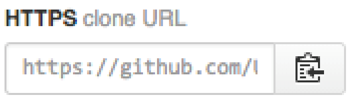
Our Git Workflow

Cloning a Repository

1. To clone a repository onto your local computer, first head to the repo's home page (for this lesson, let's clone the https://gitlab.com/esteban.vazquez/git-course)

2. On the right hand side of the page, find the clone URL, it looks like this: 

3. Copy the URL

4. In a new Terminal window, run

   ```
   git clone the_url
   ```

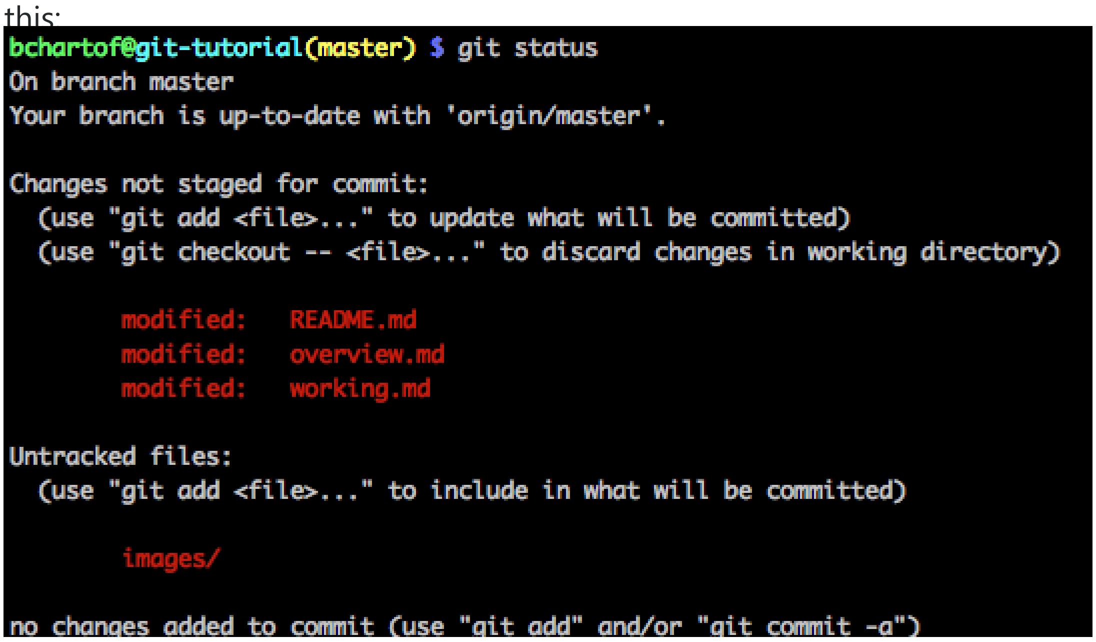
   Where `the_url` is replaced with the URL you just copied.
5. In the Terminal, to move into the folder you just cloned (which is automatically named "git-course"), run `cd git-course`

Do some work!

1. Create, delete, or edit some files or folders. If you find typos etc, feel free to edit them, but I also made a folder called workspace where you can create and edit files. Maybe make CSS stylesheet, or write a list of instructions on how to make the best turkey sandwich ever. Whatever you want.

