UNIX COMMANDS

YOUR FIRST TERMINAL SESSION

Locate and open your terminal:

Linux Users

- Use the keyboard shortcut Ctrl-Alt-T

Mac Users

- Launch Finder
- Go to Applications > Utilities
- Click on Terminal
- Right-click on dock icon, click Options > Keep In Dock

Windows Users

- You'll use git-bash instead of the windows command prompt
- Right-click on desktop
- Select Git Bash
- If not already set, right-click taskbar icon and select 'pin this program to the taskbar'

READING THE PROMPT

The first stuff you see in your terminal is called the prompt.

It will include your username, where you are, and what machine this terminal is on.

By default:

- Mac:

```
machine_name:current_directory username$
```

- Linux:

```
username@machine_name:current_path$
```

- Window (git-bash):

```
username@machine_name current_path$
```

COMMAND: `pwd`

The pwd command shows the *present working directory*.

Mac

```
$ pwd
/Users/yourName
```

Linux

```
$ pwd
/home/yourName
```

Windows (git-bash)

```
$ pwd
/c/Users/Your Name
```

CONCEPT: THE PATH

In any computer system, a path represents a location in the filesystem.

Paths are like addresses, listing a location from the general to the specific.

A bit like addressing an envelope backwards:

USA
Seattle, WA 98105
123 Somestreet
Some Person

vs.

/home/cewing/projects/someproject

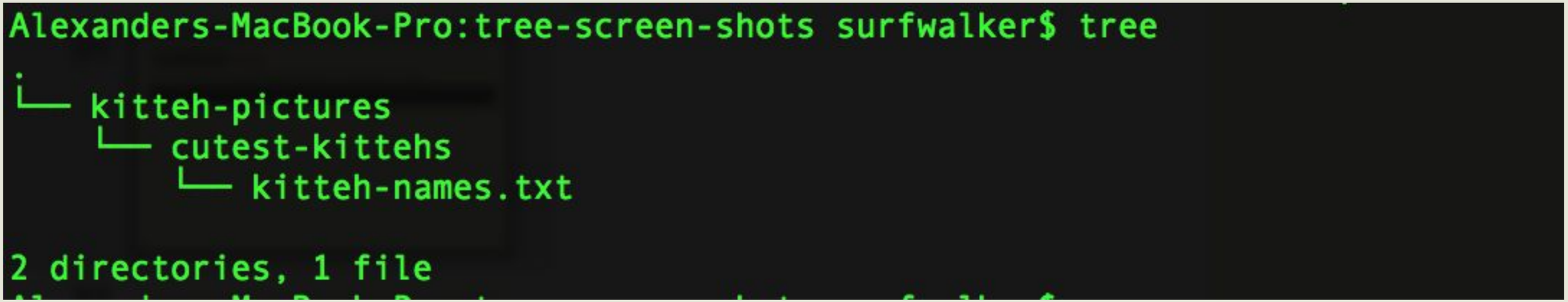
A path is absolute when it starts with /

A path is relative when it does not

COMMAND: `tree`

The `tree` command provides a visual representation of your current directory's structure.

```
$ tree
```



COMMAND: `ls`

The `ls` command shows you a listing of the contents of your present working directory.

```
$ ls
```

The behavior of this command can be altered by flags such as the `-la` which is a combination of the `-l` (long) and `-a` (all) flags.

`-l` provides more information about each directory and file.

`-a` reveals hidden files and folders .

```
$ ls -la
```

COMMAND: `cd`

The cd command allows you to *change directories*.

`cd` entered by itself it will take you to your *home directory*.

```
$ cd
```

Or it can take a path as an argument. The path can be either *absolute* or *relative.*

An absolute path always begins with a “`/`” which represents the root directory of your computer.

```
$ cd /Users/YourName/somewhere-else
```

An example of an relative path would be:

```
$ cd somewhere-else
```

COMMAND: `cd ..`

To move up a level from your present working directory simply enter `cd ..` where `..` is an alias for the parent directory.

```
$ cd ..
```

You can even chain them together, providing relative paths that go up more than one level:

```
$ cd ../..
```

And you can combine these with directory names to go back down into a different branch of your filesystem:

```
$ cd ../somewhere-else
```

COMMAND: `mkdir`

The terminal equivalent of new folder is `mkdir` which stands for *make directory*.

This command take an argument which is the name of the directory you want to create.

```
$ mkdir kitteh_pictures
```

To create nested directories (a new directory within a new directory) you can use the `-p` flag.

```
$ mkdir -p kitteh_pictures/cutest_kittehs
```

COMMAND: `touch`

The terminal equivalent of new file is `touch`.

This command take an argument which is the name of the directory you want to create.

```
$ touch kitteh_names.txt
```

COMMAND: `atom`

The `subl` command opens your Atom text editor. This is not a command that is native to the terminal.

With no argument it simply opens the program:

```
$ atom
```

You can provide a file as an argument:

```
$ atom kitteh_names.txt
```

You can provide a directory as an argument:

```
$ atom kitteh_pictures/
```

COMMAND: `mv`

The mv command allows moving files from place to place in a file system.

It expects two paths as arguments which are the file you want to move and the directory where you want to move it.

```
$ mv kitteh_names.txt kitteh_ideas/
```

Another use for the `mv` command is *renaming*. If an explicit filename is provided at the end of the second argument the targeted file will be moved and renamed. You can use this to change the name of a file and not move it.

```
$ mv kitteh_names.txt terrible_kitteh_names.txt
```

Trouble Spot

Depending on how you've named things (or due to the default names inherent in your operating system)your home directory file path may contain spaces which when entered as a command will result in an error:

```
$ cd /c/Users/Your Name
sh.exe": cd: /c/Users/Your: no such file or directory
```

The problem is the space between my first and last names

The command line expects paths to be a single continuous string of characters

Spaces are used to delimit one element of the command line from the next

You can fix this by escaping it with the `\` character:

```
$ cd /c/Users/Your\ Name
```

CONCEPT: Naming Conventions

Avoid spaces in the names you give to files and directories.

Use dashes and underscores to create visual separation between words in names.

Prefer lower-case letters in naming files and directories.

This is good:

- my_project_file.html

This is bad:

- My Project File.html

COMMAND: `rm`

The `rm` command is the equivalent of moving something to the trash with one important distinction: the file is completely deleted from the system and is no longer recoverable. Always take a moment to be sure before you execute an `rm` command.

```
$ rm terrible_kitteh_names.txt
```

BEWARE!!!
There `-rf` flag is commonly used in conjunction with the `rm` command. This allows you to delete directories and all files contained therein. The `f` stands for *force* which means your system will not provide any warning or request for confirmation. It is possible to delete your entire everything this way. Be extremely cautious with this command. Sam refers to this command as, “Remove with Fire”.

COMMAND: `cp`

The `cp` command allows you to copy a file and place that copy in a different location.

It takes two arguments with the first being the file to be copied and the second argument being the destination.

```
$ cp kitteh_names.txt ~/Desktop
```

This will make a new copy of the `kitteh_names.txt` file on your Desktop.

COMMAND: `history`

The history command allows you to review and revise the history of actions you've taken in a shell

```
$ history
```

COMMAND: `man`

The `man` command provides access to the built-in manual for all unix commands

Providing the command with the name of some other command will print detailed information about how that command may be used

Often these manual pages include useful examples for common and advanced usage patterns

```
$ man ls
```

For Windows users...Google is your friend!

COMMAND: `date/cal`

The `date` command provides the current date and time to the second.

```
Alexanders-MacBook-Pro:~ surfwalker$ date
Fri Sep 18 14:37:16 PDT 2015
Alexanders-MacBook-Pro:~ surfwalker$
```

The `cal` command provides a visual representation of the current month. You can also provide month and year arguments to see any future or previous months.

```
Alexanders-MacBook-Pro:~ surfwalker$ cal 7 1776
     July 1776
Su Mo Tu We Th Fr Sa
           1  2  3  4
 5  6  7  8  9 10 11
12 13 14 15 16 17 18
19 20 21 22 23 24 25
26 27 28 29 30 31
```

HELPFUL TIPS

Up and *Down* arrows on your keypad allow you to scroll through previously entered commands to save you from retyping them.

The *tab* key will auto complete the file, directory or command that you a currently typing.

Just as `..` is an alias for the parent directory `.` is an alias for the current directory. This is useful specifically for opening files with Sublime when you want to open all the files in a directory e.g. `subl .`

You can use the .. symbol as an element in a path (absolute or relative) as a shortcut for "one level up"

You can use the . symbol as an element in a path as a shortcut for "right here"

You can use the ~ (tilde) as a shortcut for the absolute path of your home directory

You can use the cd command without an argument to return to your home directory immediately from anywhere

ANY
QUESTIONS?

REVIEW

We've added the following unix commands to our repertoire:

COMMAND	
history	interact with your command line history
less	read large text inputs in a controlled fashion
mv	move files from one place to another, or rename them, or both
touch	create a new file, or update the modified date for an existing

REVIEW

COMMAND	
cp	copy the contents of a file or directory to a new location
rm	remove a file from the filesystem entirely
rmdir	remove a directory from the filesystem if it is empty

GIT & GITHUB

What is Git?

A "Version Control System”.

Git is used to save and backup work, share your code and collaborate on projects.

Does this Look Familiar?

Any term/school paper you've ever worked on:

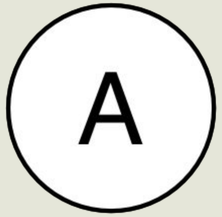
- `term_paper.docx`
- `term_paper2.docx`
- `term_paper2_with_footnotes.docx`
- `final_term_paper.docx`
- `final_term_paper_draft2.docx`
- `term_paper_for_submission.docx`
- `term_paper_for_submission_for_real_this_time.docx`

A Version Control System such as Git alleviates the above nightmare.

Snapshots in Time

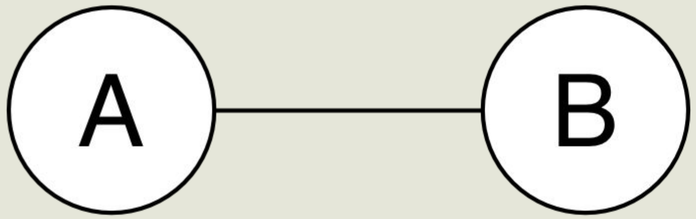
Git uses “commits” to represent each successive version of a file or files.

Commits are the Git equivalent of “Save As…”



Snapshots in Time

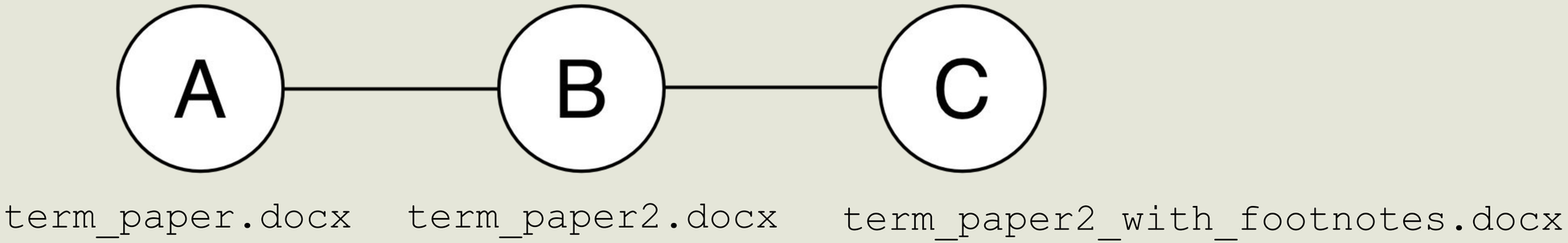
Each successive version creates a new snapshot on the timeline of the project.



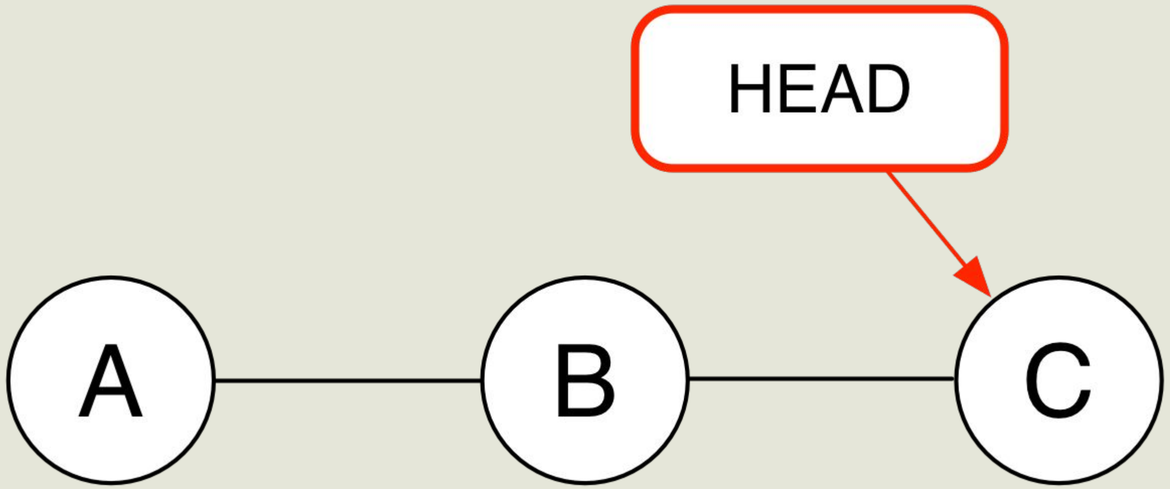
Snapshots in Time

As we continue to update and revise the files in our project Git helps us keep track of where we are and where we've been.

