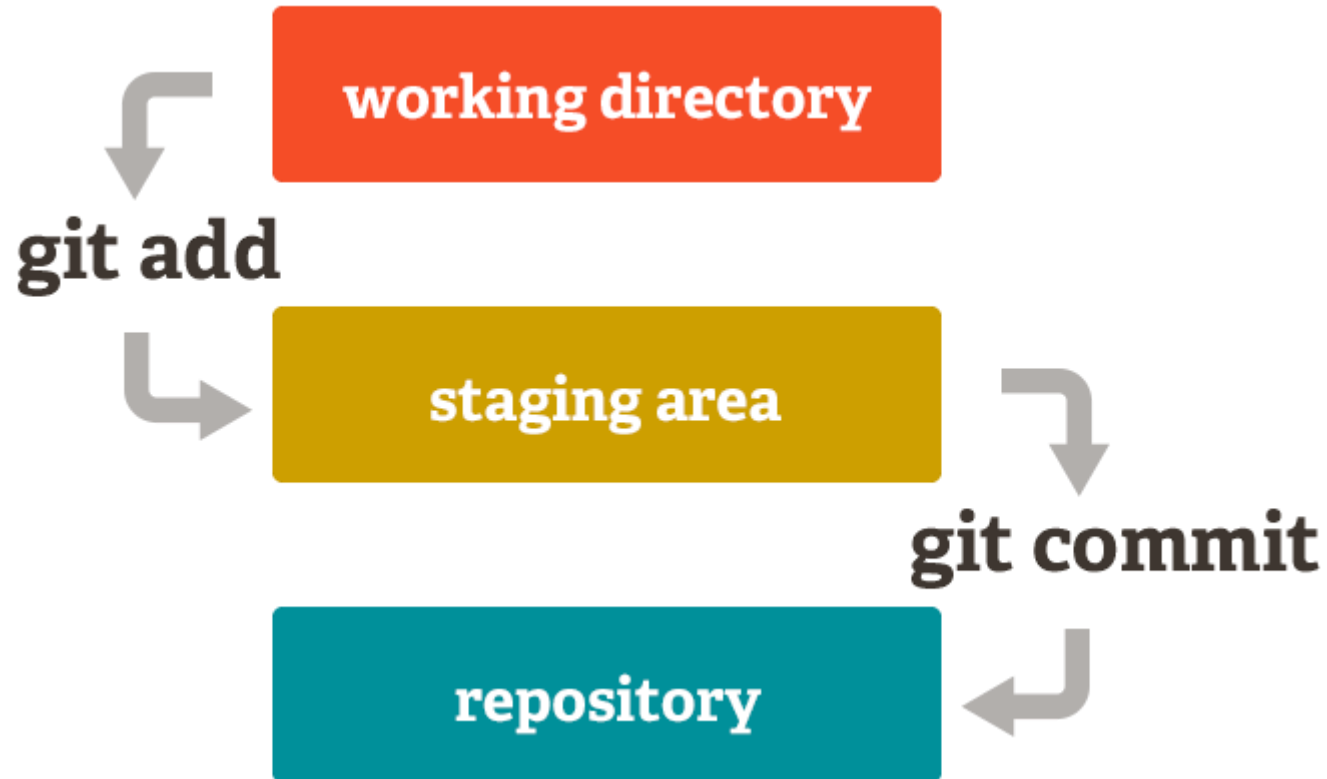
- `git init`: turns a folder into a Git repository
- `git status`: checks the status of your files
- `git add file_name`: adds file to the staging area
- `git commit -m "your commit message"`: commits your changes
- `git log`: see your commits so far

ACTIVITY: PRACTICE COMMITS

Go to **Content > Exercises > Commit Exercise > commit-exercise.pdf** and practice git!

THE MAGICAL REALMS OF GIT

Each of the states of Git corresponds to an area of the Git repo, so here's some vocab:



THE MAGICAL REALMS OF GIT

- **Working directory/tree**: The current version of your project where you are making changes (which is reflected in your code editor)
- **Staging Area**: The place where you stage your files when you are readying them to commit
- **Repository**: When you commit, Git permanently saves only the changes from your staging area to the repo's memory

REVERTING CHANGES

WE ALL MAKE MISTAKES



Don't worry. Git is your friend.