2. Now, from inside the `git-course` folder run
   `git status`

And you should see something that looks similar to this:



```
bchartof@git-tutorial(master) $ 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:   README.md
        modified:   overview.md
        modified:   working.md

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

        images/

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

This gives you a lot of information, which I'll go through piece by piece.

- `On branch master` means I am working in the main branch, called master. See below for instructions on creating new branches.
- `Your branch is up-to-date with 'origin/master'` means that there are currently no changes ready to be pushed to the remote repo.
- The `Changes not staged for commit` section lists three files that have changed (README.md, overview.md, and working.md) since my last push, but those changes have not been saved in a commit.
- The `Untracked files` section lists files or folders that did not exist at the time of my last push, and have been created since then.

3. Before you commit the changes, you need to *add* them to the commit (sort of like putting them on deck). This step is part of the workflow in order to allow you to do a bunch of work, then separate it into a few different commits. For example, say you're working for a few hours on a graphic, in order to make it colorblind compliant. You might add all the files in a `css` folder to one commit, then push it with the commit message "updated stylesheets to satisfy colorblind requirements." Then, you might add the `README.md` file or other documentation files to a new commit, and push it with the message "Documentation now describes colorblind compliance and resources." If that terminology seems way too dense, keep reading, and hopefully things will become clear.

   You have a few options on how to add files to a commit.

   ```
   git add -A
   ```

will add all files (the three .md files as well as everything in the `images` folder, in this case) to the commit. The `-A` stands for "all."
A handy aside: if there are files you never want to add to any commit, you can make a special hidden file inside the repo called `.gitignore`*. In the [git-course .gitigore]*

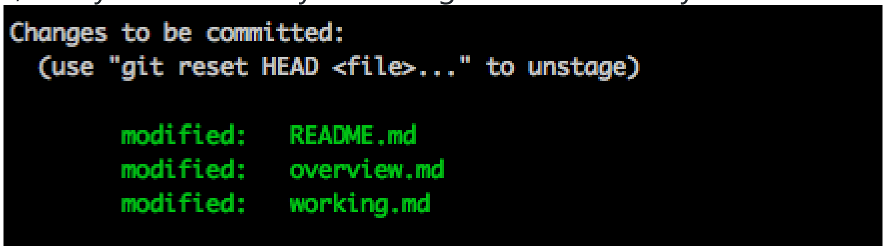
will add any files that end in ".md" to the commit. In this example, this would add README.md, overview.md, and working.md.

```
git add README.md
```

will add just one file, README.md, to the commit.

After you add all the .md files, you can run

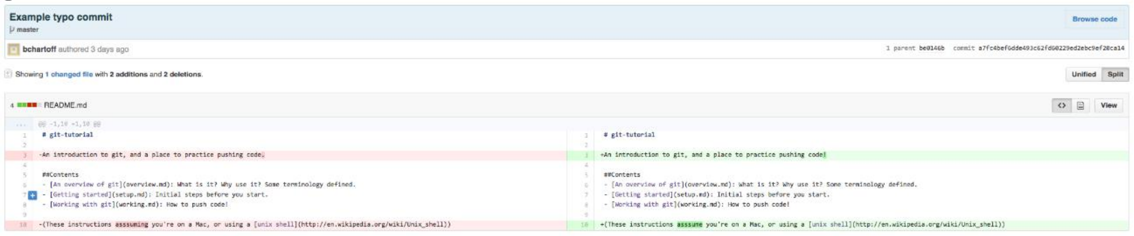
```
git status
```

again, and you'll see that your changes are now ready to be committed, yay! 

```
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        modified:   README.md
        modified:   overview.md
        modified:   working.md
```

In general, if you're not sure what the next step is, running `git status` a lot will give you an idea of what you should do.

4. Next, you need to commit the files (it's confusing that "commit" is both a verb and a noun. Really, you're "committing(v) the commit(n)" Ugh!) All commits should be described by a *commit message* that describes the changes you made, as a whole. Here's a simple example of a commit to git-course, viewed on
github.com:


Example typo commit
master
bchartoff authored 3 days ago
1 parent be0146b commit a7fc4bef6dde493c62fd60229ed2ebc9ef28ca14
Showing 1 changed file with 2 additions and 2 deletions.
4 README.md

```
@@ -1,10 +1,10 @@
 # git-tutorial

-An introduction to git, and a place to practice pushing code.
+An introduction to git, and a place to practice pushing code!

 ##Contents
 - [An overview of git](overview.md): What is it? Why use it? Some terminology defined.
 - [Getting started](setup.md): Initial steps before you start.
 - [Working with git](working.md): How to push code!