Think of each snapshot, or "commit" as each new version of the paper you were saving. However, instead of making a full copy of every file, it only keeps track of the actual differences between each version, making it very efficient and fast.



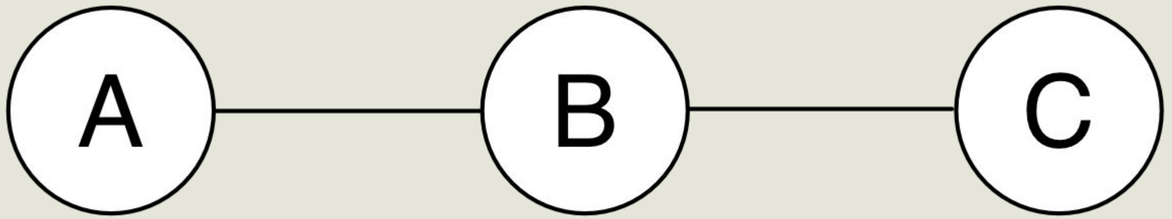
Keeping Track



Each snapshot, or “commit,” can also have a label that points to it.

One of these is HEAD, which always points to the place in the timeline that you are currently looking at. You can think of HEAD as being “You Are Here.”

A Summary of Git



A Git repository is a set of points in time, with history showing where you've been.

Each point has a name (here A, B, C) that uniquely identifies it, called a hash.

Each commit also has a user-generated message which describes its purpose.

The path from one point to the previous is represented by the difference between the two points.

Enter GitHub

Remotes serve as a way of sharing work with other developers.

GitHub has emerged as a premier location for such sharing.

It provides you with a common location that anyone can access.

In addition, it provides a number of useful tools for managing work that is being shared among a dispersed group of people.

Git is designed to be a ‘distributed’ system, where you can share code between any connected computers.

GitHub gives you a centralized ‘canonical’ repo that a team can access for the latest contributions from across the team.

SO WHAT?

HOMEWORK
ASSIGNMENTS

GROUP
PROJECTS

ANY
QUESTIONS?

WORKING WITH REPOSITORIES

Wrapping Up

Git and Github together allow for:

- Back up
- Sharing
- Collaboration

Working with Remotes

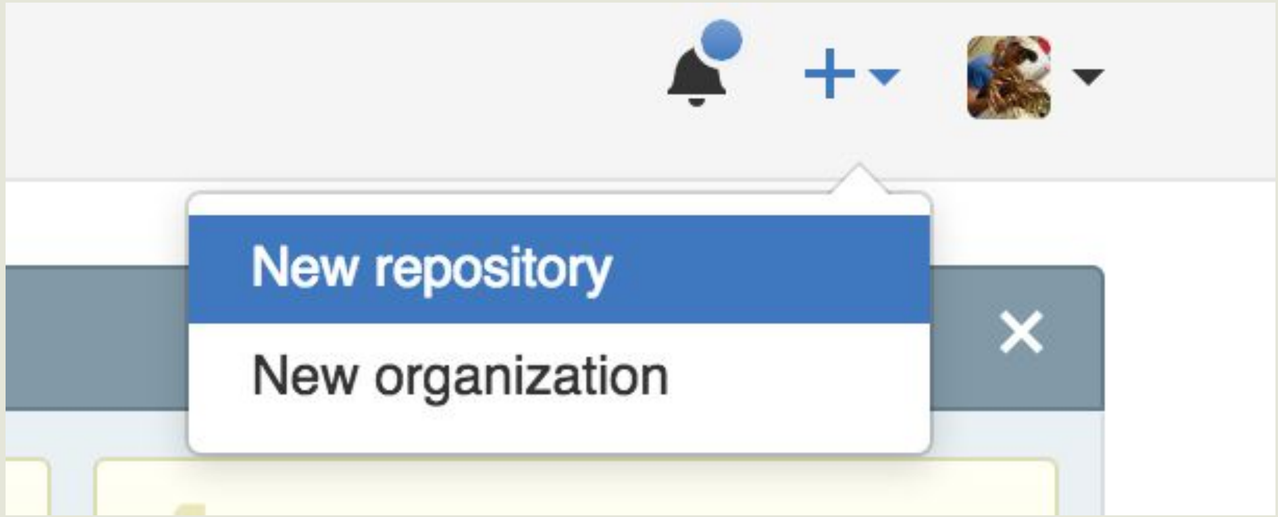
Since Git is a distributed versioning system, there is no central repository that serves as the one to rule them all.

Instead, you work with local repositories, and remotes that they are connected to.

A remote is a repository located on external servers in the cloud.

Remotes are important because they enable your work to be saved in the cloud so that if disaster strikes your local computer your project is not lost.

Creating a Repository in GitHub

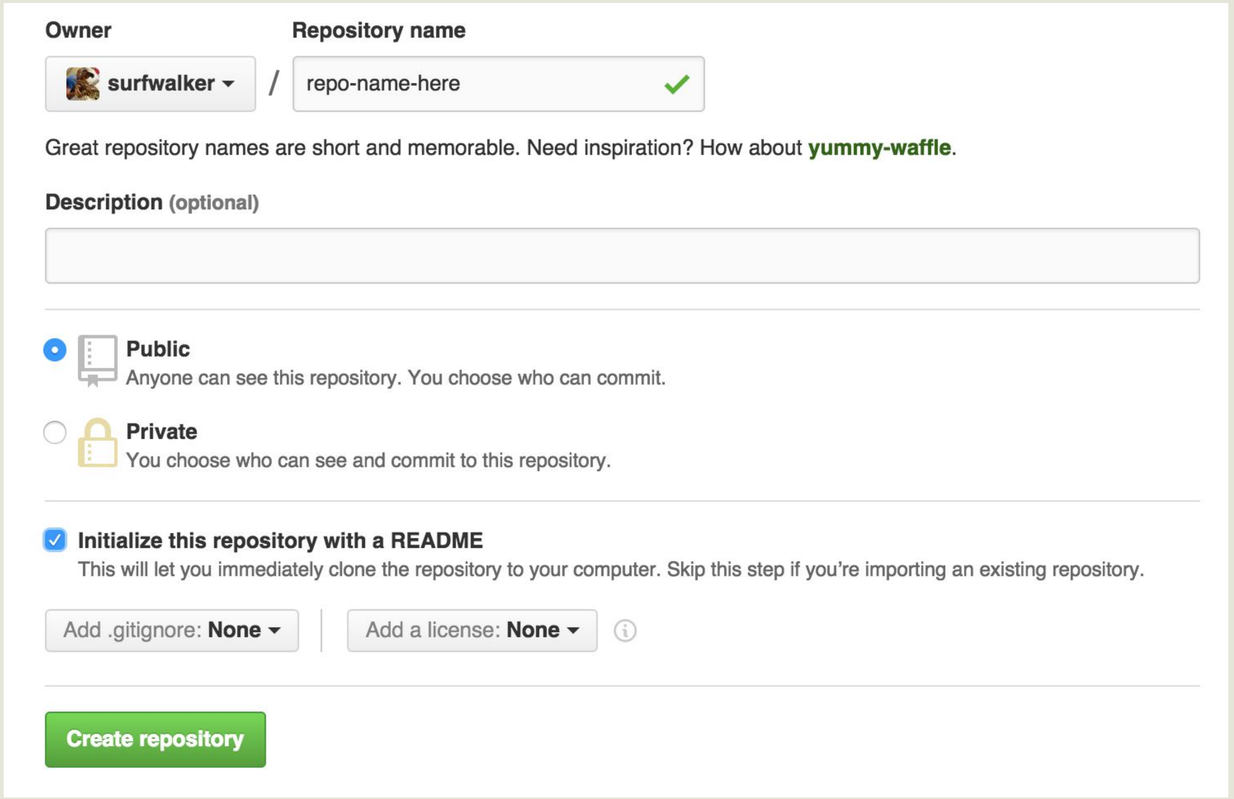


At the top right side of the window, look for your name and avatar.

Next to it you'll find a small + sign, click that.

From the menu that opens, select *New repository.*

Creating a Repository in GitHub



Set up your repository fill in *Repository name*, check the *Initialize this repository with a README* option and click *Create repository*.

Creating a Repository in GitHub

On the bottom right hand side of your screen make sure the option says *SSH clone URL* then copy the URL using either the *Copy to clipboard* button or your keyboard shortcut.

Navigate to your terminal.

GIT COMMAND: `clone`

The `clone` command creates a local copy of a remote repository (or repo).

It takes a URL as an argument. This URL is the pointer to the remote repo.

```
$ git clone git@github.com:surfwalker/repo-name-here.git
```

This will create a directory with the repo's name containing all of the repo's directories and files.

You can now copy or move any files you want to be a part of this project into the cloned directory just created.

Push To Your Remote

Now that you have files in your local repo you then need to commit (take a snapshot) those changes before “pushing” (uploading) them to your remote repo in the cloud (Github).

After navigating into your local repo directory, run the following commands in sequence:

```
$ git status
```

Git will show you what files on your computer have changed since you last made a commit.

```
$ git add <file_name_with_extension>
```

This tells Git that you’re going to want to take a snapshot of this file soon. It’s like telling your files so say, “CHEESE” for the upcoming photo.

```
$ git commit -m ‘Your message goes here’
```

`Commit` tells Git to take the snapshot. The `-m` bit tells it that you want to save a message with that snapshot (think of it like a caption).

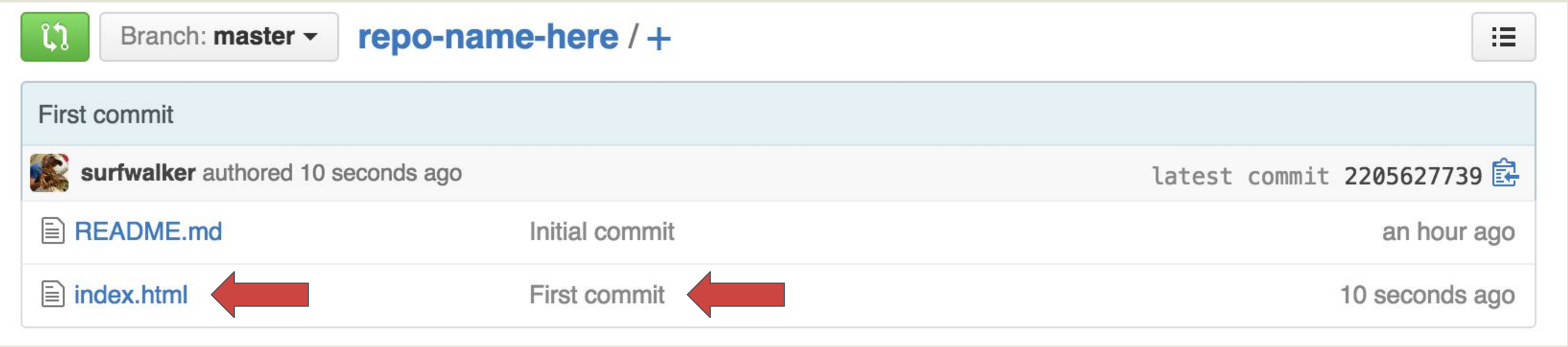
GIT COMMAND: `push`

You can now push the current version of all of your project's files to GitHub for safe keeping.

```
$ git push origin master
Counting objects: 6, done.
…
To git@github.com:surfwalker/repo-name-here.git
* [new branch]      master -> master
```

The `push` command sends your code to GitHub, and will make your GitHub repo have the same files, with the same changes, as the commit you just made.

Verify it on GitHub



In your browser on GitHub you will see the file(s) that you pushed as well as the commit message.

Git Commands

New terminology learned in this lesson:

git clone
Copies a remote repository and all its files to your local machine in a new directory.

git push
Pushes all changes from your local repository to the named remote.

ANY
QUESTIONS?

DEPLOYING YOUR CODE

Showing Off

Another reason why GitHub is so awesome is that it provides a very useful method to *deploy* and share your work.

In order to get your web page up, running and on display for an appreciative audience you need follow a few simple steps.

We will be taking advantage of a feature of GitHub called GitHub Pages.