SET UP

1. Move back to your my_project repository
2. Watch me do the next examples first, then try it yourself!

SCENARIO 1: UNDOING MODIFIED/UNTRACKED CHANGES

You made some changes to some files (have not git added or committed yet), and realize you don't want those changes!

Open **about-me.txt** and make some changes or add something new. Then:

```
$ git checkout about-me.txt
```

Look at **about-me.txt** in your editor: your changes are gone (you've gone back to the previous commit state).

SCENARIO 2: UNSTAGING A FILE

You `git add` a modified or new file, but realized you don't want it your next commit!

In your text editor, create a new file, and name it `possum.txt`. Then:

```
$ git add possum.txt
$ git status
$ git reset possum.txt
$ git status
```

The file is removed from staging, but your working copy will be unchanged.

SCENARIO 3A: UNCOMMITTING, BUT WANT TO KEEP ALL YOUR CHANGES

You made a commit, but then realize that a piece of code doesn't work, so you just want to uncommit!

Open **about-me.txt** and make some changes. Then:

```
$ git add about-me.txt
$ git status
$ git commit -m "Make changes to about-me.txt file"
$ git reset --soft HEAD~1
$ git status
```

EXPLANATION

Your most recent commit is called the **HEAD**.

Passing `git reset` the options of `--soft HEAD~1` essentially asks to move the HEAD back by one commit (essentially uncommitting your most recent commit).

`--soft` means you won't lose your changes—they'll just move to staging.

SCENARIO 3B: UNCOMMITTING, BUT YOU DON'T WANT TO KEEP ANY CHANGES, PERIOD



You realize you don't want any of the code in your previous commit, so just getting rid of that commit completely

You still have the change in the staging area for about-me.txt

```
$ git add about-me.txt
$ git status
$ git commit -m "Make changes to about-me.txt file"
$ git reset --hard HEAD~1
$ git status
```

EXPLANATION

passing `git reset` the options of `--hard HEAD~1` will delete the last specified commit **and** all the work related to it.

Heads up—there are many, many different ways to undo changes. That's what's powerful about Git. Learn more at [Atlassian tutorial](#)

BRANCHING

BRANCHING



A branch is essentially another copy of your repo that will allow you to isolate changes and leave the original copy untouched. You can later choose to combine these changes in whole or part with the "master" copy, or not.

Branches are good for features!

