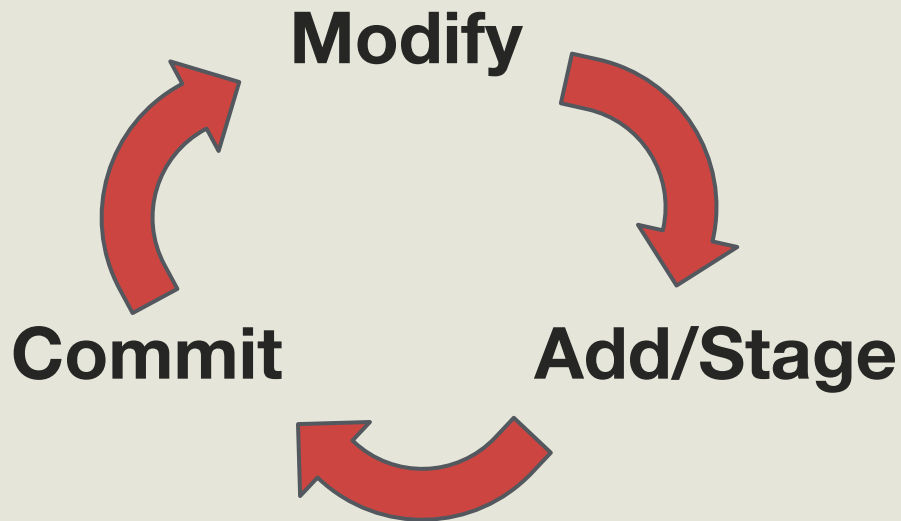


ADVANCED GIT WORKFLOW



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*.

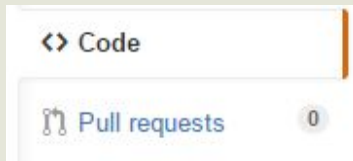
A pull request lets the manager of the project know that work is ready to be reviewed.

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:

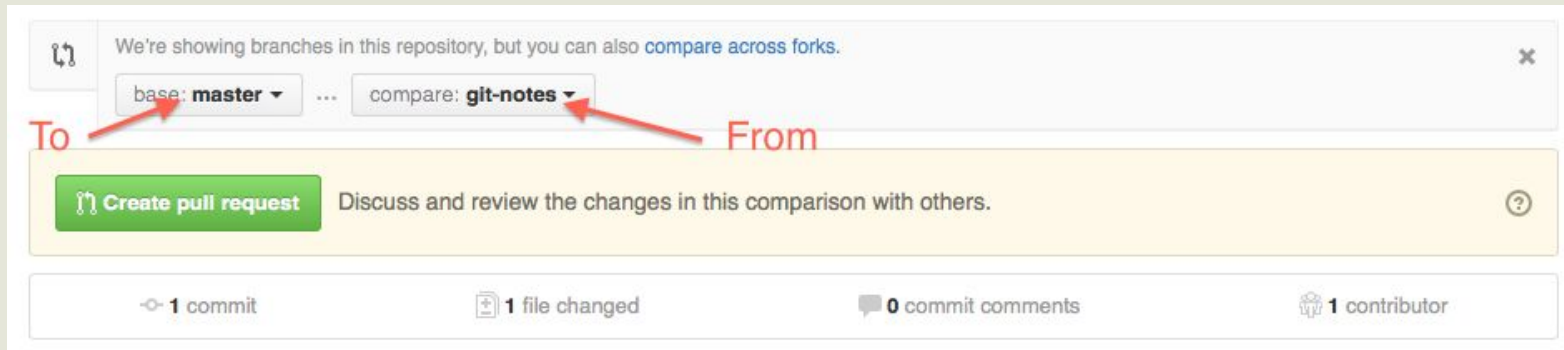


It will open a page listing all open (unmerged) pull requests. At the start of a project, there should be none.

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.