Showing Off

In your terminal, in your project directory run the following command:

```
$ git checkout -b gh-pages
```

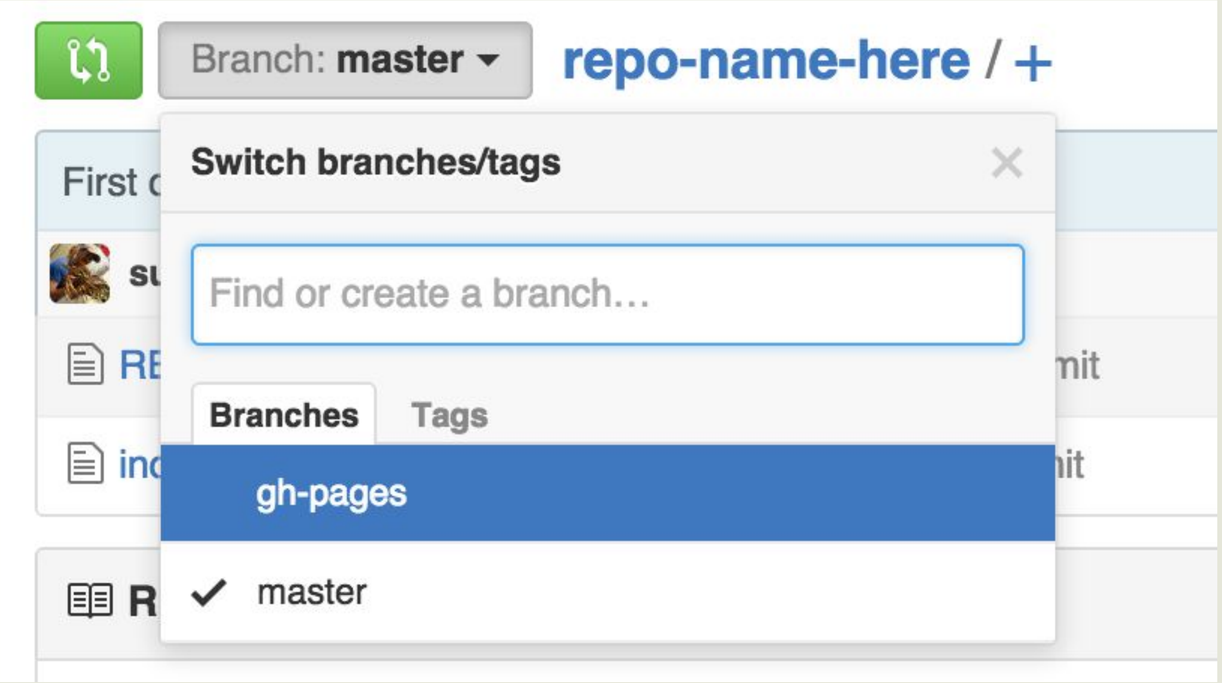
The `checkout` command essentially creates an alternate snapshot timeline for your project.

Pushing Up

In the same way that we pushed our master branch, we now need to push up this alternate branch.

```
$ git push origin gh-pages
```

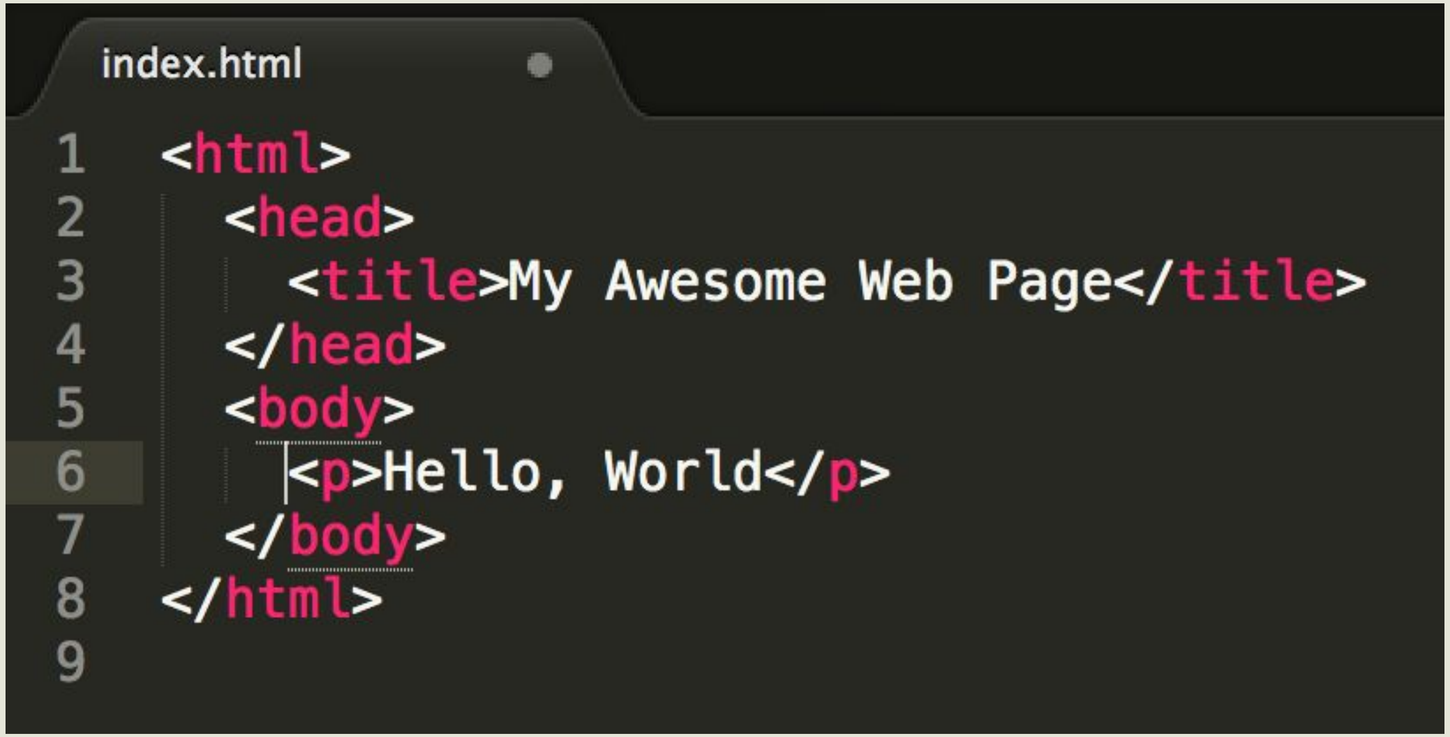
Just like last time this pushes up our work to GitHub to a branch of the same name i.e. `gh-pages`.



Showing Off

Anytime there is a branch called `gh-pages` GitHub automatically recognizes it as containing code to be rendered into a web page.

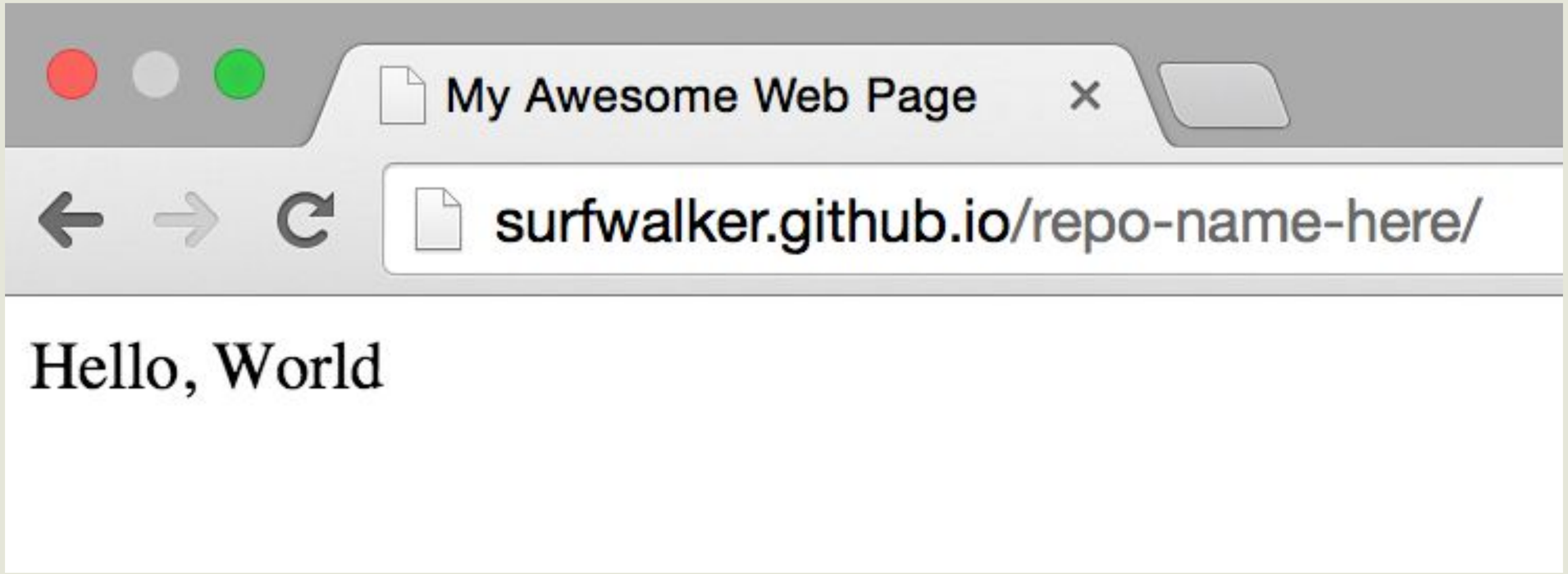
It will turn this:



Showing Off

Into a real web page. The URL for your web page is *yourGitHubUserName.github.io/your-project-name*.

Like this:



Be Amazed

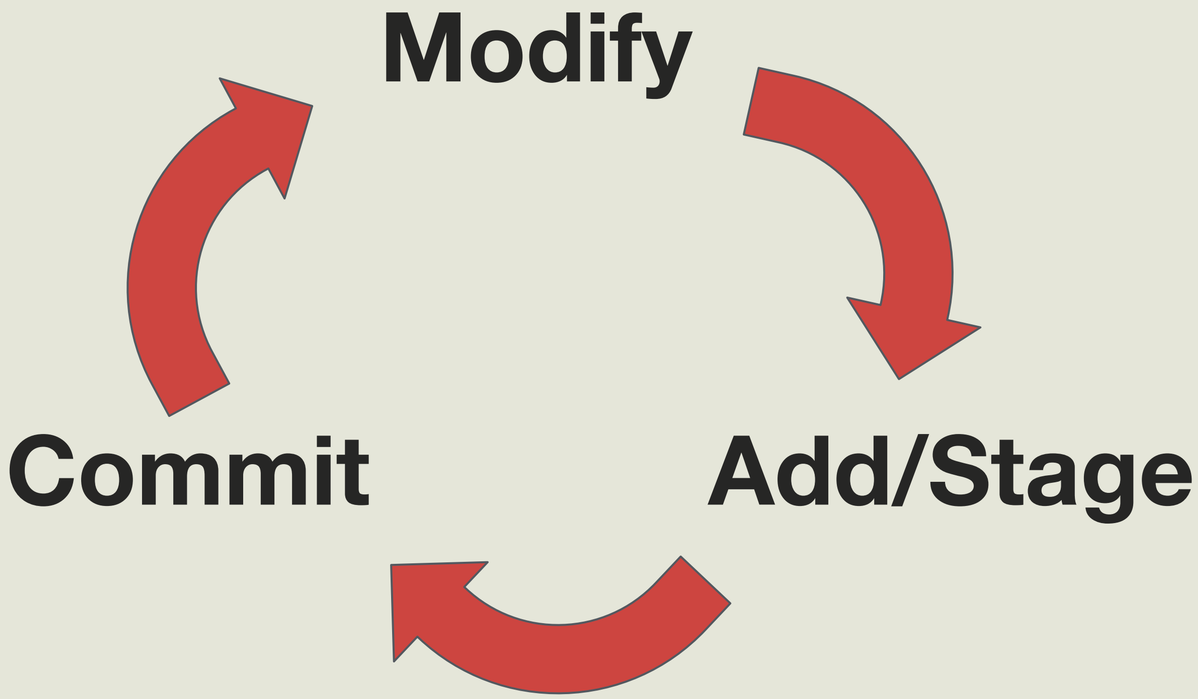
Congratulations you have successfully deployed your awesome work on the internet.

Give your neighbor a pat on the back!

ANY
QUESTIONS?

GIT WORKFLOW

Basic Git Workflow



GIT COMMAND: `status`

`git status` will provide information about the current state of your repository.

You use it to see which files need to be added and which have been changed and are awaiting commits.

You should make a habit of frequently checking the status of your repository to develop a good awareness of how things are changing.

```
$ git status
```

Tracking Changes

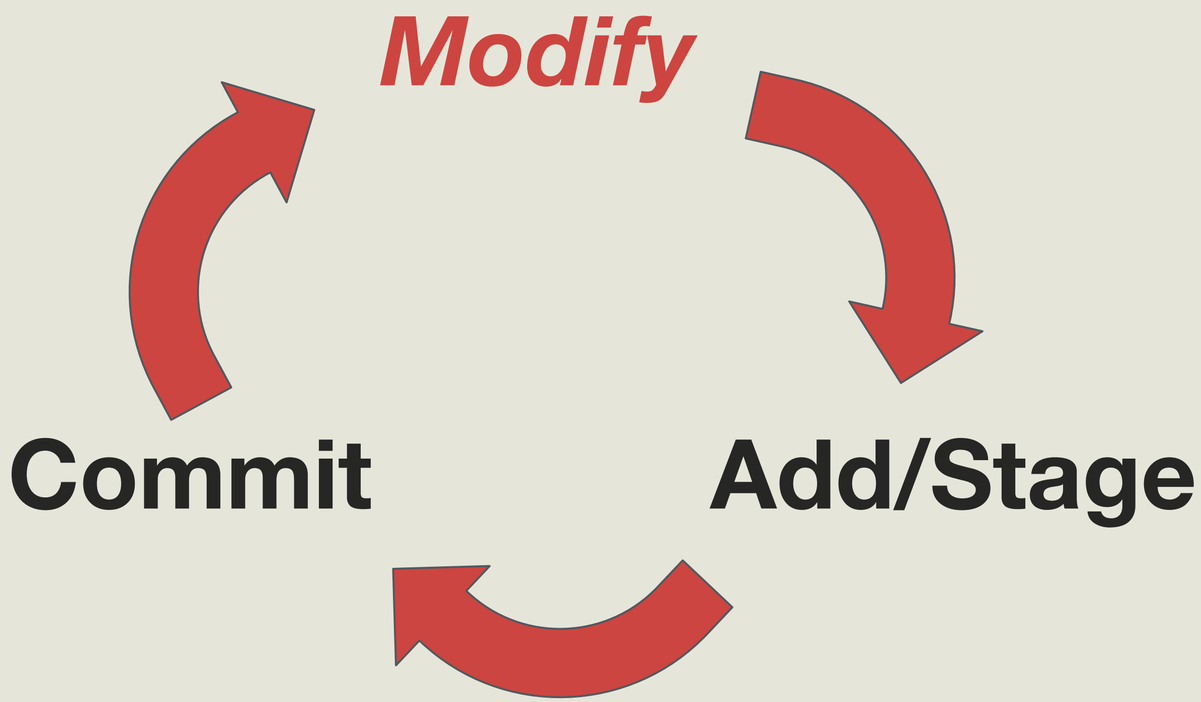
Git keeps track of all changes made to any file within your project repo.