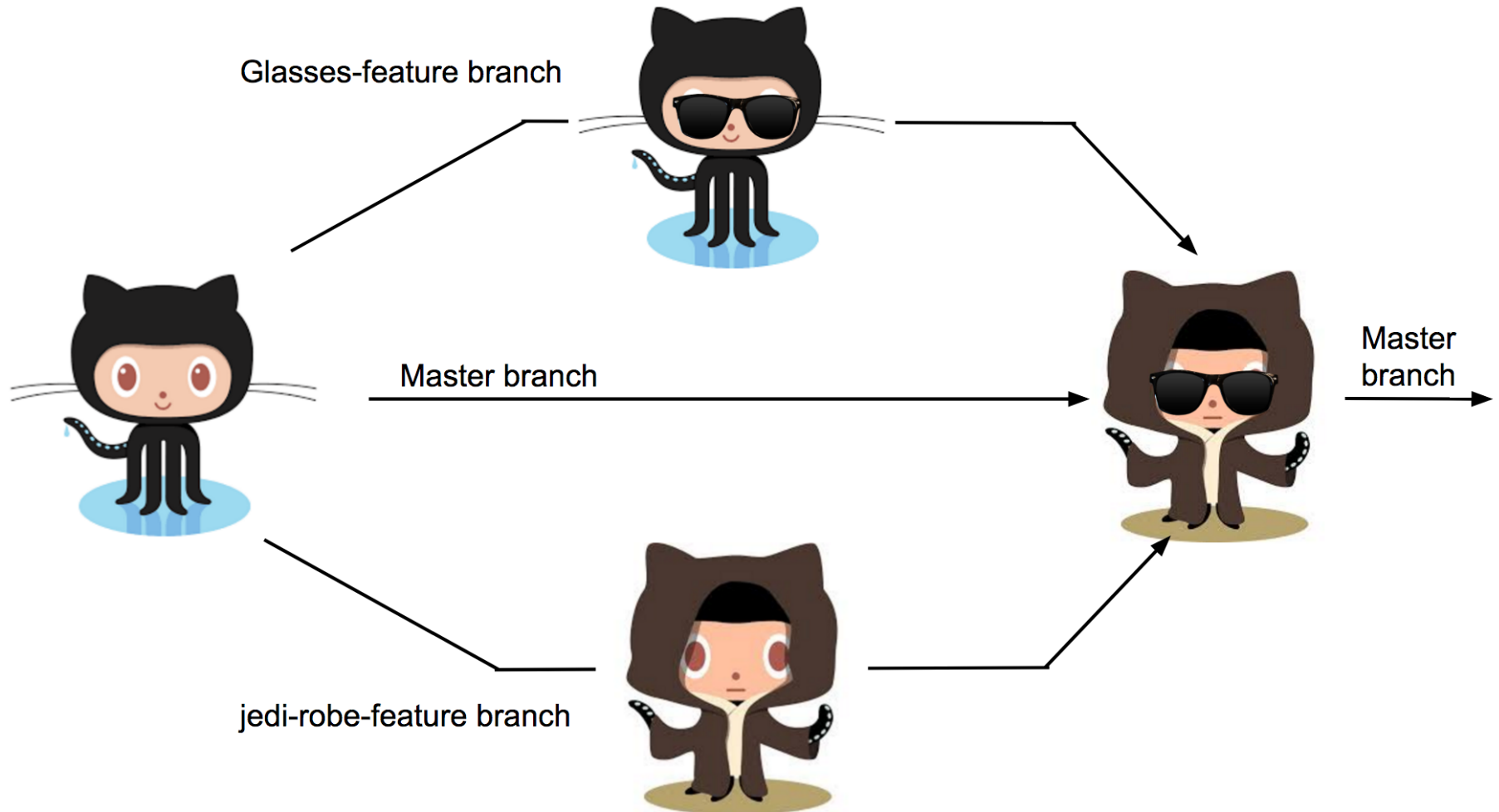
WHAT'S A FEATURE?

Can be large or small additions.

Examples:

- **Big** - Adding a new section to a site or app
- **Small** - Adding a carousel to the homepage

BRANCHING



Credit to Jen Gilbert & DBC for the idea of this slide

WHY DO WE NEED BRANCHING?

- Develop different code on the **same base**
- Conduct **experimental work** without affecting the work on master branch
- Incorporate changes to your master branch **only if and when you are ready**...or discard them easily

Branches are cheap!

BRANCHING CYCLE

So, you want to develop a new feature:

1. Make sure you are on master
2. Create a new branch and jump over to it
3. Develop your code. Commit commit commit!
4. When the feature is done, merge it into master
5. Delete your feature branch!

BRANCHING - DO IT WITH ME!

Create a new branch called feature

```
$ git branch  
// you should see only * master  
$ git checkout -b feature  
$ git branch  
// you should see * feature and master
```

OKAY, LET'S BREAK THAT DOWN

- `git branch`: tells you what branches you have, and `*` indicates which branch you are currently on
- `git checkout -b branch-name`: the `-b` creates a new branch, and `checkout` will hop you over to that branch

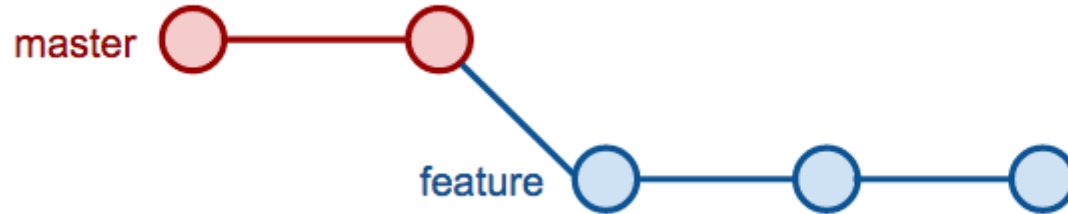
COMMITTING ON A NEW BRANCH - DO IT WITH ME!

