GIT

TO INITIALIZE YOUR NAME AND EMAIL:

git config –global user.name “YOUR NAME”

git config –global user.email “YOUR EMAIL”

git config –global color.ui auto ->enables colorization of command line output

TO EDIT NAME AND EMAIL:

git config –global --edit

1. Do git bash to the folder you want to make into repo.
2. Initialize an empty local Git repository using git init .

`ls` - *used to list files and directories*

`mkdir` - *used to create a new directory*

`cd` - *used to change directories*

`rm` - *used to remove files and directories*

GIT STATUS

`git status`

```
On branch master
Your branch is up-to-date with 'origin/master'.
nothing to commit, working directory clean
```

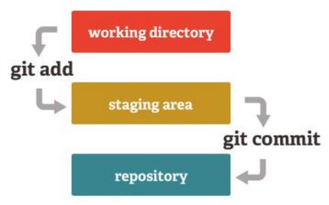
The output tells us two things:

1. `On branch master` – this tells us that Git is on the `master` branch. You've got a description of a branch on your terms sheet so this is the "master" branch (which is the default branch). We'll be looking more at branches in lesson 5
2. `Your branch is up-to-date with 'origin/master'.` – Because `git clone` was used to copy this repository from another computer, this is telling us if our project is in sync with the one we copied from. We won't be dealing with the project on the other computer, so this line can be ignored.
3. `nothing to commit, working directory clean` – this is saying that there are no pending changes.

GIT COMMIT

`git commit -m “YOUR MESSAGE”`

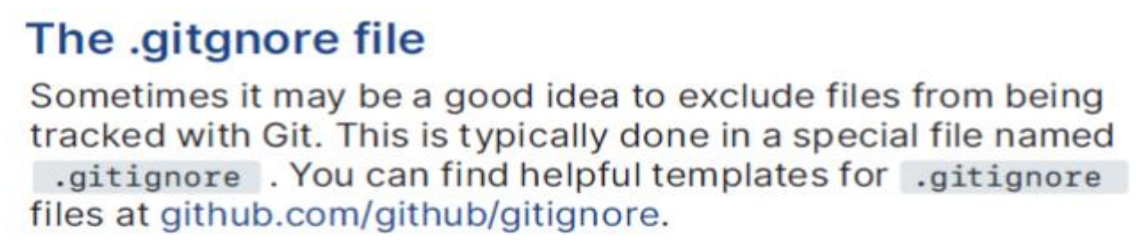
Commit is used when all the files are added in staging area by git add command.



GIT CLONE

The `git clone` command is used to create an identical copy of an existing repository. Clone(download) a repository that already exists on Github, including all of the files, branches, and commits.

```
$ git clone <path-to-repository-to-clone>
```



The .gitignore file

Sometimes it may be a good idea to exclude files from being tracked with Git. This is typically done in a special file named `.gitignore` . You can find helpful templates for `.gitignore` files at github.com/github/gitignore.

The Git Log Command

```
$ git log
```

The Log

If you're not used to a pager on the command line, navigating in Less can be a bit odd. Here are some helpful keys:

- to scroll **down**, press
  - `j` or `↓` to move *down* one line at a time
  - `d` to move by half the page screen
  - `f` to move by a whole page screen
- to scroll **up**, press
  - `k` or `↑` to move _up_ one line at a time
  - `u` to move by half the page screen
  - `b` to move by a whole page screen
- press `q` to **quit** out of the log (returns to the regular command prompt)

git log --oneline

the `--oneline` flag is used to alter how `git log` displays information:

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

This command:

- lists one commit per line
- shows the first 7 characters of the commit's SHA
- shows the commit's message

NEXT

`git log --stat`

the `--stat` flag is used to alter how `git log` displays information:

```
$ git log --stat
```

This command:

- displays the file(s) that have been modified
- displays the number of lines that have been added/removed
- displays a summary line with the total number of modified files and lines that have been added/removed