If no changes have been made then `git status` will return this:

```
$ git status
On branch gh-pages
nothing to commit, working directory clean
```

Modifying

The first part of the Git workflow is modifying. This includes adding a new file, adding a new sub directory or making any change no matter how small to any existing project file.



Creating a New File

If you've created a new file in the project directory, `git status` will return this:

```
$ git status
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)

    kitteh_names.txt

nothing added to commit but untracked files present (use "git add" to track)
```

Notice that Git is quite verbose in telling you what's going on, and even gives hints as to the next step.

Tracking Changes

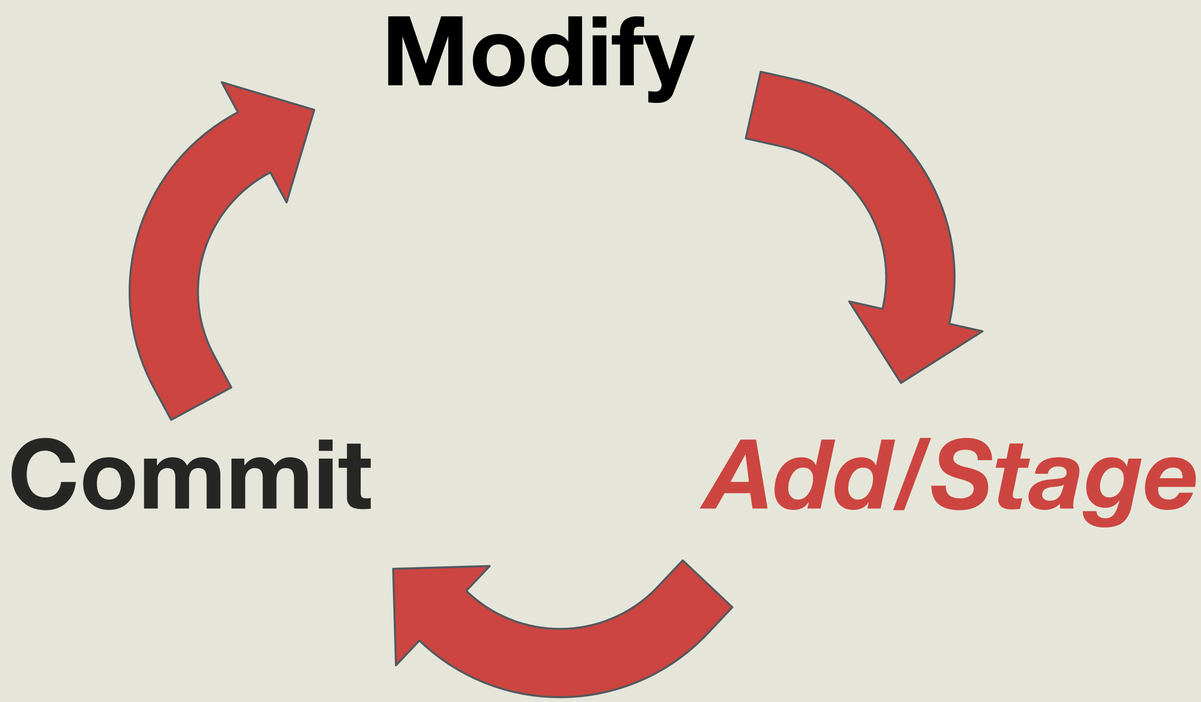
If you've made a change to an existing file, `git status` will return this:

```
  $ git status
On branch 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:   kitteh_names.txt

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

Notice that Git provides hints once again, and now you have two choices, to add the file or to discard the changes.

Adding / Staging

The next stage of the Git workflow is adding (often referred to as *staging*). When a file added, this tells Git that you care about saving the work done on that file.



GIT COMMAND: add

The add command places new files or files that have been modified from their known state to the *stage*.

Once a file has been added, Git will track specific changes to it.

```
$ git add kitteh_names.txt
```

Each modified or untracked file should be added individually.

Staged Changes

Again, we want to check the status after each step of the Git workflow:

```
$ git add kitteh_names.txt
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    modified:   kitteh_names.txt
```

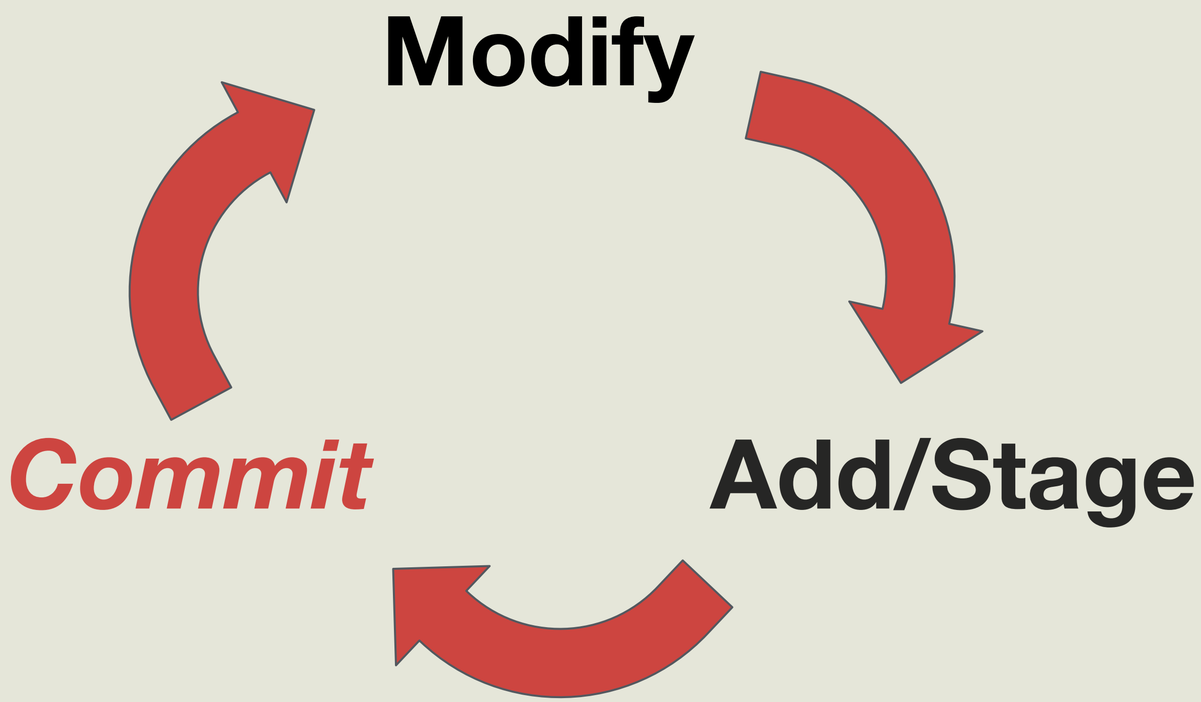
Notice that this time, the file is marked as modified instead of new

You can now commit it:

```
$ git commit -m "added note about hidden files"
[master 4eca5ad] added note about hidden files
1 file changed, 1 insertion(+)
```

Committing

The final part of the Git workflow is committing. This is when we take the actual snapshot of our project and its current state.



GIT COMMAND: commit