Add new lines to **about-me.txt**

```
$ git add about-me.txt  
$ git commit -m "Adding changes to feature"  
$ git log --oneline
```

BRANCHING

What we just did, in picture form:



SWITCHING BRANCHES - DO IT WITH ME!

```
$ git branch
```

Switch to master branch and look at the commit history

```
$ git checkout master  
$ git log --oneline
```

Switch to feature branch and look at the commit history

```
$ git checkout feature  
$ git log --oneline
```


MERGING - DO IT WITH ME!

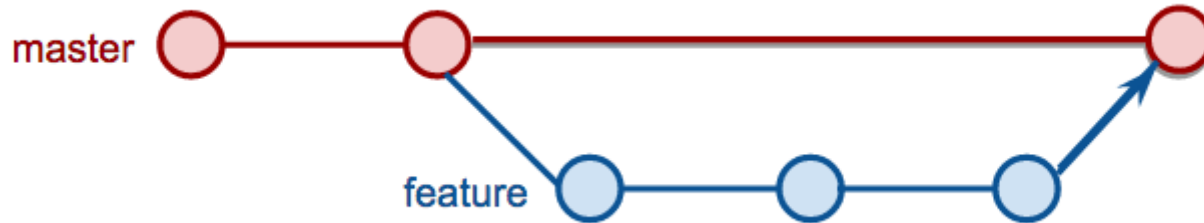
MERGE TO GET CHANGES FROM ONE BRANCH INTO ANOTHER

Switch to master and merge changes

```
$ git checkout master  
$ git merge feature  
$ git log --oneline
```

MERGING BRANCHES

When you merge, you create a new commit on the branch you just merged into



DELETE FEATURE BRANCH - DO IT WITH ME!

Since your code from your feature branch is merged into master, you don't need the branch anymore!

```
$ git branch -d feature
```

Hint: Make sure not to be on the branch you are deleting.

QUICK REVIEW

- `git checkout -b branch_name`: creates a new branch and hops over to it
- `git checkout branch_name`: switch to another branch
- `git branch`: lists all your branches
- `git merge branch_name`: merges branch into the current branch
- `git branch -d branch_name`: deletes branch

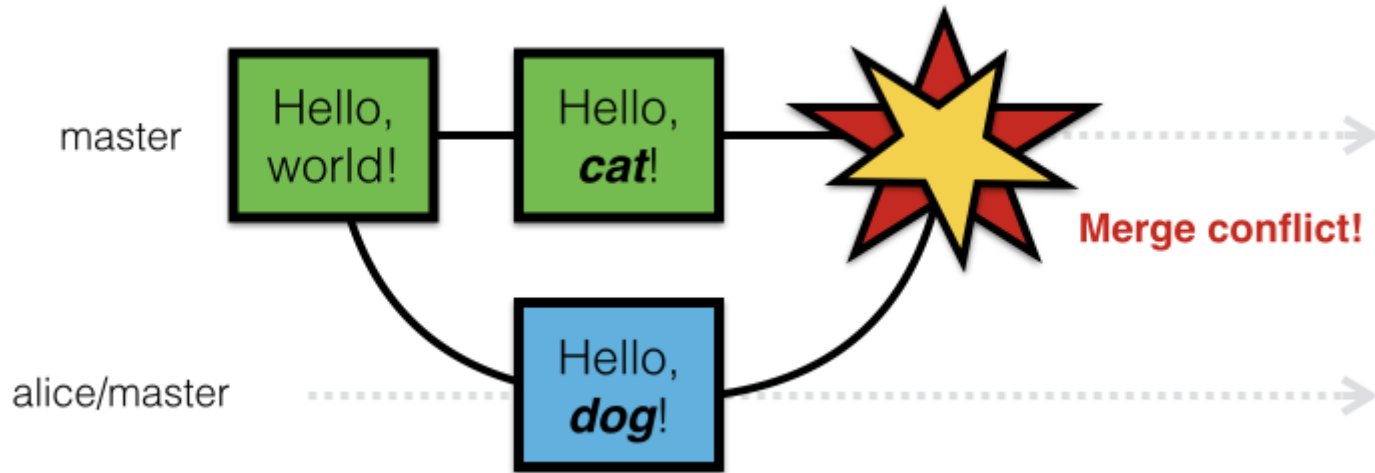
ACTIVITY: PRACTICE BRANCHING AND MERGING

Go to **Content > Exercises > Branching and Merging Exercises > [branching-merging-exercise.pdf](#)** and practice git!