`git log -p`

The `git log` command has a flag that can be used to display the actual changes made to a file. The flag is `--patch` which can be shortened to just `-p`:

```
$ git log -p
```

Annotated `git log -p` Output

Using the image above, let's do a quick recap of the `git log -p` output:

- ◍ - the file that is being displayed
- ◈ - the hash of the first version of the file and the hash of the second version of the file
    - not usually important, so it's safe to ignore
- ♡ - the old version and current version of the file
- 🔍 - the lines where the file is added and how many lines there are
    - `-15,83` indicates that the old version (represented by the `-`) started at line 15 and that the file had 83 lines
    - `+15,85` indicates that the current version (represented by the `+`) starts at line 15 and that there are now 85 lines...these 85 lines are shown in the patch below
- ✏ - the actual changes made in the commit
    - lines that are red and start with a minus (`-`) were in the original version of the file but have been removed by the commit
    - lines that are green and start with a plus (`+`) are new lines that have been added in the commit

`git log -p` Recap

To recap, the `-p` flag (which is the same as the `--patch` flag) is used to alter how `git log` displays information:

```
$ git log -p
```

This command adds the following to the default output:

- displays the files that have been modified
- displays the location of the lines that have been added/removed
- displays the actual changes that have been made

What does `git show` do?

The `git show` command will show *only one commit*. So don't get alarmed when you can't find any other commits - it only shows one. The output of the `git show` command is exactly the same as the `git log -p` command. So by default, `git show` displays:

- the commit
- the author
- the date
- the commit message
- the patch information

However, `git show` can be combined with most of the other flags we've looked at:

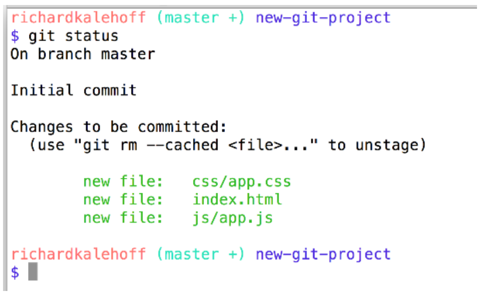
- `--stat` - to show the how many files were changed and the number of lines that were added/removed
- `-p` or `--patch` - this the default, but if `--stat` is used, the patch won't display, so pass `-p` to add it again
- `-w` - to ignore changes to whitespace

Staging Files

Terminal which uses `git add` to add `index.cpp` to the Staging Index:

```
$ git add index.cpp
```

COMMIT:



```
richardkalehoff (master +) new-git-project
$ git status
On branch master

Initial commit

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

        new file:   css/app.css
        new file:   index.html
        new file:   js/app.js

richardkalehoff (master +) new-git-project
$ 
```

Git Diff Recap

To recap, the `git diff` command is used to see changes that have been made but haven't been committed, yet:

```
$ git diff
```

This command displays:

- the files that have been modified
- the location of the lines that have been added/removed
- the actual changes that have been made

Git Tag Recap

To recap, the `git tag` command is used to add a marker on a specific commit. The tag does not move around as new commits are added.

```
$ git tag -a beta
```

This command will:

- add a tag to the most recent commit
- add a tag to a specific commit *if a SHA is passed*

delete a tag

```
$ git tag -d beta
```

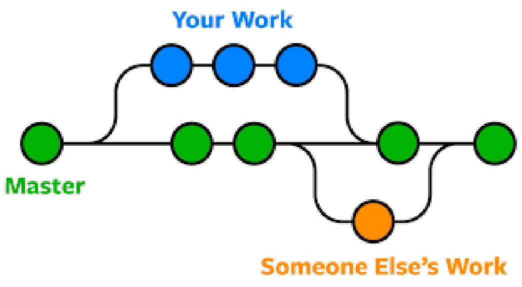
The `git branch` command

The `git branch` command is used to interact with Git's branches:

```
$ git branch
```

It can be used to:

- list all branch names in the repository
- create new branches
- delete branches