```
$ git commit -m 'Initial commit'
```

The commit command is used to create a permanent record of changes to your repository.

It saves all the changes that have been staged.

Each commit saves:

- The changes made to each file on the stage
- The identity of the person who made the changes
- The date and time the change was made
- A brief message about the nature of the changes made
- A universally unique identifier for the set of changes

GIT COMMAND: `commit`

Every commit requires a corresponding brief message about the nature of the changes made.

The `-m` flag allows you to input your message as a part of the commit command.

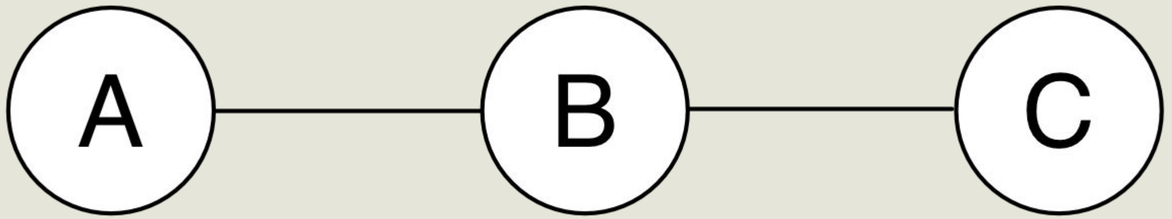
If you neglect to use the `-m` flag a program called VIM will open in your terminal in response.

If you do not type anything you can enter `:q` to escape VIM and return to your prompt and try again.

If you have entered text in VIM you must enter `:q!` to escape.

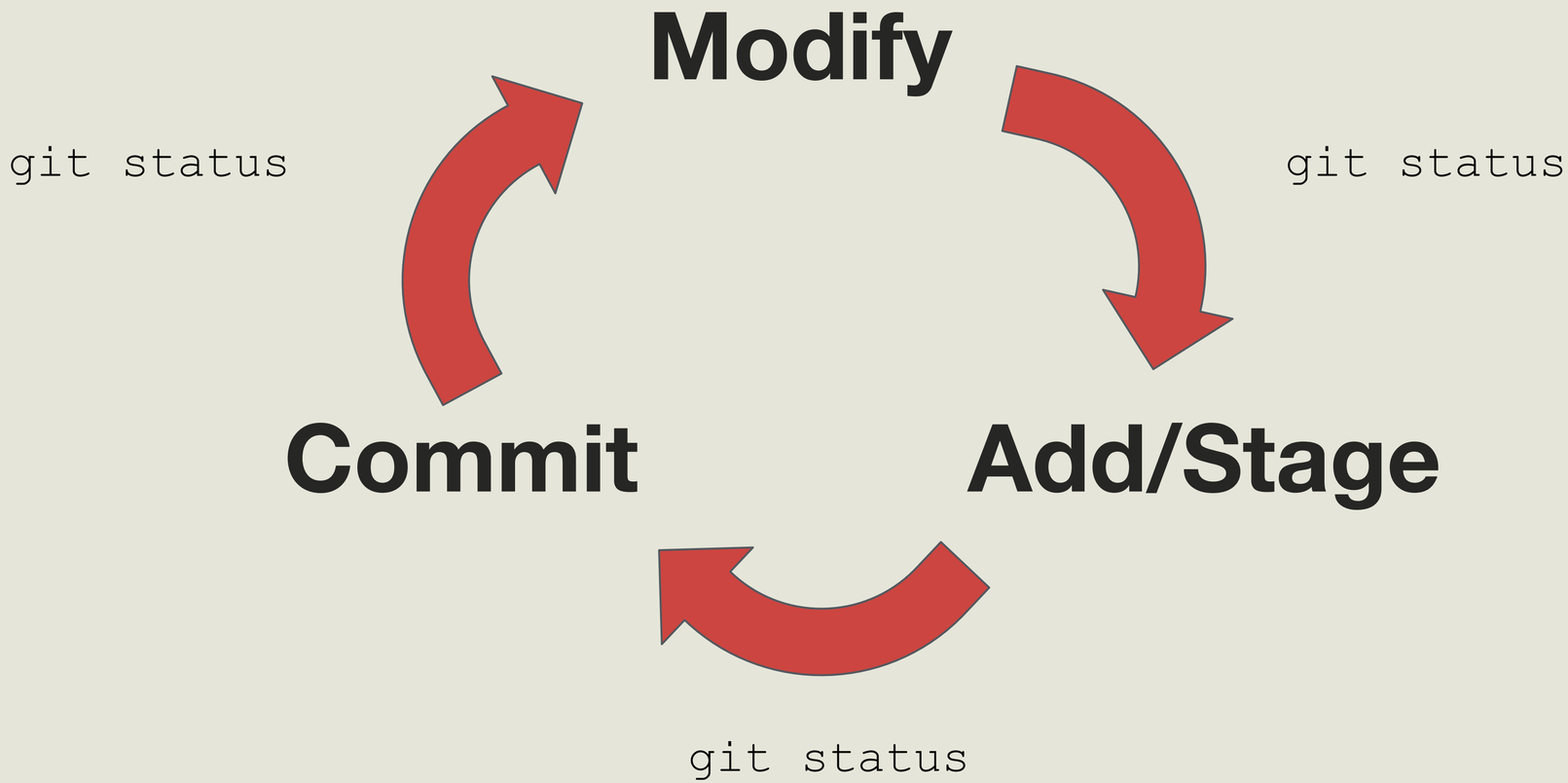
It saves all the changes that have been staged.

An Iterative Process



Each cycle of the Git workflow results in a new snapshot on your project’s timeline.

Basic Git Workflow



ANY
QUESTIONS?

Git Commands

New terminology learned in this lesson:

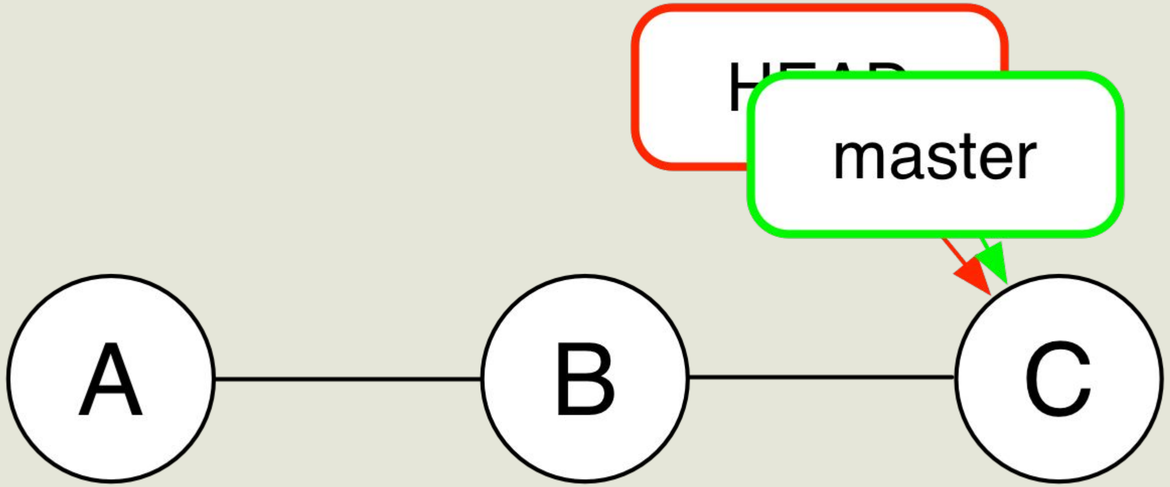
git status
Shows the state of the repository and all files in the same directory, including ones not yet added.

git add <file>
Adds a new file to the repository, or adds a modified file to the stage so they can be committed.

git commit
Commits all staged changes to files in the repository for safe keeping.

BRANCHING

A Picture of Git

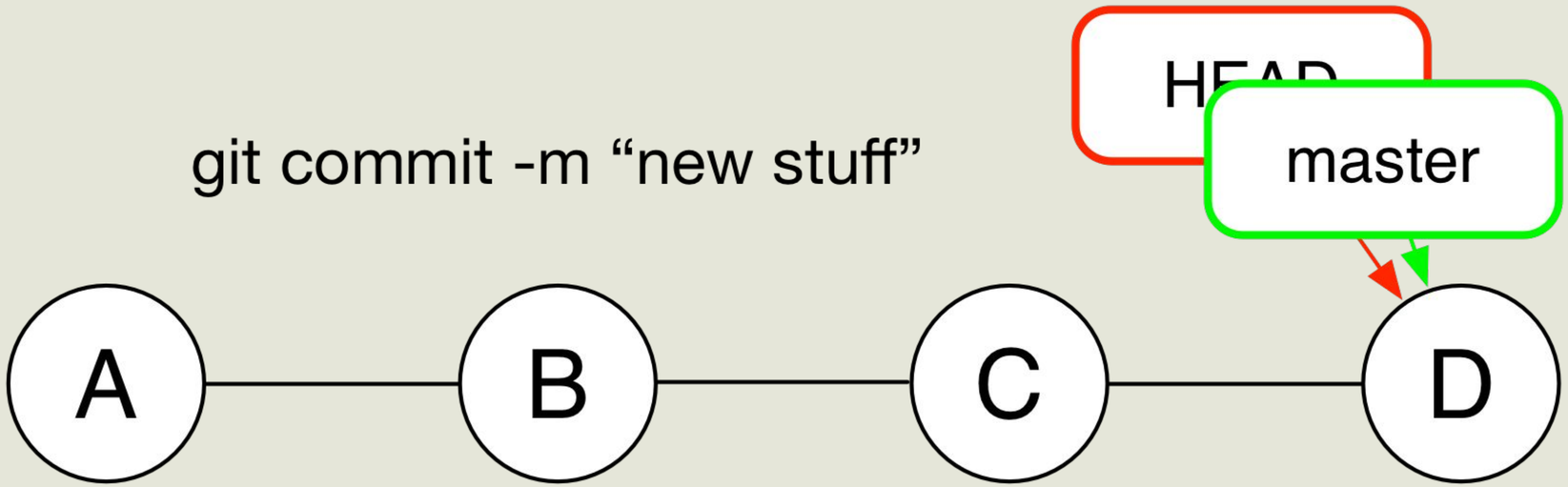


You may also be familiar with the label "master".

This is the name that Git automatically gives to the first branch in a repository.

A branch is actually just a label that points to a specific point in time.

A Picture of Git



When you make a commit in Git, you add a new point to the timeline.

The HEAD label moves to this new point.

So does the label for the branch you are on.

More Git Workflow

Remember the basic Git workflow: modify, add, commit.

Now imagine your repository is code for a vital website.

Further imagine that your production server is running using code on the master branch.

You wouldn't want anyone making willy-nilly changes to master.

It would be much better to have only tested, vetted code end up in master.

So, you ask your development team to implement fixes and features on branches.

For example:

NASA has code on GitHub. You can imagine they wouldn’t want untested code affecting their satellites and other toys.

GIT COMMAND: `checkout`

The command `checkout` has two uses.

The first takes a branch name as an argument and moves you to that branch (timeline).

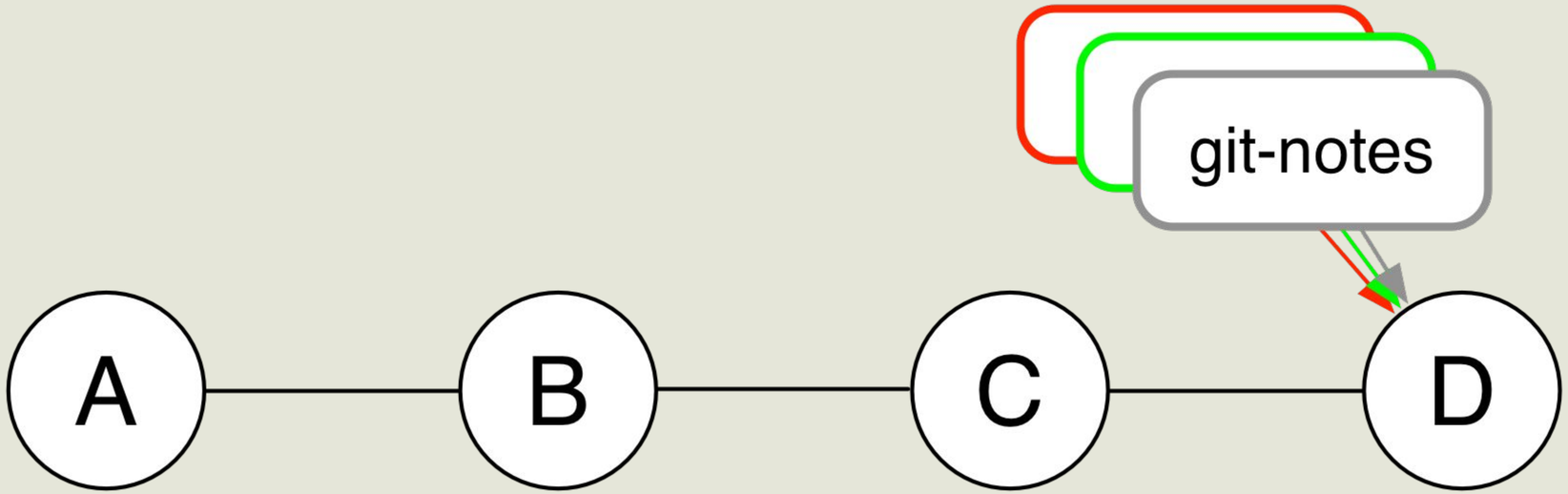
```
$ git checkout branch-name
```

But, how do you make a new branch? That's the second use.

If you enter `-b` before the argument this will create a branch with that name and move you to it.

```
$ git checkout -b new-branch-name
```

Making a Branch



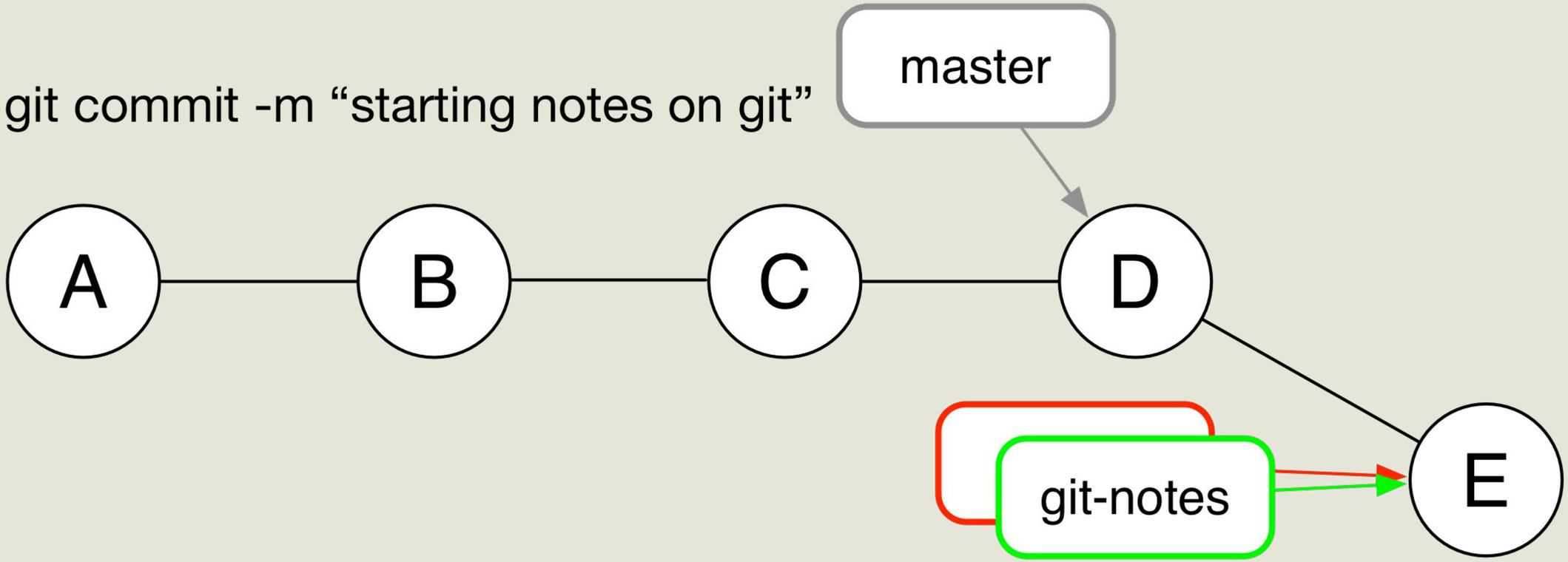
Checking out a new branch creates a new label. Since nothing has actually changed you are still on the same commit snapshot as you were previously.

GIT COMMAND: `branch`

The command `branch` will list all the branches that exist in your local repo and indicate which one you are currently on.

