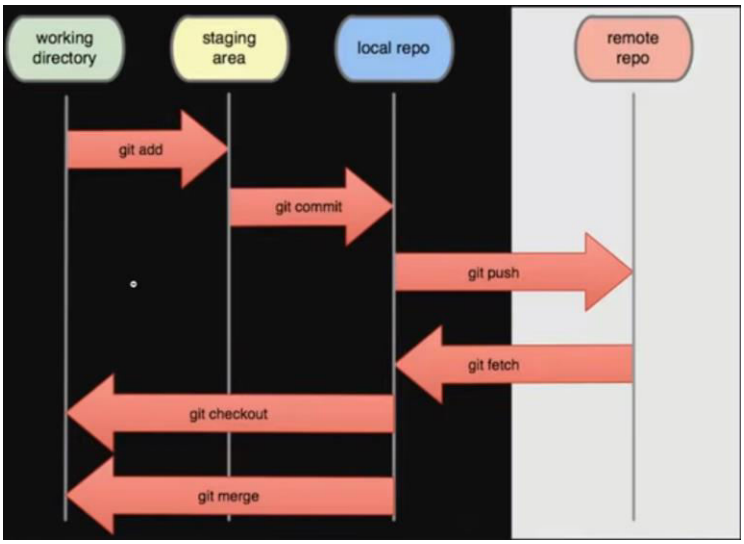
Version Control With Git

- **a Version Control** System is just software that helps you control (or manage) the different versions...of something (typically source code).
- There are two different categories of **version Control systems**: **Centralized** model and **Distributed** model
- In a **centralized model**, there's one powerful central computer that hosts the project, every interaction must go through this central computer.
- In a **distributed model**, there's no central repository of information, each developer has a complete copy of the project on their computer.
- **Git** is a distributed version control system
- **Git** is a version control tool while **Github** is a service that hosts git projects.
- the main point of a **version control system** is to help you maintain a detailed history of the project as well as the ability to work on different versions of it. Having a detailed history of a project is important because it lets you see the progress of the project over time. If needed, you can also jump back to any point in the project to recover data or files.



First Time Git Configuration

Before you can start using Git, you need to configure it. Run each of the following lines on the command line to make sure everything is set up.

```
# sets up Git with your name
git config --global user.name "<Your-Full-Name>"

# sets up Git with your email
git config --global user.email "<your-email-address>"

# makes sure that Git output is colored
git config --global color.ui auto

# displays the original state in a conflict
git config --global merge.conflictstyle diff3

git config --list
```

- **$ git config –global --unset user.name**
  - to unset a configuration
- **$git config –global --edit**
  - to open configuration file in your editor
- **$ git config --global init.defaultBranch <name>**
  - To configure the initial branch name to use in all of your new repositories

- **$ git config –global alias.’’alias-name” “Command-to-be-aliased”**
    - Make and alias for an existing command ex: git config –global alias.cm “commit -m”
    - Search web for git alias list to get some predefined aliases

Git & Code Editor

The last step of configuration is to get Git working with your code editor. Below are three of the most popular code editors. If you use a different editor, then do a quick search on Google for "associate X text editor with Git" (replace the X with the name of your code editor).

Atom Editor Setup

```
git config --global core.editor "atom --wait"
```

Sublime Text Setup

```
git config --global core.editor "'C:/Program Files/Sublime Text 2/sublime_text.exe' -n -w"
```

VSCode Setup

```
git config --global core.editor "code --wait"
```

Required Commands

- **ls** :used to list files and directories
- **mkdir** : used to create a new directory
- **cd** : used to change directories
- **rm** : used to remove files
- **rm** -r : used to remove directories
- **pwd** : which stands for “print working directory”
- **start** . : opens the current directory
- **mv “old-name” “new-name”** : rename files
- **cd ..** : to move to parent directory
- **touch :** to create a new file

Create Course Directories

If you're a copy/paster like me, just run this command on the terminal - `mkdir -p udacity-git-course/new-git-project && cd $_` (Before running this command, make sure you `cd` to where you want these files stored. For example, if you want the files stored on your Desktop, then make sure you `cd` to the Desktop before running the command.)

Create A Repo From Scratch

- To create a new repository with Git, we'll use the **git init** command.
- The **init** subcommand is short for "initialize", which is helpful because it's the command that will do all of the initial setup of a repository.
- **$ git init**

Git Init's Effect

- Running the **git init** command sets up all of the necessary files and directories that Git will use to keep track of everything.
- All of these files are stored in a directory called **.git** (notice the . at the beginning - that means it'll be a hidden directory on Mac/Linux).
- This .git directory is the "repo"! This is where **git** records all of the commits and keeps track of everything!
- **WARNING**: Don't directly edit any files inside the .git directory. This is the heart of the repository. If you change file names and/or file content, **git** will probably lose track of the files that you're keeping in the repo, and you could lose a lot of work! It's okay to look at those files though, but don't edit or delete them.
- running **git init** multiple times doesn't cause any problems since it just re-initializes the Git directory.