Git Branch Recap

The `git branch` command is used to manage branches in Git:

```
# to list all branches
$ git branch

# to create a new "sidebar" branch
$ git branch sidebar

# to delete the "sidebar" branch
$ git branch -d sidebar

# to merge the current branch to given branch
$ git merge sidebar

# to switch to other branch
$ git checkout [branch name]
```

The `GIT CHECKOUT`

Remember that when a commit is made that it will be added to the current branch. So even though we created the new `sidebar`, no new commits will be added to it since we haven't *switched to it*, yet. If we made a commit right now, that commit would be added to the `master` branch, *not* the `sidebar` branch. We've already seen this in the demo, but to switch between branches, we need to use Git's `checkout` command.

```
$ git checkout sidebar
```

It's important to understand how this command works. Running this command will:

- remove all files and directories from the Working Directory that Git is tracking
    - (files that Git tracks are stored in the repository, so nothing is lost)
- go into the repository and pull out all of the files and directories of the commit that the branch points to
- go to any other branch and do some changes in the desired branch.

So this will remove all of the files that are referenced by commits in the master branch. It will replace them with the files that are referenced by the commits in the sidebar branch. This is very important to understand, so go back and read these last two sentences.

The funny thing, though, is that both `sidebar` and `master` are pointing *at the same commit*, so it will look like nothing changes when you switch between them. But the command prompt will show "sidebar", now:

GIT MERGE

git merge “branch name”

- want to merge existing branch with your branch.

Revert

To recap, the `git revert` command is used to reverse a previously made commit:

```
$ git revert <SHA-of-commit-to-revert>
```

This command:

- will undo the changes that were made by the provided commit

- creates a new commit to record the change

Reset Recap

To recap, the `git reset` command is used erase commits:

```
$ git reset <reference-to-commit>
```

It can be used to:

- move the HEAD and current branch pointer to the referenced commit
- erase commits with the `--hard` flag
- moves committed changes to the staging index with the `--soft` flag
- unstages committed changes `--mixed` flag

Typically, ancestry references are used to indicate previous commits. The ancestry references are:

- `^` – indicates the parent commit
- `~` – indicates the first parent commit

Changing The Last Commit

You've already made plenty of commits with the `git commit` command. Now with the `--amend` flag, you can alter the *most-recent* commit.

```
$ git commit --amend
```

If your Working Directory is clean (meaning there aren't any uncommitted changes in the repository), then running `git commit --amend` will let you provide a new commit message. Your code editor will open up and display the original commit message. Just fix a misspelling or completely reword it! Then save it and close the editor to lock in the new commit message.

Redo commits

Erase mistakes and craft replacement history

```
$ git reset [commit]
```

Undoes all commits after [commit], preserving changes locally

```
$ git reset --hard [commit]
```

Discards all history and changes back to the specified commit

CAUTION! Changing history can have nasty side effects. If you need to change commits that exist on GitHub (the remote), proceed with caution. If you need help, reach out at github.community or contact support.

Make changes

Browse and inspect the evolution of project files

```
$ git log
```

Lists version history for the current branch

```
$ git log --follow [file]
```

Lists version history for a file, including renames

```
$ git diff [first-branch]...[second-branch]
```