```
$ git branch
  kitteh-feature
  git-notes
* master
```

Visualizing the Results



Once you have completed a Git workflow cycle the branches actually diverge.

Pushing Different Branches

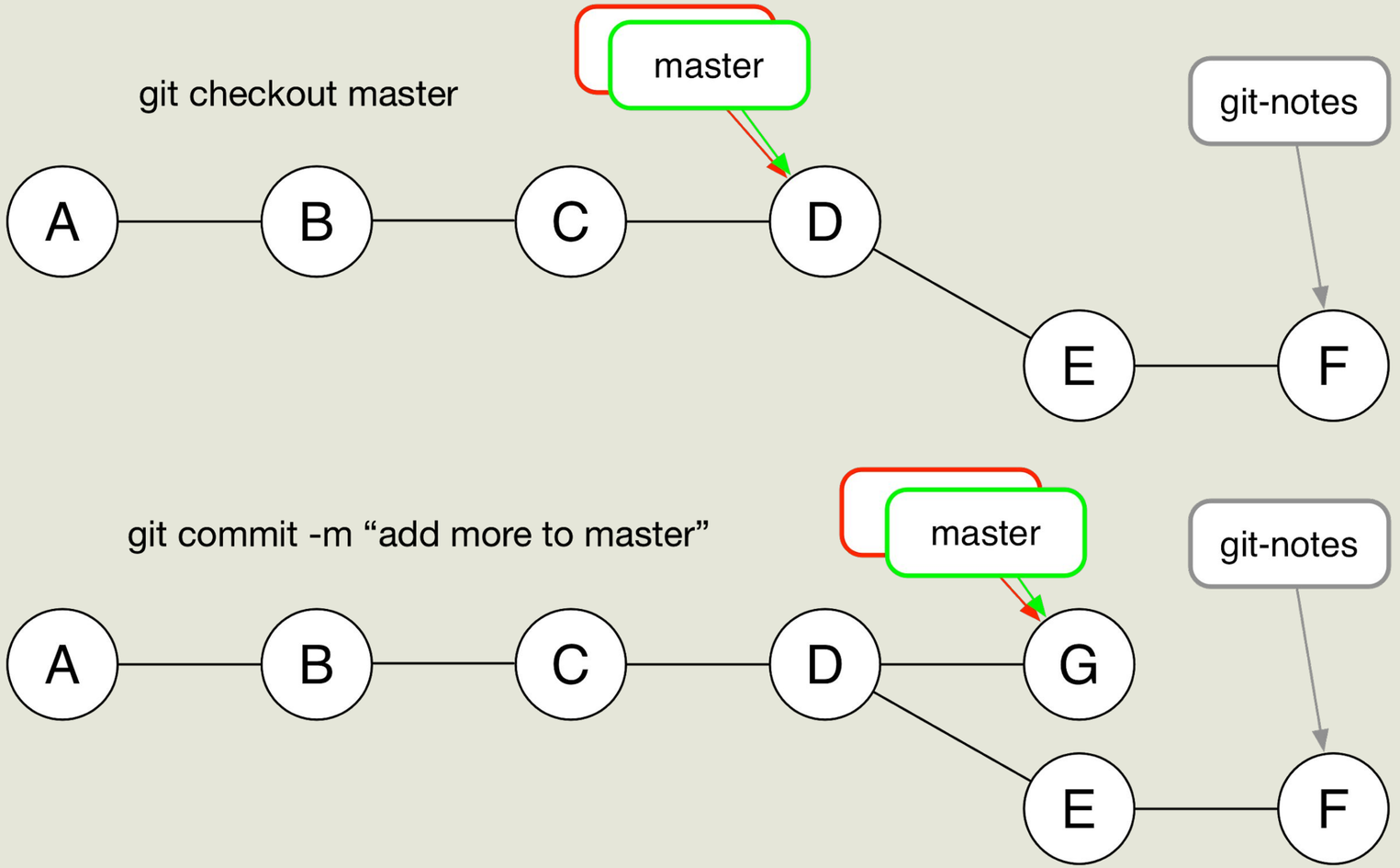
In order to push this new branch to GitHub we need to clarify in the `push` command which branch these commits should belong to.

When you do a push you always include the name of the branch that you are on locally after the `origin` alias.

```
$ git push origin git-notes
…
To git@github.com:cewing/uge-workshop.git
 * [new branch]      git-notes -> git-notes
```

Isolating your Changes

You can switch back and forth between branches to keep changes isolated to their relevant branches (timelines).



ANY
QUESTIONS?

Git Commands

New terminology learned in this lesson:

git branch
View existing branches in your repository.

git checkout
Create new branch or change the active branch.

OPEN SOURCE

Getting Someone Else's Code

Now we come to one of the most powerful features of Git and GitHub.

All code on GitHub, unless specified otherwise, is publicly accessible.

Any project you find, you can contribute to. You can fix bugs, raise issues, add new features, write documentation.

This is the heart of *open source*. Projects, and code, become better through open collaboration and the sharing of ideas and resources.

You don't have write access to just any project however. You will need to *fork* and *clone* the original repository in order to contribute.

Forking

When you *fork* a repo, you make an exact copy of it in its most current state.

Your profile on GitHub is the owner of this copy to do with whatever you like.

To *fork* a repo, navigate to that repository's main page. At the top left, you should see something like this:

Click the *Fork* button. It will go through a bit of an animation, and then you will have a shiny repo of your very own.

This information is conveyed here:

Getting it Local

In order to bring this code down to your local machine, it’s as simple as cloning.

This is the exact same step you’ve done when creating your own repos on GitHub.

GIT COMMAND: `remote`

When you fork and clone a repository, it's easy to accidentally clone the original repo.

Git offers a convenient way to check this.

The URL that your local repository is connected to is known as the *upstream*. `git remote -v` will tell you what that URL is. You'll want to make sure that the username portion is *your username*, otherwise your local repository is pointing at the wrong remote, and you will have permission errors when you try to push.

```
$ git remote -v
origin  git@github.com:yourUserName/repo-name-here.git (fetch)
origin  git@github.com:yourUserName/repo-name-here.git (push)
```

fetch refers to the URL that your local repository receives information from.

push refers to the URL that your local repository sends information to.

For all intents and purposes, these should be the same.

ANY
QUESTIONS?

Git Commands

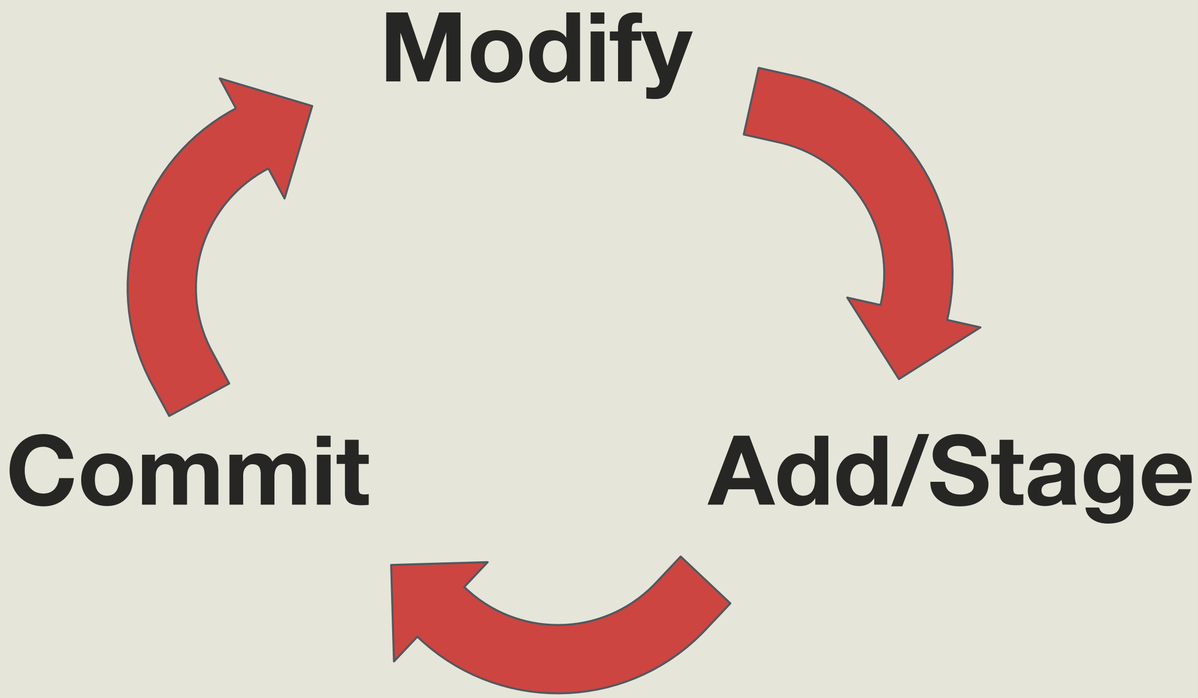
New terminology learned in this lesson:

git remote -v
Reveals the upstream of the local repository.

ADVANCED GIT WORKFLOW

More Git Workflow

Remember the basic Git workflow: modify, add, commit.



More Git Workflow

Now imagine your repository is code for a vital website.

Further imagine that your production server is running using code on the master branch.

You wouldn't want anyone (including yourself) making willy-nilly changes to master.

It would be much better to have only tested, vetted code end up in master.

So, you ask your development team to implement fixes and features on branches.

More Git Workflow

Each of these branches will be *merged* with the master branch once the code has been reviewed.

Each new feature should then branch off of this newly updated master branch.

How does this all actually work, though?

The Mighty Pull Request

When a feature is completed, you or your team will make *pull requests*.

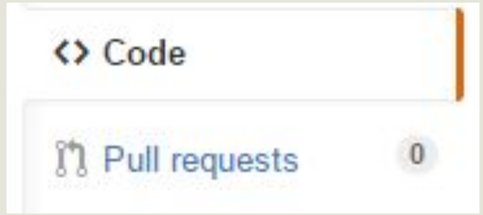
A pull request lets the manager of the project know that work is ready to be reviewed.

Developers can make changes in response to comments and get them reviewed as well.

All this is done, again, in the web browser.

Creating a PR

On the homepage of your repository, find the Pull requests link in the menu on the right:



It will open a page listing all open (unmerged) pull requests. At the start of a project, there should be none.

To initiate one, click the big, green button that says “New pull request.”

PR Setup

You'll be offered a chance to set the to and from points for your new PR.

The *base* is the branch you want to merge code into. Generally, this is the master branch.

The *compare* branch is where your new feature was completed.

Finishing Up

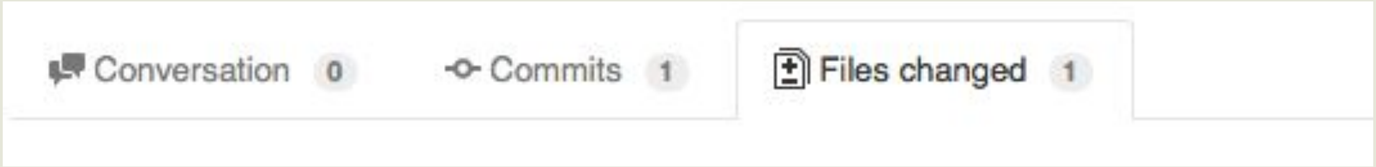
When you're set, click the “Create pull request” button.

On the next screen, enter a note about why the PR should be merged (what the branch accomplishes).

Then click Create pull request again to finalize the request.

It’s time for code review!

PR Setup



In reviewing a pull request, the owner of a project is given quite a few tools.

This tab bar shows that you can view comments made so far, the commit history, or all changed files.

Reviewing the files changed will allow you to see the line by line difference comparison of the base branch and the compare branch.

If you hover over a line, you can even leave comments on that specific line of code!

So long as the request is open, any additional commits pushed to the compare branch will automatically appear.

Merging the PR

When work is completed to everyone's satisfaction, the PR can be merged.

The manager can click on the Conversation tab and look for this green button.

It indicates that the pull request can be merged without conflict.

If not present, work will be required to resolve conflicts before a merge can be completed.

We will cover merge conflicts in a later section.

Keeping Current Locally

Now that you've merged your feature to master on GitHub, your local master branch is out-of-date.

To catch up, we have to *pull* the changes made remotely back down to the local machine.

Return to your terminal, and `checkout` the master branch of your repository.

Make sure to use `git branch` to verify that you have master checked out.

GIT COMMAND: `pull`

```
$ git pull origin master
```

`git pull` completed two steps with one command.

- It fetches changes to a named branch from the named remote.
- Then it merges those changes into the current local branch.

You can perform these steps individually in order to gain more control or better predictability for integrating changes from remotes.

GIT COMMAND: `fetch`