.Git Directory Contents

Here's a brief synopsis on each of the items in the .git directory:

- **config file** - where all project specific configuration settings are stored.
  - Git looks for configuration values in the configuration file in the Git directory (.git/config) of whatever repository you're currently using. These values are specific to that single repository.
- **description file** - this file is only used by the GitWeb program, so we can ignore it
- **hooks directory** - this is where we could place client-side or server-side scripts that we can use to hook into Git's different lifecycle events
- **info directory** - contains the global excludes file
- **objects directory** - this directory will store all of the commits we make
- **refs directory** - this directory holds pointers to commits (basically the "branches" and "tags")

Clone An Existing Repo

- what is **cloning?**
  - to create an identical copy of an existing repository.
- You pass a path (usually a URL) of the Git repository you want to clone to the **git clone** command.

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

This command:

- takes the path to an existing repository
- by default, will create a directory with the same name as the repository that's being cloned
- can be given a second argument that will be used as the name of the directory
- will create the new repository inside of the current working directory

Git Status

- **$ git status**
- To figure out what's going on with a repository, we use the **git status** command. Knowing the status of a Git repository is extremely important
- The **git status** is our key to the mind of Git. It will tell us what Git is thinking and the state of our repository as Git sees it. When you're first starting out, you should be using the **git status** command all of the time! Seriously.
- You should get into the habit of running it after any other command. This will help you learn how **Git** works and it'll help you from making (possibly) incorrect assumptions about the state of your files/repository.
- **$ git restore <file>...** to discard changes in working directory
- **$ git restore --staged <file>..."** to unstage file from staging index

Git Log Command

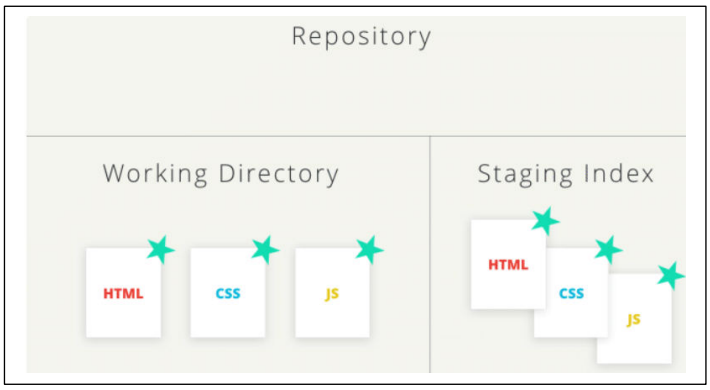
- **$ git log**
- The git log command is used to display all of the commits of a repository.
- By default, this command displays:
  - the SHA
  - the author
  - the date
  - and the message
- I stress the "By default" part of what Git displays because the git log command can display a lot more information than just this.
- Navigating The Log
  - 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**
  - --**oneline** flag is used to alter how git log displays information:
    - lists one commit per line
    - shows the first 7 characters of the commit's SHA
    - shows the commit's message
- **$git log –stat**
  - **--stat** flag is used to display the files that have been changed in the commit
    - displays the file(s) that have been modified
    - displays the number of lines that have been added/removed
    - displays a summary line with the total number of modified files and lines that have been added/removed
- **$git log -p**
  - **–patch** or **-p** flag is used to display the actual changes made to a file.
    - 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
- you can also combine flags like **$git log -p –stat**
- **$git log -p -w**
  - **-w** will show the patch information but will not highlight lines where only whitespace changes have occurred.
- **$ git log -p** fdf5493
  - By supplying a **SHA**, **the git log -p** command will start at that commit! No need to scroll through everything! Keep in mind that it will also show all of the commits that were made prior to the supplied SHA.
- **$ git log --oneline --graph --all**
  - We use this command to show all branches and commits at once
  - **--graph** flag adds the bullets and lines to the leftmost part of the output. This shows the actual branching that's happening.
  - **--all** flag is what displays all of the branches in the repository.