Shows content differences between two branches

```
$ git show [commit]
```

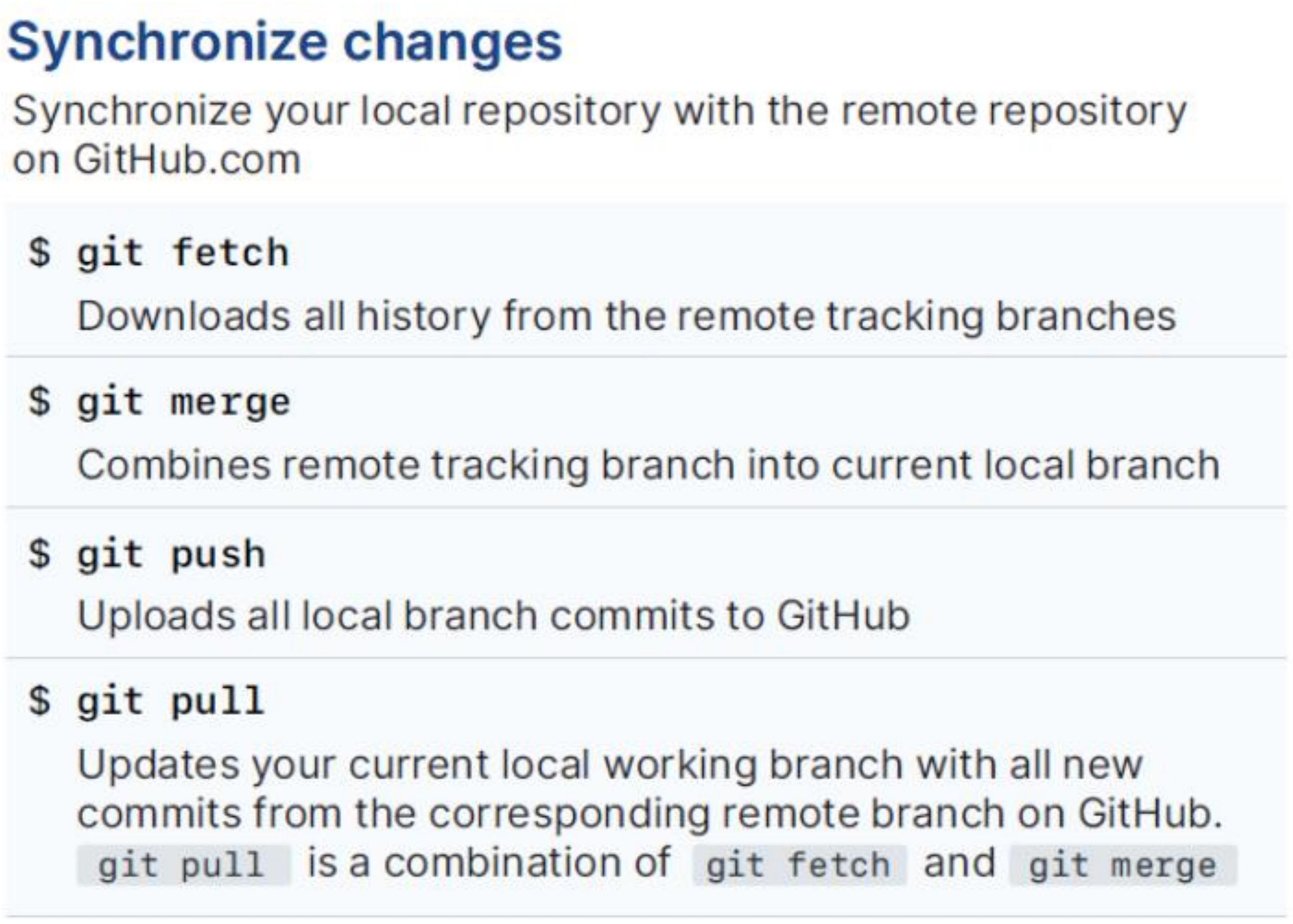
Outputs metadata and content changes of the specified commit

```
$ git add [file]
```

Snapshots the file in preparation for versioning

```
$ git commit -m "[descriptive message]"
```

Records file snapshots permanently in version history



Synchronize changes

Synchronize your local repository with the remote repository on GitHub.com

```
$ git fetch
```

Downloads all history from the remote tracking branches

```
$ git merge
```

Combines remote tracking branch into current local branch

```
$ git push
```

Uploads all local branch commits to GitHub

```
$ git pull
```

Updates your current local working branch with all new commits from the corresponding remote branch on GitHub. `git pull` is a combination of `git fetch` and `git merge`

BY
ASHWANI VERMA