```
$ git fetch
```

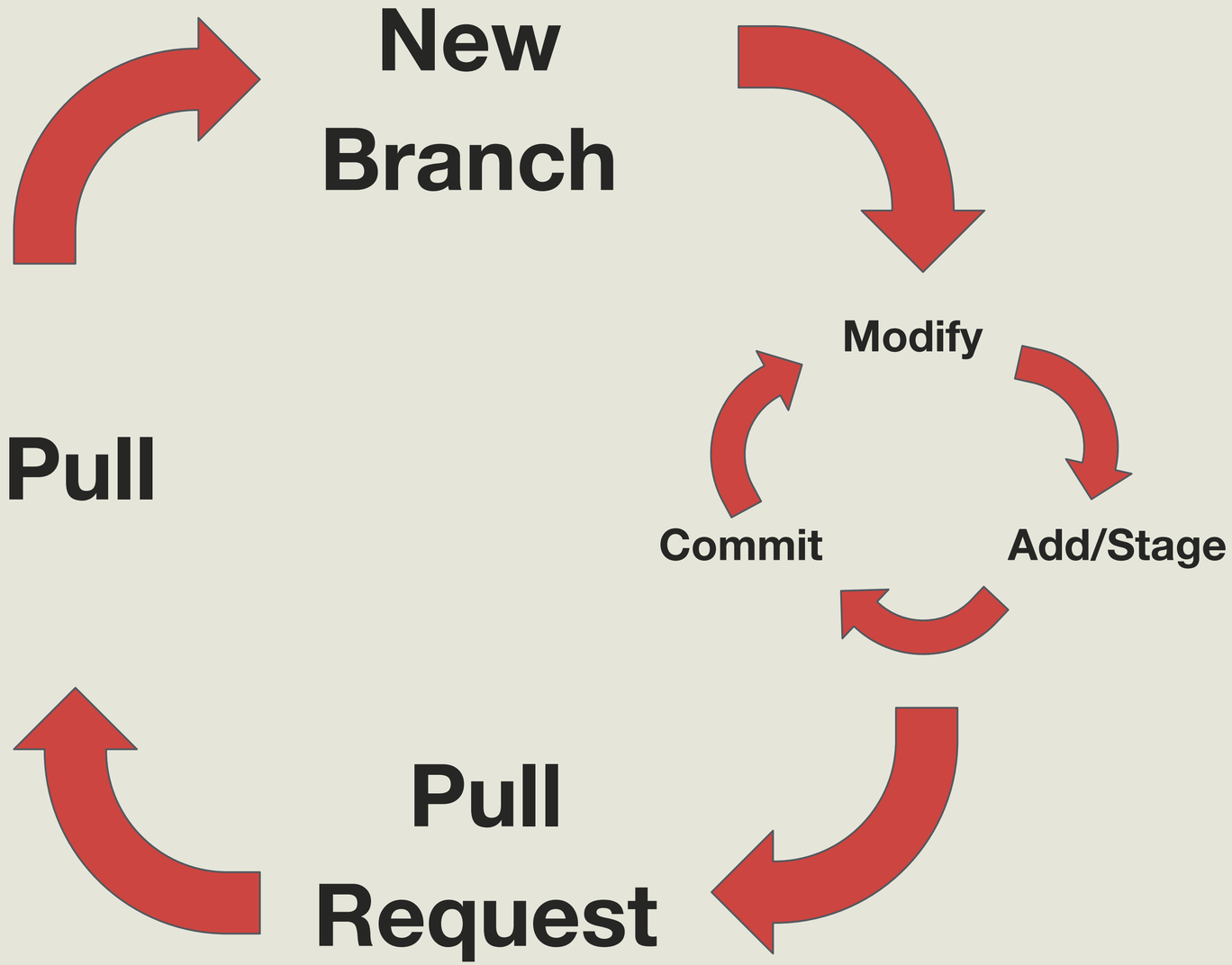
If you aren’t sure if there are differences between your local repo and the remote, this command will help you.

This command checks the remote for any new branches as well as any differences between local and remote branches.

If there is a difference you simply `checkout` to the branch with a difference and perform a `git pull` to bring in the updated code.

Advanced Git Workflow

Don’t forget to use `git status`!



ANY
QUESTIONS?

Git Commands

New terminology learned in this lesson:

git fetch
Compares the local repo to the remote to determine if changes have occurred on any branch.

git pull
Fetch and merge all changes from a remote repository branch to a local branch.

STARTING LOCALLY

Starting Out

You've already learned how to start a project on GitHub and clone it to your local machine.

But say you want to start a project without getting GitHub involved (yet)?

To do this, first create an empty directory for your project to live in, and then navigate into that directory.

GIT COMMAND: `init`

The init command creates a brand new repository in your current working directory. (remember `pwd`?)

You only need to run this command once for any project you start.

You do not run this command for projects you clone from other sources like GitHub.

```
$ git init
Initialized empty Git repository in /home/yourComputerName/directory_name.git
```

Peek Behind the Curtain

Alright, so what's actually going on here?

If you use the `ls` command inside your repository, all you'll see is your files, but there is more...

Add an `-a` flag to `ls` to see *all* items in the folder:

```
$ ls -la
total 0
drwxr-xr-x   4 myusername  computername  136 Nov 15 03:15 .
drwxr-xr-x   6 myusername  computername  204 Nov 15 03:15 ..
drwxr-xr-x  13 myusername  computername  442 Nov 15 03:33 .git
-rw-r--r--   1 myusername  computername    0 Nov 15 03:15 kitteh_names.txt
```

Whoa, more stuff! That .git directory is a *hidden folder* and is the special secret sauce.

Everything that git knows about your repository is held in that folder.

If you `init` in a directory you didn't intend, you should remove this hidden folder, which will return that directory to normal. Be careful when doing this!

CONCEPT: Hidden Files and Folders

The .git directory is an example of a *hidden folder*.

In Unix, any file whose name begins with `.` is, by default, not shown to the user unless specifically asked for.

This helps to keep the clutter associated with maintenance and configuration out of sight.

The `.` and `..` items in every directory on the filesystem are also examples of this type of file.

You know what they do, right?

Hooking it Up

Ok, you now have a great project on your local machine, but you want to make sure it is safe.

In order to connect it to GitHub, you will need to create an empty repo on GitHub to act as the *remote*.

Previously, when you cloned a GitHub repo, all that linking was taken care of for you automatically.

You are also able to manually point your local repo to a remote destination where you want it saved.

GIT COMMAND: `remote`

The remote command controls interactions with and configuration of remote repositories.

You can use it to connect new remotes, edit the status of existing connections, or remove them entirely.

By allowing connections between local and remote repositories, Git facilitates collaboration between developers.

To create a connection, use `git remote add`, which takes two arguments.

The first, `origin`, is just an alias for a much longer URL representing GitHub itself.

The second argument is the unique URL for the remote repo. This is the same URL you use to clone a repo.