Git show Command

- **$ git show** `fdf5493`
  - the **git show** command will show only one commit.
  - The output of the **git show** command is exactly the same as the **git log -p** command.
  - --**stat** to show the how many files were changed and the number of lines that were added/removed
  - **-w** to ignore changes to whitespace
  - **-p** (default) , but if --**stat** is used, the patch won't display, so pass **-p** to add it again

Staging Files

Git add Command



- **$ git add <file1> <file2> … <fileN>**
  - is used to move files from the Working Directory to the Staging Index
  - Running the **git add** command produces no output (as long as there wasn't an error).
  - the **period .** can be used in place of a list of files to tell Git to add the current directory (and all nested files)
- **$ git rm --cached <file>**
  - is used to move files from the staging index back to the working directory

Git Commit

- **$ git commit**
  - The **git commit** command takes files from the Staging Index and saves them in the repository.
  - This will open the code editor to commit a message in the first line of code
  - **$ git commit -m "message"** , will pass a message without opening the code editor
- What To Include In A Commit?
  - The goal is that each commit has a single focus.
  - Each commit should record a single-unit change. Now this can be a bit subjective, but each commit should make a change to just one aspect of the project.
  - Conversely, a commit shouldn't include unrelated changes
- What makes a good **commit** message?
  - do keep the message short (less than 60-ish characters)
  - do explain what the commit does (not how or why!)
  - do not explain why the changes are made
  - do not explain how the changes are made (that's what git log -p is for!)
  - do not use the word "and", if you have to use "and", your commit message is probably doing too many changes - break the changes into separate commits
- If you need to explain **why** a commit needs to be made, After the message, leave a blank line, and then type out the body or explanation including details about why the commit is needed
- Udacity style commit message guide https://udacity.github.io/git-styleguide/
- **$ git commit --amend**
  - you can use the **–amend** flag to edit the most-recent commit
  - it can let you provide a new commit message instead of the last commit message if the working directory is clean
  - also, it can let you include files or changes to files you might have forgotten to include.
- **$ git revert <SHA-of-commit-to-revert>**

    - the **git revert** command is used to reverse a previously made commit
    - This command:
        - will undo the changes that were made by the provided commit
        - creates a new commit to record the change
- **$git reset <reference-to-commit>**
    - The **git reset** command is used to reset (erase) commits
    - **It can be used to:**
        - move the HEAD and current branch pointer to the referenced commit
        - erase commits
        - move committed changes to the staging index
        - unstage committed changes
    - **--mixed** flag, **git reset --mixed HEAD^**
        - will take the changes made in the parent commit and move them to the working directory.
    - **--soft flag, git reset --soft HEAD^**
        - will take the changes made in the parent commit and move them directly to the Staging Index.
    - **--hard Flag, git reset --hard HEAD^**
        - will take the changes made in the parent commit and erases them.
- A good approach when resetting is to do a backup branch on the most recent commit so that you can get back to the commits if you make a mistake.
    - **$ git branch backup**
    - To go back
        - **$ git checkout -- index.html**
        - **$ git merge backup**
- **$ git reflog**
    - Git does keep track of everything for about 30 days before it completely erases anything.

Relative Commit References

- **^** – indicates the parent commit
- **~** – indicates the first parent commit
- the parent commit – the following indicate the parent commit of the current commit
    - HEAD^
    - HEAD~
    - HEAD~1
- the grandparent commit – the following indicate the grandparent commit of the current commit
    - HEAD^^
    - HEAD~2
- The main difference between the **^** and the **~** is when a commit is created from a **merge**. A **merge** commit has two parents. With a merge commit, the **^** reference is used to indicate the first parent of the commit while **^2** indicates the second parent. The first parent is the branch you were on when you ran git merge while the second parent is the branch that was merged in.

Git Diff

- **$ git diff**
    - the **git diff** command is used to see changes that have been made but haven't been committed, yet.
    - 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 ignore

- the **.gitignore** file is used to tell Git about the files that Git should not track.
- This file should be placed in the same directory that the **.git** directory is in.
- Globbing lets you use special characters to match patterns/characters. In the .**gitignore** file:
  - blank lines can be used for spacing
  - **#** - marks line as a comment
  - ***** - matches 0 or more characters
  - **?** - matches 1 character
  - **[abc]** - matches a, b, or c
  - ****** - matches nested directories - a/**/z matches
    - a/z
    - a/b/z
    - a/b/c/z

Git Stach

- **$ git stash**
  - Stash working directory and staging index
- **$ git stash save "message"**
  - Add message when you stash
- **$ git stash pop**
  - Pop the last stashed files to the working directory or staging index and delete it from the stash
- **$ git stash pop stash@{'id'}**
  - Pop a specific stash
- **$ git stash list**
  - List of current Stashes
- **$ git stash apply**
  - Pop the last stashed files to the working directory or staging index but doesn't delete it from the stash
- **$ git stash drop**
  - Delete the last stashed files
- **$ git stash show**
  - Returns what's inside the last stashed files
- **$ git stash clear**
  - Delete all the stashed files
- **$ git stash branch <branch-name>**
  - create a branch from the latest stash and apply its changes to it.