WHAT COULD POSSIBLY GO WRONG?



WHAT IS A MERGE CONFLICT?

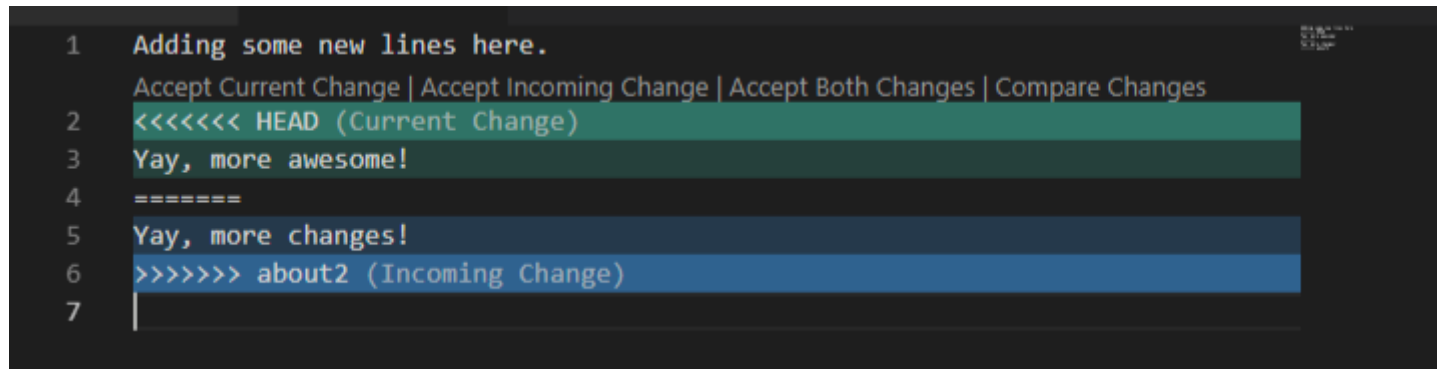


WHAT A MERGE CONFLICT LOOKS LIKE

You may see something like this in your command line:

```
D:\Documents\test-sait>git merge about2
Auto-merging about-me.txt
CONFLICT (content): Merge conflict in about-me.txt
Automatic merge failed; fix conflicts and then commit the result.
```

You will see this in the affected file inside VS Code.



```
1 Adding some new lines here.
   Accept Current Change | Accept Incoming Change | Accept Both Changes | Compare Changes
2 <<<<<<< HEAD (Current Change)
3 Yay, more awesome!
4 =====
5 Yay, more changes!
6 >>>>>>> about2 (Incoming Change)
7 |
```


MERGE CONFLICTS - DO IT WITH ME!

Go back to your my_project on your desktop

Change the first line in **about-me.txt** in **master** branch

```
$ git add about-me.txt  
$ git commit -m "Changing about-me in master"
```

Now change the first line in **about-me.txt** in **feature** branch

```
$ git checkout feature  
# open about-me.txt and change the first line  
$ git add about-me.txt  
$ git commit -m "Changing about-me in feature"
```

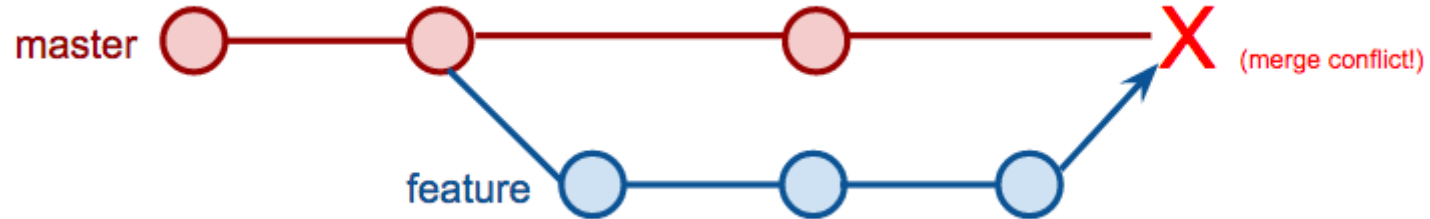
MERGE CONFLICTS, CONT.