```
$ git remote add origin git@github.com:yourUserName/your-remote-repo-name.git
```

Ready for More

There you go!

Your local repo is now pointing to a secure location on GitHub where your hard work can be saved.

Notice that the name of your local project directory does not have to be the same as the remote.

However, it is generally a good idea for them to at least be similar, to avoid confusion.

ANY
QUESTIONS?

Git Commands

New terminology learned in this lesson:

git init
Creates a new local repository in the present working directory.

git remote add
Connects a local repository to an existing remote repository.

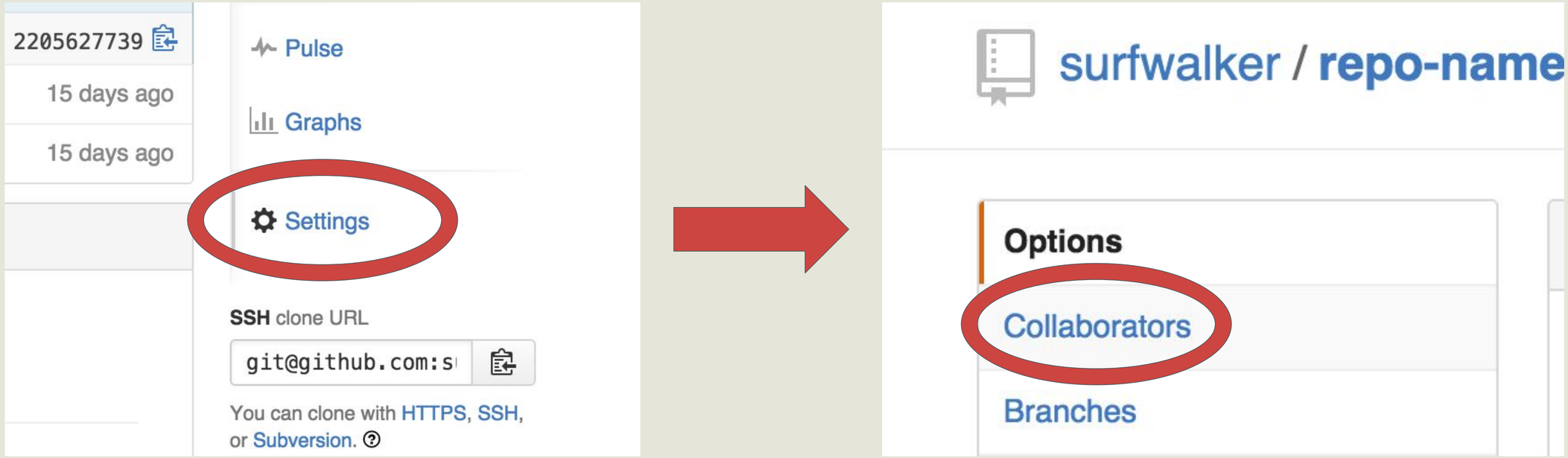
TEAM COLLABORATION

Real World

Generally, a team actually shares permissions on one repo. This helps prevent forking confusion

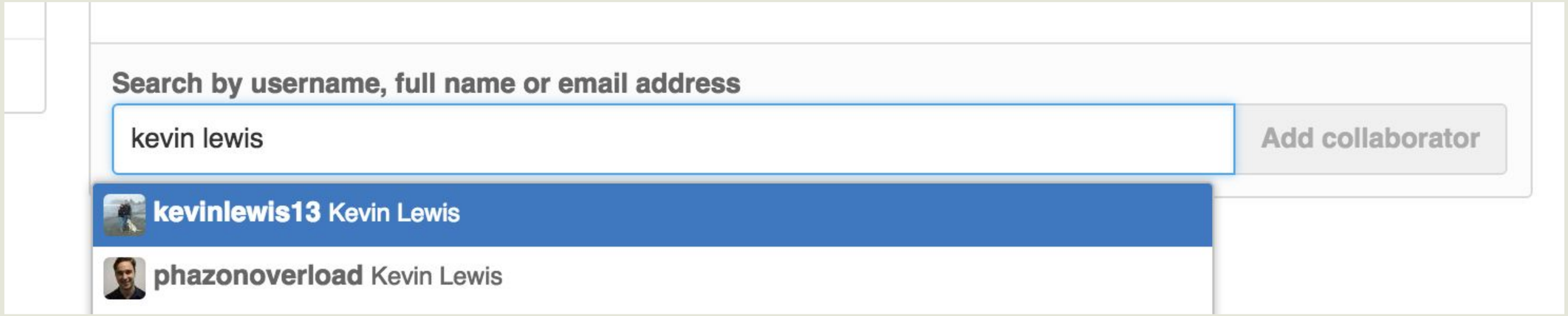
To do this, the actual owner of the repository adds each additional team member as a *collaborator*.

In order to add collaborators, the owner of the repo goes to settings then selects collaborators:



Real World

Now, the owner searches for the team members by their GitHub usernames:



They will each receive an email invitation that they must accept before they can directly contribute.

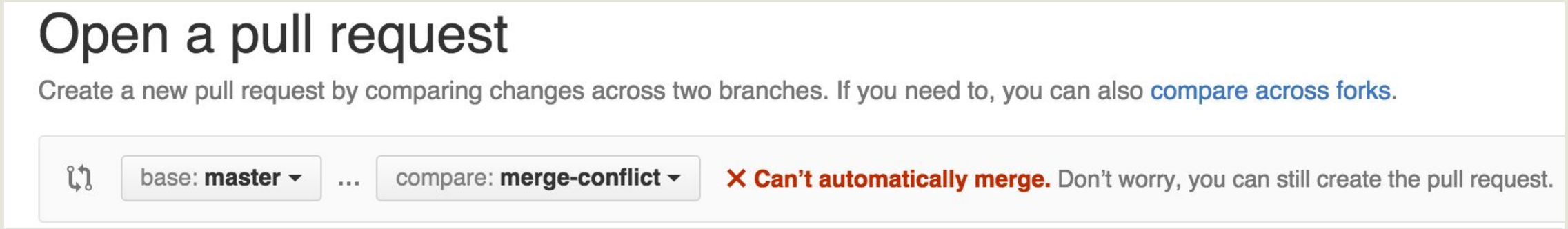
Once a team member is a collaborator, they can actually push directly to the repo, even though it is technically owned by someone else!

This is not a reason to adapt bad Git habits though. In fact, this is where all your learning regarding feature branches and pull requests will come into play the most.

Best Intentions

We are all, however, only human. Errors will happen!

The typical result of collaborating on a repository project is a *merge conflict*:

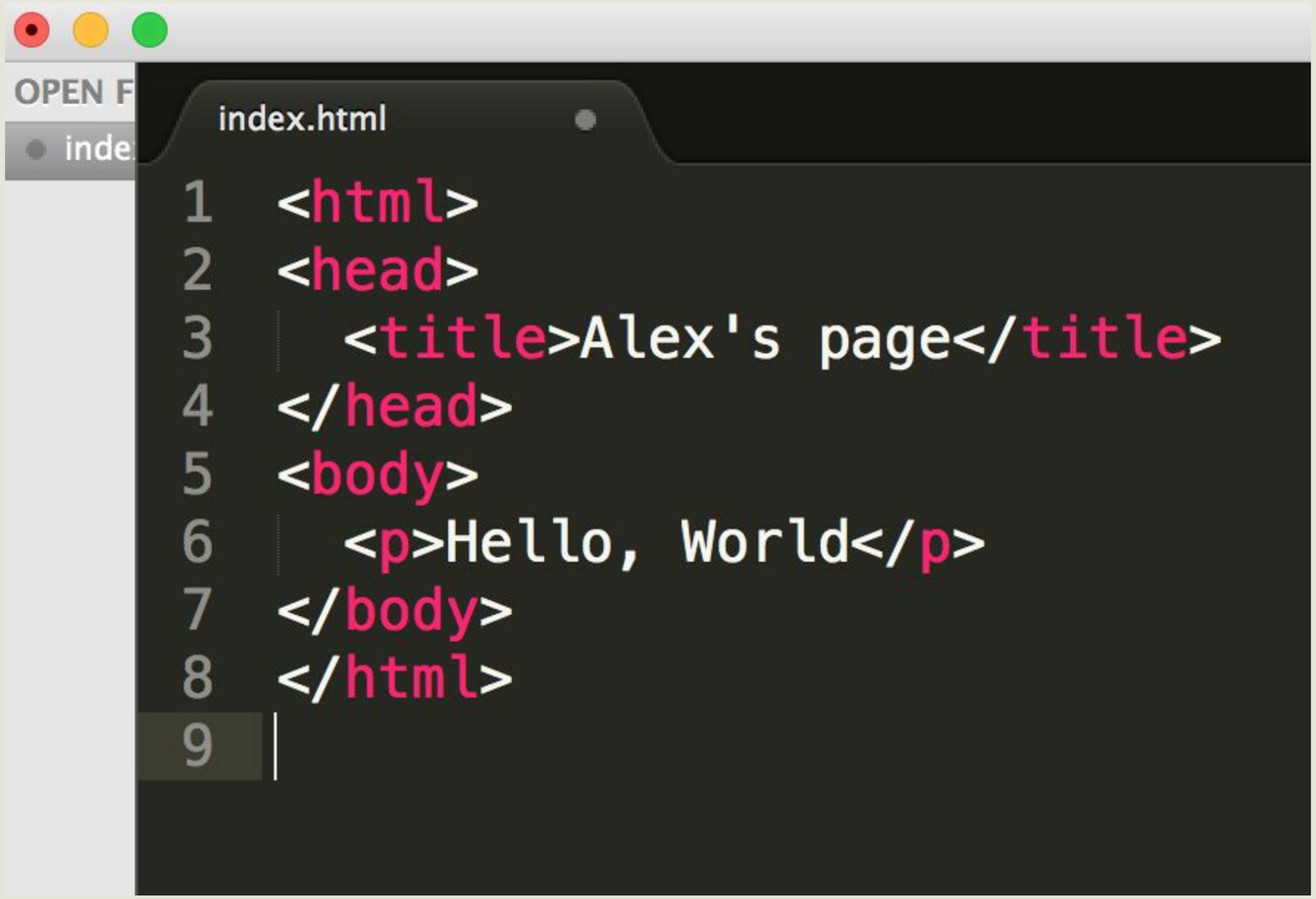


It sounds terrifying, but keep calm and code on. We will get through this.

Merge Conflicts

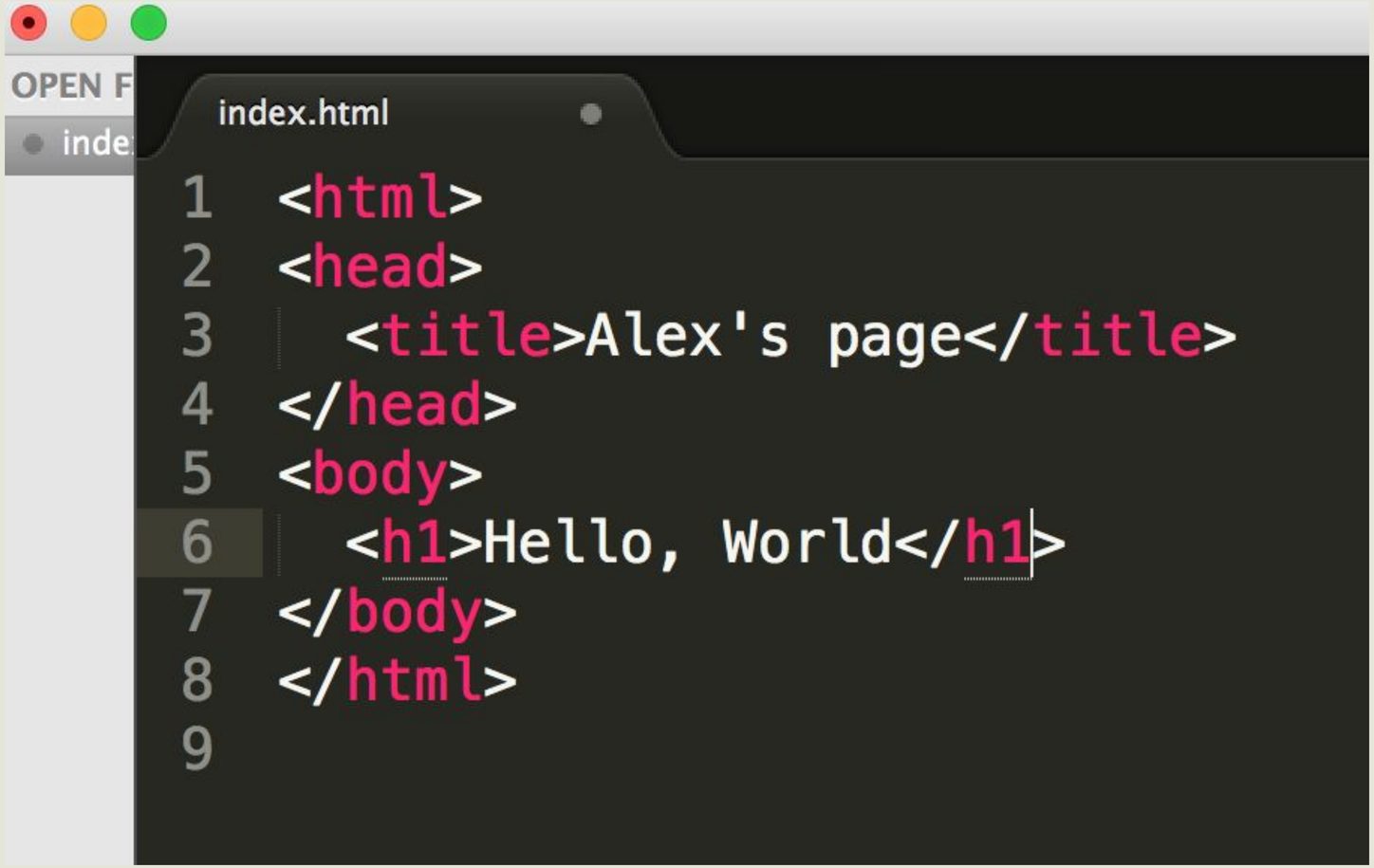
Merge conflicts arise when there are two distinct *diffs* for any individual line in the repository.

Let’s say the original code was Hello, World :



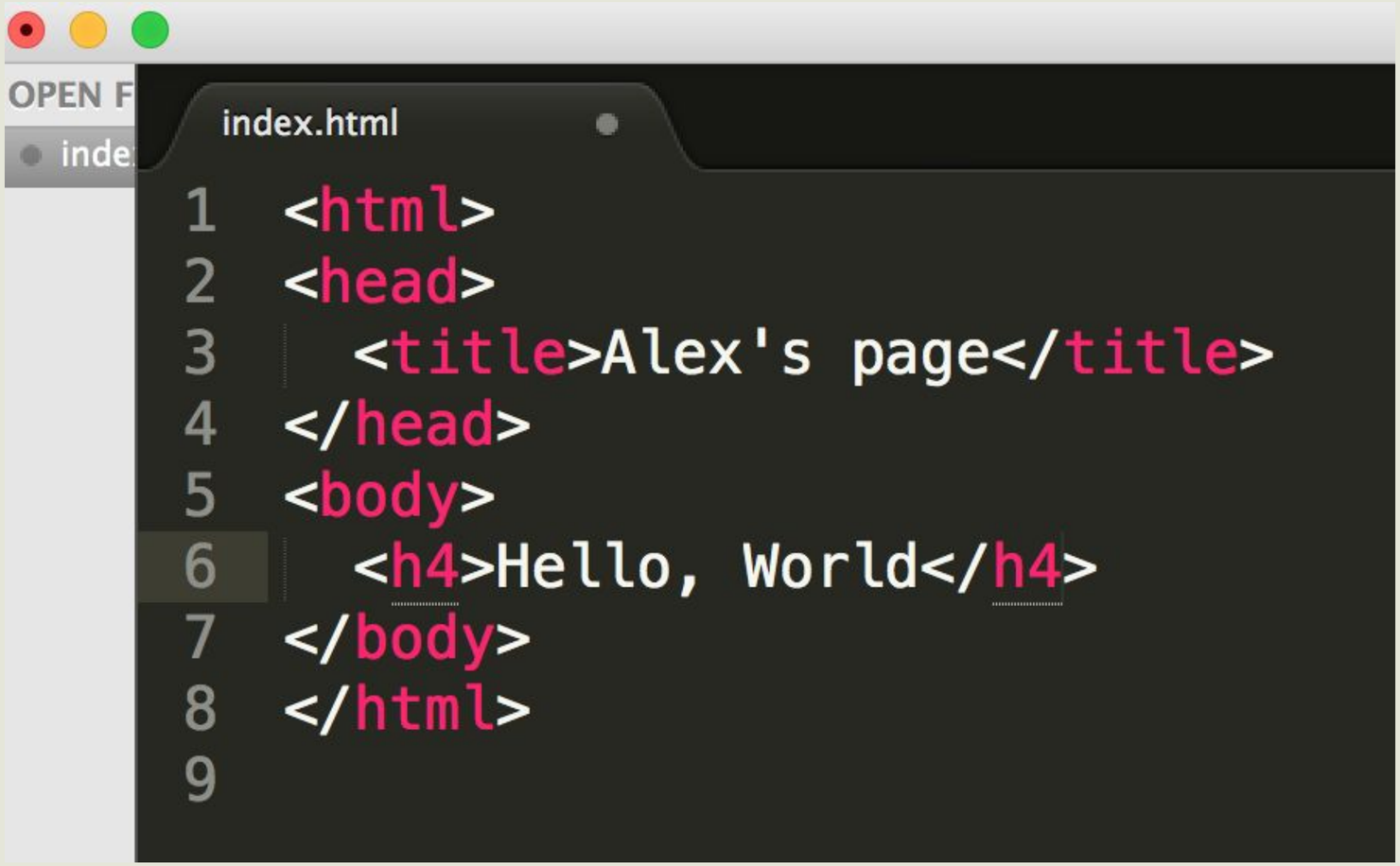
Merge Conflicts

Team member A changes that to <h1>Hello, World</h1> on her branch:



Merge Conflicts

Team member B changes that to <h4>Hello, World</h4> on his branch:



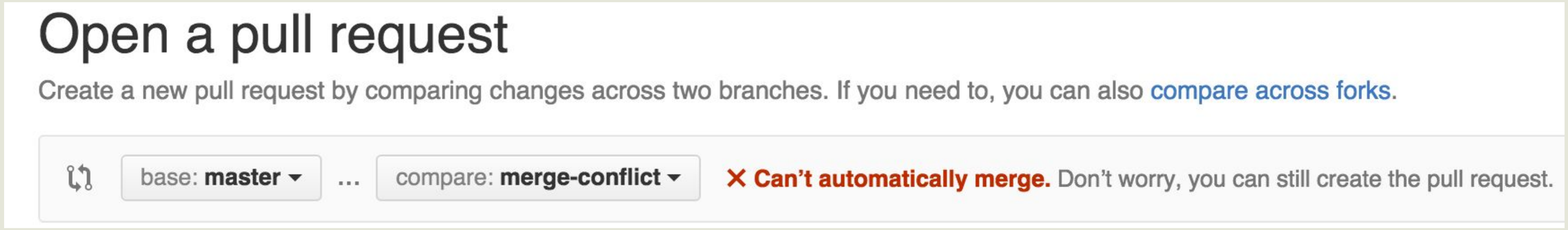
Merge Conflicts

Which one is correct?

Git is aware of these discrepancies and will make you aware of them when you try to merge. A decision will have to be made about which is the correct version and that will be the actual change made to the original code.

Encountering Merge Conflicts

Here is how you might discover a problem on GitHub:



GitHub is nice enough to give us steps on how to resolve these.

Encountering Merge Conflicts

Here is what that merge conflict looks like in your terminal (image):

Now we know what file has a problem. Let’s go fix it in our text editor.

Resolving Merge Conflicts

Here is what that merge conflict looks like in your text editor (image):

Anything after HEAD and before ==== is what exists on your local repo. Anything between ??? and ???? is what is conflicting. Frequently, you will pick bits and pieces out of both sections for your final fix.

Resolving Merge Conflicts

To complete the fix, you return to your terminal and complete a commit. You will not be able to switch branches or do anything else until a merge conflict is resolved.

ANY
QUESTIONS?

Git Commands

New terminology learned in this lesson:

git init
Creates a new repository in the present working directory
git status
Shows the state of the repository and all files in the same directory, including ones not yet added
git add <file>
Adds a new file to the repository, or adds a modified file to the stage so they can be committed
git commit
Commits all staged changes to files in the repository for safe keeping
git push
Pushes all changes from your local repository to the named remote

More Git Commands

git branch
Create and manage new and existing branches in your repository

git checkout
Change the active branch and/or the location of HEAD``

git pull
Fetch and merge all changes from a remote repository branch to a local branch

git log
Shows a list of the commits in the repository along with information about the time, owner and message associated with the change

git config
Set configuration for how Git operates, either globally or per repository

MISC SLIDES

Basic Configuration

You should also be sure to set up the basic configuration Git requires

In order to make commits, Git wants to know your name and email address

We use the config Git command to set these up:

```
$ git config --global user.name "Your Name"
$ git config --global user.email "me@mydomain.com"
```

Using this information, each time you make a commit Git will record that you made the changes, and will provide contact information for any who wish to consult with you.

GIT COMMAND: `config`

The config command sets configuration values either globally or for a single repository.

You can use it to let Git know who you are and control the way Git works for you.

You can read more about this powerful command in the Git Configuration chapter of the Pro Git book.