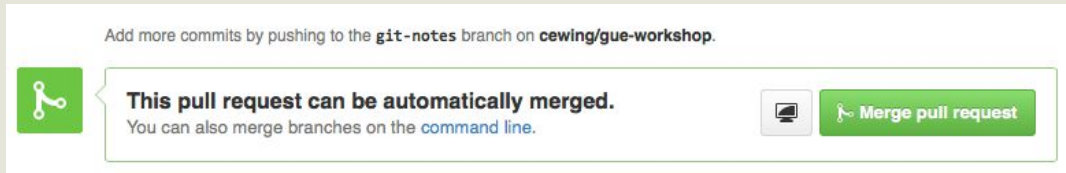
This tab bar shows that you can view comments made so far, the commit history, or all changed files.

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
```

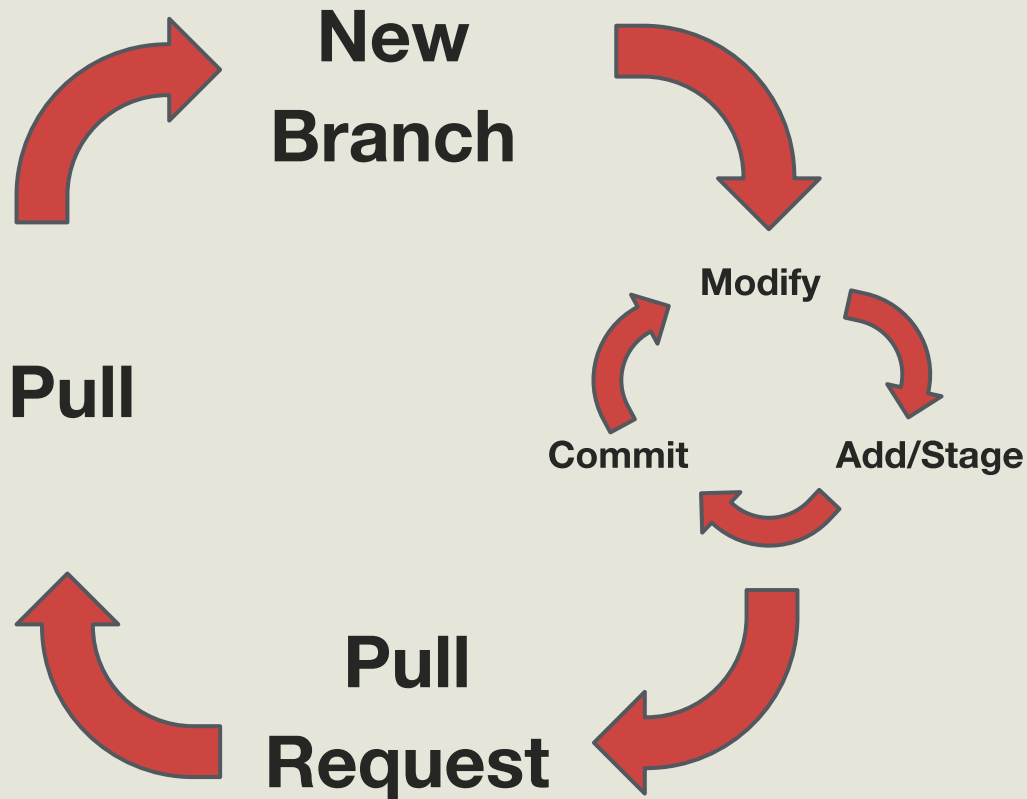
If you aren't sure if there are differences between your local repo and the remote, this command will help you.

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 kittteh_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

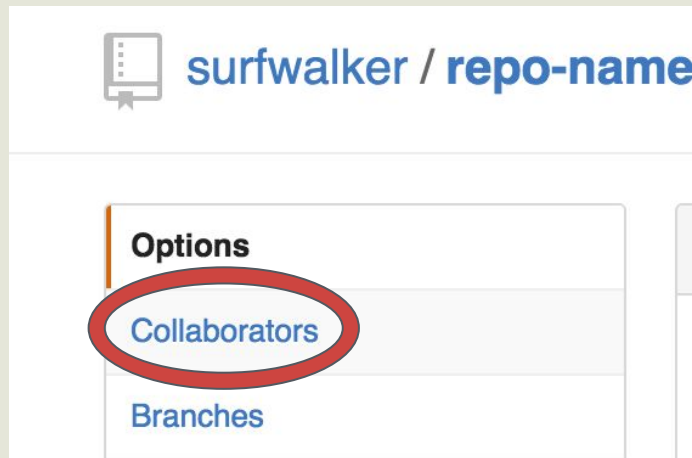
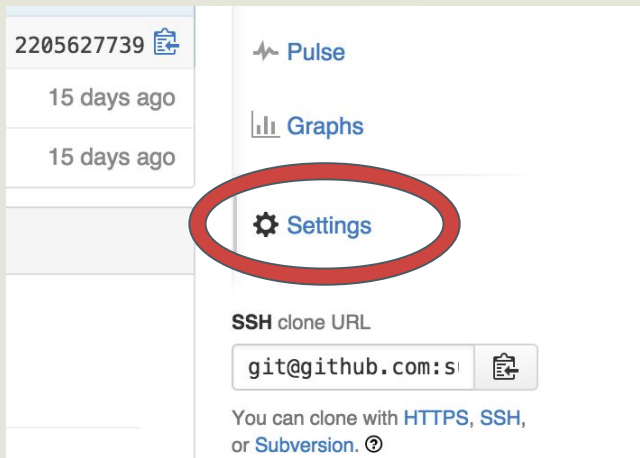


Real World

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

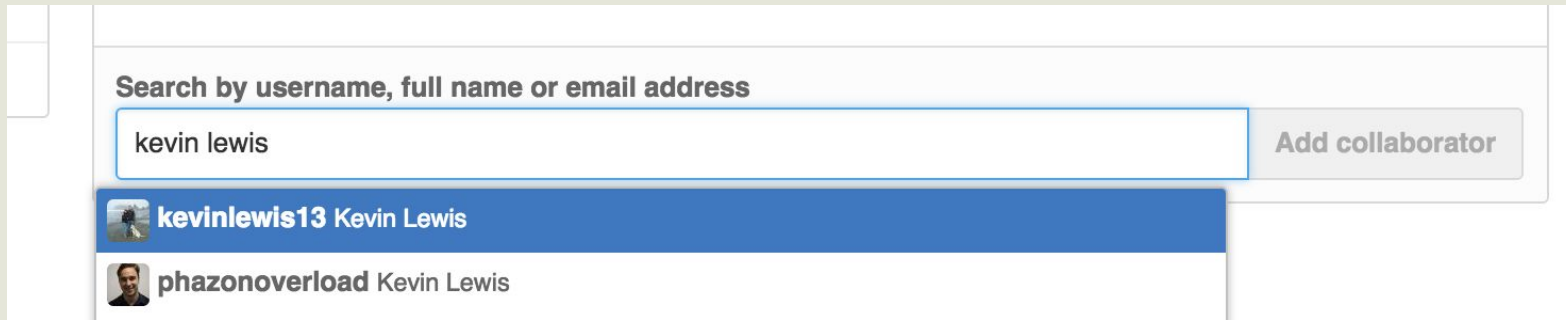