-(These instructions asssuming you're on a Mac, or using a [unix shell](http://en.wikipedia.org/wiki/Unix_shell))
+(These instructions assume you're on a Mac, or using a [unix shell](http://en.wikipedia.org/wiki/Unix_shell))
```

Note a few things. The commit message is "Example typo commit", which tells us that I (bchartoff) made the commit in order to show you an example of a simple commit to fix a typo. README.md has been changed, and github highlights lines with deletions in red (on the left), with the specific deleted characters in darker red. Added lines/ characters are shown in green on the right.

You can click on 12 commits at the top of the repo's main page.

To make a commit, run the following command:

```
git commit -m "Here is where you put your commit message"
```

Now all those changes you made are stored in a single commit.

Sometimes, you might do a lot of work for one commit, that's difficult to describe in a brief commit message. If so, you can make a longer, more organized commit message by just running:

```
git commit
```

5. Ok, you've commited your changes (you can double check with another `git status`), now you need to push them to github. First, run
   `git pull`

   This will pull changes down from the remote github repo that other users have made since the last time you pushed or pulled. It's worth noting that almost everything you've done so far in this tutorial (`add`, `status`, `commit`) can be done without an internet connection. It's only when you push or pull that you're connecting to the remote github, through the internet. Sometimes, the changes you are trying to pull down might contradict the changes you yourself have made, but not yet pushed.
   If the pull went OK, you can now run
   `git push`

   and your changes will be saved to github.

6. It's good practice to make lots of frequent commits, with descriptive messages, for a number of reasons:

   - Git makes it easy to rollback a project to the way it looked at the time of a certain commit. If you make lots of little commits, you can get make it easy to rollback a mistake or edit, without also erasing work you'd like to keep.

   - Looking at a well kept commit history, it's easy to see who worked on a project, when they worked, what they did, and why they did it. If something's broken, or you have a question, you can go to the commit history to help diagnose the problem, or figure out who to ask for help.

   - The more often you push changes, the less likely it is you'll encounter a merge conflict.

Note: Sometimes, you might want to pull changes down from github, and totally overwrite whatever local work you've done that is different from the remote version on github. **CAUTION:** running this command is sort of the equivilant of closing a file and selecting "Don't Save," so make sure you really want to do it. Without adding, commiting, pushing, or pulling, run