Merge the changes from **master** into the **feature** branch

```
$ git merge master #remember, you are on the feature branch here
```

You will be notified of a conflict. Go to the file in your editor and fix the problem. Then add and commit your edits.

MERGING



The merge conflict occurred because the feature branch (which is based off of master) both had divergent histories for the same file.

VOCABULARY REVIEW

- A **repository** is where you keep all the files you want to track.
- A **branch** is the name for a separate line of development, with its own history.
- A **commit** is an object that holds information about a particular change.
- **HEAD** refers to the most recent commit on the current branch.

COMMAND REVIEW

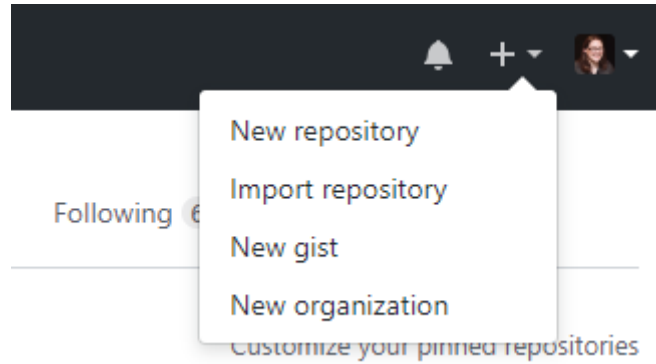
- `init`
- `status`
- `add`
- `commit`
- `log`
- `branch`
- `checkout`
- `merge`

GITHUB

GITHUB

- Launched in **2008**
- Leader in **Social Coding**
- GitHub is a commercial site that allows users to **host Git repositories** publicly and privately
- **Open source projects** host or mirror their repositories on GitHub
- **Post your own code** for others to use or contribute to
- Use and **learn** from the code in other people's repositories


CREATE YOUR FIRST REPOSITORY



CREATE YOUR FIRST REPOSITORY



Owner

Repository name *

 hrtovey ▾ / HelloWorld ✓

Great repository names are short and memorable. Need inspiration? How about [laughing-meme](#).

Description (optional)

- ☒  **Public**
Anyone can see this repository. You choose who can commit.
- ☐  **Private**
You choose who can see and commit to this repository.

☐ **Initialize this repository with a README**

This will let you immediately clone the repository to your computer. Skip this step if you're importing an existing repository.

Add .gitignore: None ▾

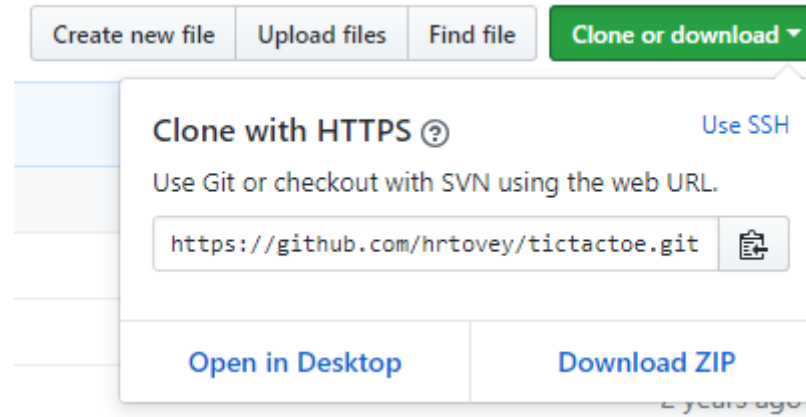
Add a license: None ▾



Create repository

SET UP REMOTE CONNECTION: GITHUB LINK

Copy HTTPS link from Github repo



SET UP A REMOTE CONNECTION

```
# From inside your local git repo
git remote add origin [pasted link from Github]

# Verify the new remote URL
git remote -v

# Push your local work to the repo
git push origin master
```