Tagging

- **$ git Tag -a** v1.0 **-m "message"**
- 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.
- This command will:
  - add a tag to the most recent commit
  - add a tag to a specific commit if a **SHA** is passed
- **-a :** This flag tells Git to create an **annotated** flag.
  - annotated tags are recommended because they include a lot of extra information such as:
    - the person who made the tag

    - the date the tag was made
    - a message for the tag
- **$ git tag** will display all tags that are in the repository.
- **$ git tag -d v1.0** : to delete a tag

Branching

Git branch

- **$ git branch**
- The **git branch** command is used to interact with Git's branches
- **$ git branch :** list all branch names in the repository
- **$ git branch ”name”** create new branch
- **$ git branch -d “name”** delete a branch
    - You can’t delete a branch that you are currently on, you need to switch to another branch first
    - Git won't let you delete a branch if it has commits on it that aren't on any other branch (meaning the commits are unique to the branch that's about to be deleted).
    - Use **git branch -D ‘’name’’** If you want to force deletion
- **$ git branch “name’’ “sha” :** create a branch and have it point to the commit with **SHA**
- **$ git branch -m “NewNameBranch”** : rename the just created branch

Git checkout

- **$git checkout “branch-name”**
- The **git checkout** is used to switch between branches
- Running this command will:
    - remove all files and directories from the Working Directory that Git is tracking (files that Git tracks are stored in the repository, so nothing is lost)
    - go into the repository and pull out all of the files and directories of the commit that the branch points to
- **$ git checkout -b “footer” “master”**
    - create a new footer branch and have this footer branch start at the same location as the master branch, switch to this branch.
- **$ git checkout -- ‘’file name”**
    - remove the edits you did in the working directory on this file if it is in staging index or committed
- **$ git restore --staged <file>..."**
    - to Remove file from staging index
- **$ git clean -n**
    - List of files you can delete in the working directory
- **$ git clean -p**
    - Delete the files from the working directory

Merging

- **$ git merge <name-of-branch-to-merge-in>**
    - The **git merge** command is used to combine Git branches
- When a merge happens, Git will:
    - look at the branches that it's going to merge
    - look back along the branch's history to find a single commit that both branches have in their commit history
    - combine the lines of code that were changed on the separate branches together
    - makes a commit to record the merge

- **Fast-forward Merge** – the branch being merged in must be ahead of the checked-out branch. The checked-out branch's pointer will just be moved forward to point to the same commit as the other branch.
- **the regular type of merge** :
  - two divergent branches are combined
  - a merge commit is created
- How to solve a merge conflict?
  - First, do **git status** to figure out where are the merge conflicts
  - Open the file that have the merge conflicts in your editor
  - Merge Conflict Indicators Explanation:
    - **<<<<<<< HEAD** everything below this line (until the next indicator) shows you what's on the current branch.
    - **||||||| merged common ancestors** everything below this line (until the next indicator) shows you what the original lines were.
    - **=======** is the end of the original lines, everything that follows (until the next indicator) is what's on the branch that's being merged in.
    - **>>>>>>>** is the ending indicator of what's on the branch that's being merged in.
  - Then you have to choose which lines to keep, and remove all lines with indicators
  - Then save, add file to staging index, then commit it.
- Be careful that a file might have merge conflicts in multiple parts of the file, so make sure you check the entire file for merge conflict indicators - a **quick search** for **<<<** should help you locate all of them.

Dealing with Remote repository

- **$ git push "RemoteRepoName" "branch"**
  - This command pushes your commits to the remote repository
- **$ git push "RemoteRepoName" "branch" --force**
  - This command pushes your commits to the remote repository by force
- **$ git remote -v**
  - This command return the link to the origin Remote Repository to Fetch and Pull
- **$ git pull "RemoteRepoName"**
  - This commands pulls your remote repository to the local repository; it does two commands Fetch and Merge

Generate and Test Github Public Key

- **ssh --keygen -t rsa -b 4096 -C "email"**
  - **--keygen** key generator
  - **-t** type of algorithm, **rsa** one of the algorithms
  - **-b** no. of bits**, 4096**
  - This command generates public/private key
- **Cat ~/.ssh/id_rsa.pub**
  - Returns the content of public key file
- Ssh -T git@github.com
  - To authenticate your key on Github

Create a Repository and upload it to Github

- **git init** => initialize the repository in the current file
- **git add README.md** => add the readme file to the staging index
- **git commit -m "first commit"** => commit the changes
- **git branch -M main** => rename the current branch to main
- **git remote add origin "link of ssh or https"** => link the repository with remote repository

- **git push -u origin main** => Pull(-u) and push changes to the remote repository

Pull Request

- you can Fork an Existing Repository from Github
- add and commit your changes to this repository
- then submit a pull request, and wait for your commits to be merged to the main repository