```
git reset --hard
```

And then

```
git pull
```

and your local repo will sync up with the remote github repo, overwriting all your local changes.

Working on branches

Sometimes, you might want to work on a separate version of a project, and make lots of frequent commits to the project (as you always should!), but you want the separate version to be isolated from the main version. The different versions of a git project are called **branches**, and the main version is always called the **master** branch. Every project has a master branch by default, but you can create as many new branches as you'd like. At any time, you can **merge** a branch back with master, pushing whatever changes you made in that branch to the master branch as well. You might want to branch a project for a number of reasons:

- Often, if you're working on new feature for a project, you want to be able to work on that new feature without stepping on the toes of other users. Say, for example, you were updating all of the Urban styles to fit a new styleguide. You could make a `new-styleguide` branch, and work on that, but while you were working, other team members could continue pushing to the master branch, without having to use your half-finished wonky styles.
- Down the road, Urban graphics or features might be directly linked to a git repo (i.e. the live version of something on our site could live as a cloned repo on one of our servers). If/when that's the case, the best practice is to **never** push directly to master. Instead, we could push changes to a branch called "staging" or something similar. On a separate server (not the one that hosts the live version), we could have a git repo that is a clone of the staging branch. After internally viewing the staging version, we would then merge the staging branch with the master branch.

Ok, so let's get branching! In the examples below, Urban has just changed leadership, and our new president, Arahsay Artellway, who wants us convert all our documentation to pig latin. I decide to do the translation on a separate branch, so that while I'm working, people can continue to view the master branch without seeing my partway done translations.

1. To create a new branch, called "piglatin" the command is:

   ```
   git checkout -b piglatin
   ```

   Next, let github know that you have create a new branch, by running

   ```
   git push --set-upstream origin piglatin
   ```

   This is a slightly different use of the "push" command from above, since you're not pushing changed files, you're pushing the fact that a new branch exists. The `--set-upstream` flag means that now, every time you type `git push` as you're working in the piglatin branch on your computer, the changes get pushed to the piglatin branch on github.
2. Now, you can work on the branch.
3. On a branch, the git workflow is exactly the same as described above. As you change files, `add`, `commit`, `pull`, and `push`.
4. If you want to switch branches, you use the `checkout` command again, but without the `-b` flag (b stands for "branch" and is just used to create a *new* branch). It's **important to remember** to commit and push your changes before switching to another branch. In some cases (and it's a bit complex to go into the specifics of which cases), not commiting and pushing can make you lose local changes (i.e. changes you've made on your own computer, but haven't yet synced up with github), when switching between branches. Once you've committed and pushed, run

   ```
   git checkout piglatin
   ```

   to switch to the piglatin branch, or

   ```
   git checkout master
   ```

   to switch back to the main branch.

5. Sometimes, you will want to combine two branches together. There are two variations on why and when you'd do this:

Merging from master into a branch

Sometimes, you'll want to keep working on a new branch, but also pull in changes that other users have made to the master branch. Using the example from above, say you're on a branch called `new-styleguide`, where you're doing some major Urban CSS updates. As you're working, folks have been building a new feature over on the `master` branch. You want to pull that new feature into your branch, so you can see how your new styles fit with it.

1. Make sure you're on the `new-styleguide` branch, or switch to it with `git checkout new-styleguide`

2. Run

   ```
   git merge master
   ```

   This pulls changes *from* master *into* new-styleguide.

3. This is called **merging** the two branches, and you might need to resolve differences that contradict each other (see below for this process).

Merging a branch into master

So you've finished all your work updating the new styleguide on your branch. High five! If you want to update the master branch with all your changes, the process is just the reverse of above:

1. Make sure you're on the `master` branch `git checkout master`

2. Then pull in your changes

   ```
   git merge new-styleguide
   ```

Deleting a branch

Finally, there are times you might want to delete a branch. Major branches, like `new-styleguide` or other big features, might be good to keep around. Lots of software products have branches for each new version (1.0, 2.0 etc), and looking back at those branches can often be helpful, to help diagnose bugs, or even let users roll back to old versions of the software. But little branches, for small fixes or updates, can sometimes be deleted. To delete a branch:

1. Switch to a branch other than the one you want to delete using `git checkout`

2. Running

   ```
   git branch
   ```

   will show you a list of all the branches on the project.

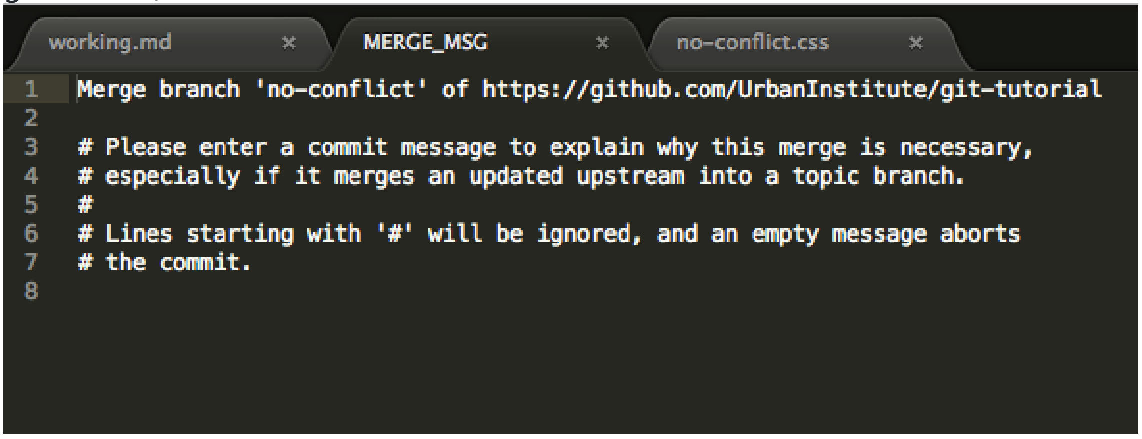
3. Then, to delete a branch called `temporary-stuff`, for example `git branch -D `temporary-stuff``

Resolving a merge conflict

When you merge a branch — either merging changes another user made on a single branch or merging one branch into another — things will easy go smoothly or slightly less smoothly.

Merging with no conflicts

If the files you merge don't conflict with each other, git does the merge automatically. When you run `git pull` or another command that causes the merge to take place, a text file will open up that looks like this (in this example, in opened in Sublime, see the setup file for instructions on using Sublime as git's editor):



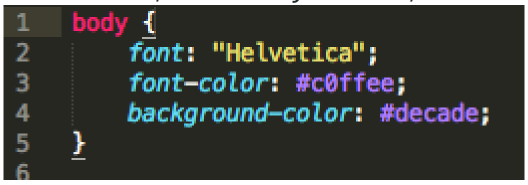
```
Merge branch 'no-conflict' of https://github.com/UrbanInstitute/git-tutorial