README.MD

- A README isn't required, but it's good to have one
- Describe your project and add documentation (for example, how to install or use your project)
- Include contact information or how people can help you with the project

ACTIVITY: ADD A README.MD

1. Create a new file in your local project (the project on your Desktop).
2. Save it as README.md.
3. Add some information like your name, the project name, and a short description of the project.
4. Add and commit your README.md. Then push!

```
git push origin master
```

5. Go look at your Github repo online to see your changes.

For more information about creating a good README, check out <https://www.makeareadme.com/>

PULLING FROM REMOTE REPOSITORY

If you are working with a team, you want to make sure that you have everyone's changes before pushing your changes to the GitHub repo.

```
# Pull current state from Github
git pull origin master

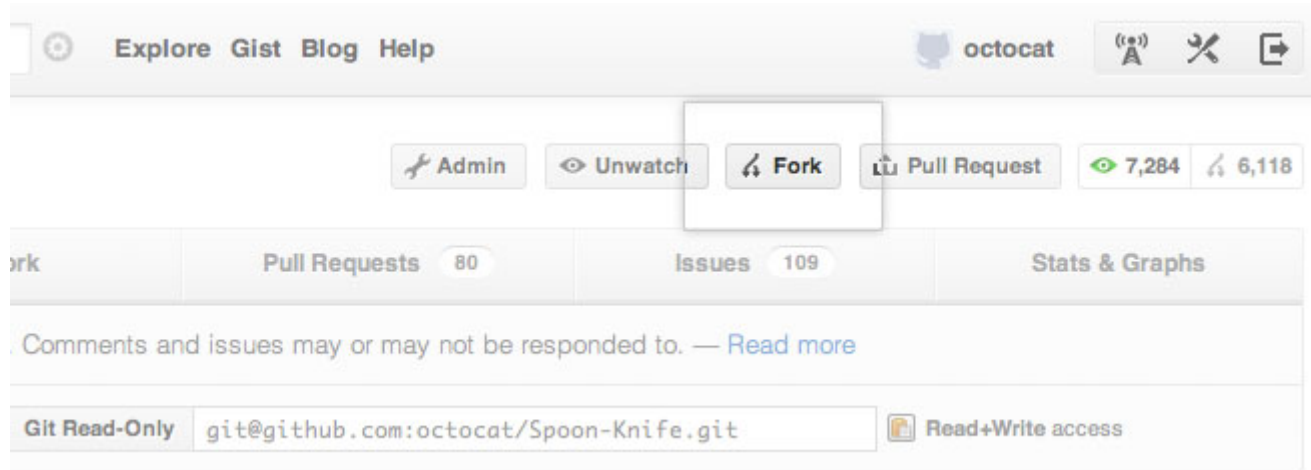
# Fix any merge conflicts and commit. Then, push local changes to GitHub
git push origin master
```

FORKING

- There are **MILLIONS** of public repositories on GitHub
- If you want to **use** or **contribute** to a repository, you can fork it.
- This will create a copy of it in your Github account

FORKING

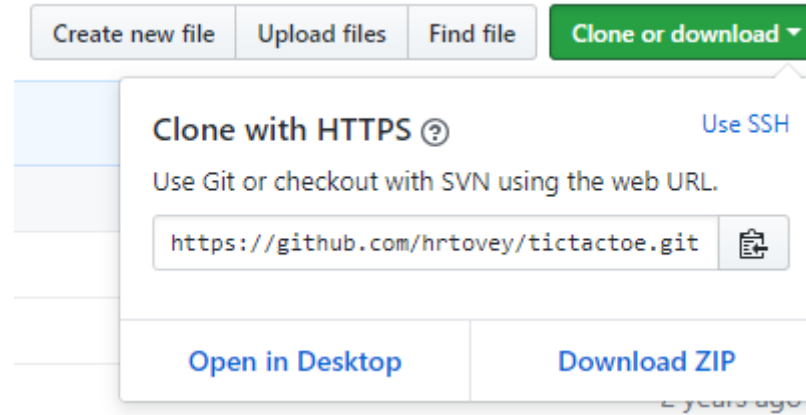
Go to <https://github.com/gdisseattle/GitResources> and hit the fork button.