To do this, the actual owner of the repository adds each additional team member as a *collaborator*.

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:



The screenshot shows the GitHub interface for adding a collaborator. At the top, there is a search bar with the placeholder text "Search by username, full name or email address". Below the search bar, the text "kevin lewis" is entered. To the right of the search bar is a button labeled "Add collaborator". Below the search bar, a list of search results is displayed. The first result is highlighted in blue and shows a profile picture, the username "kevinlewis13", and the full name "Kevin Lewis". The second result shows a profile picture, the username "phazonoverload", and the full name "Kevin Lewis".

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*:

Open a pull request

Create a new pull request by comparing changes across two branches. If you need to, you can also [compare across forks](#).



base: **master** ▼

...

compare: **merge-conflict** ▼

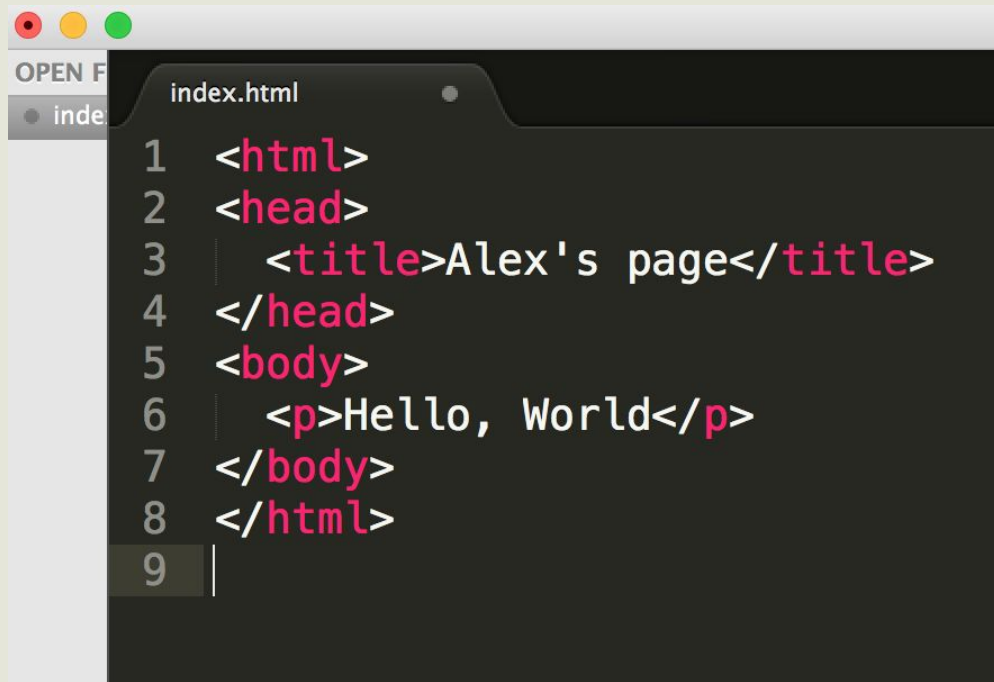
✗ Can't automatically merge. Don't worry, you can still create the pull request.

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.

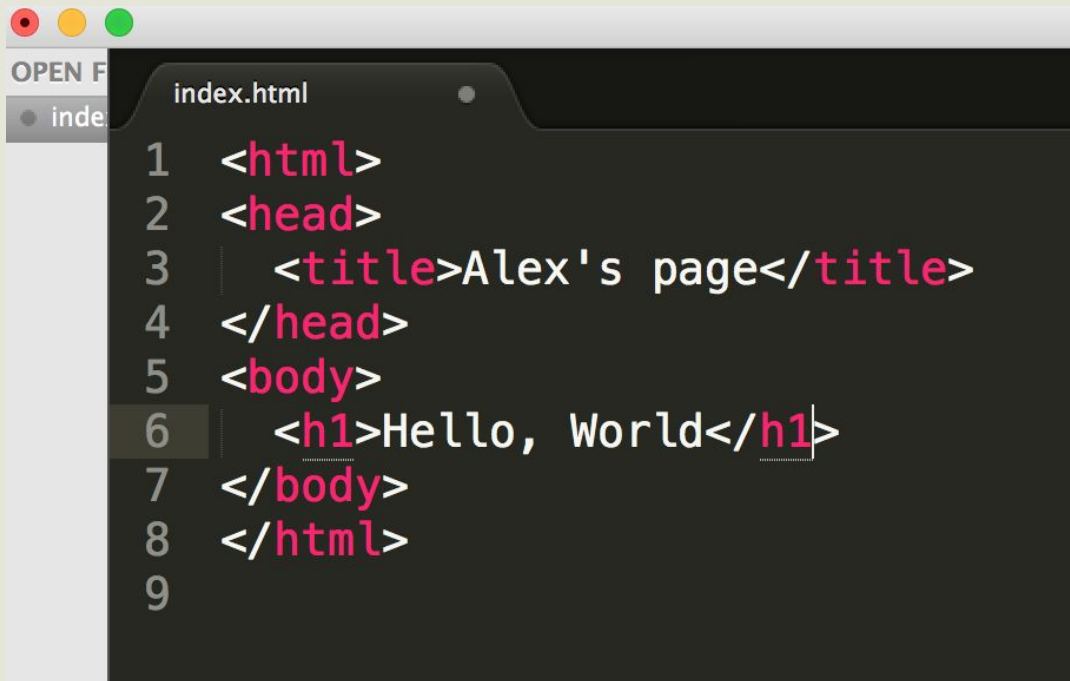
Let's say the original code was `<p>Hello, World</p>`:



```
1  <html>
2  <head>
3    <title>Alex's page</title>
4  </head>
5  <body>
6    <p>Hello, World</p>
7  </body>
8  </html>
9  |
```

Merge Conflicts

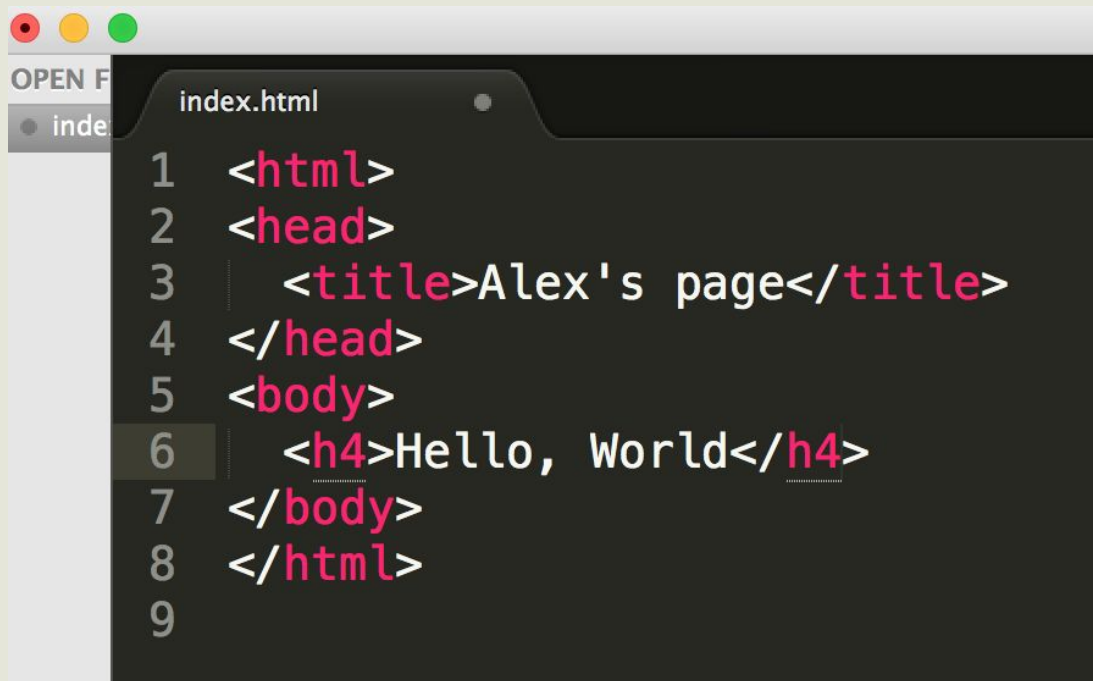
Team member A changes that to `<h1>Hello, World</h1>` on her branch:



```
1  <html>
2  <head>
3    <title>Alex's page</title>
4  </head>
5  <body>
6    <h1>Hello, World</h1>
7  </body>
8  </html>
9
```


Merge Conflicts

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



```
1 <html>
2 <head>
3   <title>Alex's page</title>
4 </head>
5 <body>
6   <h4>Hello, World</h4>
7 </body>
8 </html>
9
```

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:

Open a pull request

Create a new pull request by comparing changes across two branches. If you need to, you can also [compare across forks](#).



base: **master** ▼

...

compare: **merge-conflict** ▼

✗ Can't automatically merge. Don't worry, you can still create the pull request.

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