# Please enter a commit message to explain why this merge is necessary,
# especially if it merges an updated upstream into a topic branch.
#
# Lines starting with '#' will be ignored, and an empty message aborts
# the commit.
```

Close the file and you're done!

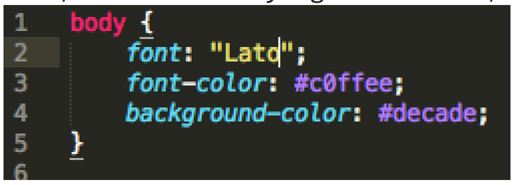
Merging with conflicts

Ok, let's look at an example of code that would cause a merge conflict. Let's say this css file, called `styles.css`, is on the `master` branch:



```css
body {
    font: "Helvetica";
    font-color: #c0ffee;
    background-color: #decade;
}
```

And, on the `new-styleguide` branch, the **same** css file looks like this:



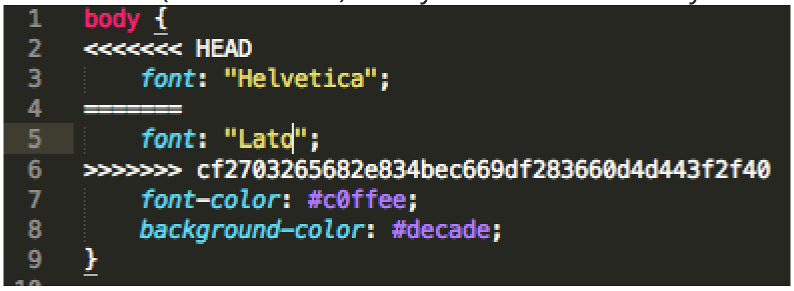
```css
body {
    font: "Lato";
    font-color: #c0ffee;
    background-color: #decade;
}
```

As you can see, the branch css has switched fonts from Helvetica to Lato. When you try to merge these two files together, git doesn't know which one is correct, i.e. they contain *conflicting* information. So you need to manually tell git which version is correct.

When you try to `git pull` to merge these two files, you will get an error message. It could say a few different things, but in general might look like:

```
CONFLICT (content): Merge conflict in styles.css
Automatic merge failed; fix conflicts and then commit the result.
```

So, there's conflicting information in `styles.css`, as we've seen. Open the file in a text editor (like Sublime) and you'll see this wacky looking business:



```
body {
<<<<<<< HEAD
    font: "Helvetica";
=======
    font: "Lato";
>>>>>>> cf2703265682e834bec669df283660d4d443f2f40
    font-color: #c0ffee;
    background-color: #decade;
}
```

This is special syntax that git puts into your files to show you where merge conflicts exist. The syntax can be read like this:

```
this;
code;
is;
OK;
<<<<<<< HEAD
Version 1 of conflicting code
=======
Version 2 of conflicting code
>>>>>>> an-id-number-for-this-merge-conflict
this;
code;
is;
ok;
```

Look back at the actual example above to make sure the syntax makes sense.

To resolve this conflict:

1. Choose which version of the code you want to keep. Say, for example, the font should be "Lato".
2. Delete the incorrect line (`font: "Helvetica";`)
3. Delete the conflict markers. I.e. delete the lines `<<<<<<< HEAD`, `=======`, and `>>>>>>> be283f461d967160c94c1d298e568aa458e1b529`
4. Save the file.
5. Now commit this file just like we've done above:

```
git add styles.css
git commit -m "resolved merge conflict"
git push
```