FORKING

CLONING

Go to **your** GitResources repo on Github, find this link and copy it.



FORKING

CLONING

Navigate to the location you want new directory to be in- NOT INSIDE ANOTHER GIT REPO! Then, clone to get a local repository of your fork.

```
git clone https://github.com/username/FORKED-REPO-NAME.git  
cd FORKED-REPO-NAME
```

CLONING

The `git clone` command does a lot!

- Creates a local directory for the repo
- Initializes that directory as a Git repo
- Adds a remote connection called `origin` to the Github repo it was cloned from
- Pulls from the Github repo so it's perfectly up-to-date

CLONING

Add a remote connection to the **original** repository, so you can stay up to date with their changes:

```
# Creates a remote connection, called "upstream",  
#   to the original repo on Github  
git remote add upstream https://github.com/original-username/FORKED-REPO-NAME.git  
  
# Pulls in changes not present in your local repository,  
#   without modifying your files  
git fetch upstream
```

PULL REQUESTS

- After you fork and clone a repository, all pushed changes will go to **your** fork
- These changes will not affect the original repository
- If you would like to get your changes to be incorporated into the original repo, you can submit a pull request

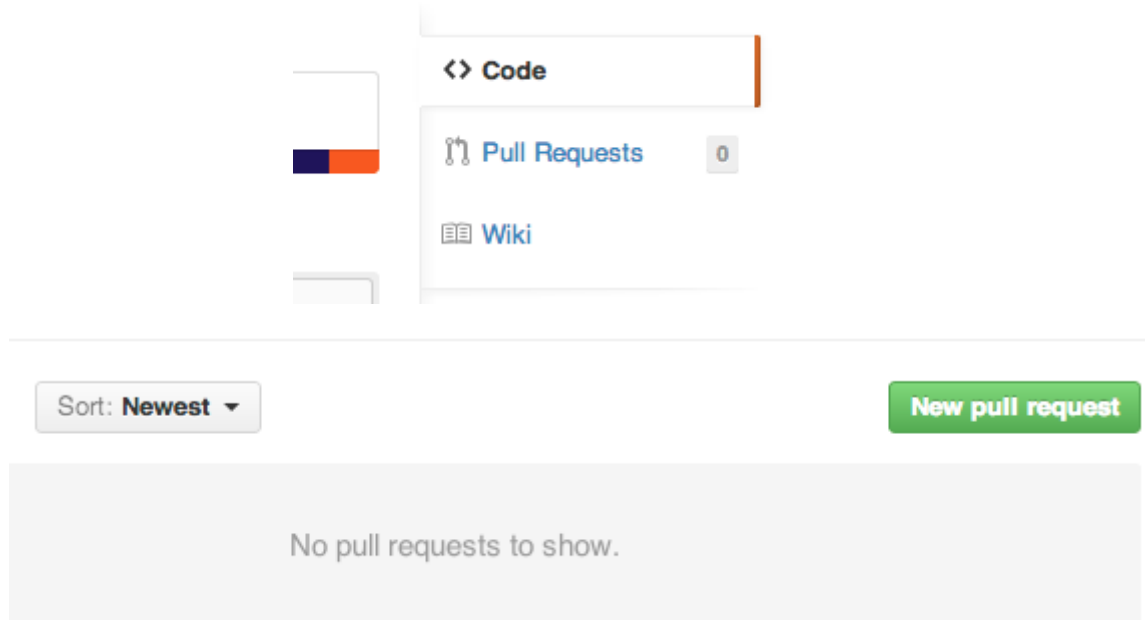
PULL REQUESTS

- Go to your GitResources directory
- Add a resource to the list
- Save, add, and commit your changes
- Push your changes to your Github repo

```
git add .  
git commit -m "Add great new resource"  
git push origin master
```

STARTING A PULL REQUEST

Visit **your** Github repo page



PREVIEWING AND SENDING PULL REQUEST

□ How to preview and send a pull request. Image from <https://help.github.com/articles/using-pull-requests>

Fill out the form with a polite, helpful description, and submit.

MANAGING PULL REQUESTS

How to manage pull requests is out of the scope of this short workshop, but you can learn more from the [Github Collaborating Tutorials](#)