**Git Notes**

**Git Commands:**

* **$ git --version** – *displays the version of Git that is installed on the computer.*
* **$ git config --global user.name ‘Your Name’** *– sets your name. This is important because every Git commit uses this information.*
* **$ git config --global user.email ‘Your E-Mail’** – *sets your email. This is important because every Git commit uses this information.*
* **$ git touch index.html** – *creates a html file called “index”.*
* **$ git init** – *initializes a Git repository.*
* **$ git add <file> -** *adds the specified file to the staging area.*
* **$ git add \*.html** – *adds every html file to the staging area.*
* **$ git add .** – *adds every file to the staging area.*
* **$ git rm** **--cached index.html** – *removes “index.html” from the staging area.*
* **$ git status** *– checks what you have in the staging area.*
* **$ touch .gitignore** – *creates a text file in which you put all the files you wish to not be able to* *go to the staging area. (eg. “log.txt”; eg. for directories “/dir2”).*
* **$ git commit** - *commits files. Sends everything from the staging area to the local repository.*
* **$ git commit –m ‘You Comment’** *– faster way to commit files.*
* **$ git push** – *pushes the local repository to the remote one.*
* **$ git clone** – *will copy a remote repository into the current folder you are in.*
* **$ git branch “name of branch”** – *will create a branch with the specified name.*
* **$ git checkout “name of branch”** – *moves the user to a branch with the specified name.*
* **$ git merge “name of branch”** – *merges the master branch with the specified branch.*
* **$ git remote add “name of repository” “link to Git repository”** – *specifies the given Git repository as the remote repository.*
* **$ git remote** – *lists all the remote repositories.*
* **$ git push –u “name of remote repository” “name of branch”** – *pushes the files from the local repository to the remote one.*
* **$ git clone “link to Git repository”** – *creates a copy of the remote repository in the selected folder.*
* **$ git pull**– *if there are any changes to the repository (liked new files added) it puts them in your working directory.*

**Git Repositories**

**$ git init** – the *“$ git init”* command creates a new Git repository. It can be used to convert an existing, unversioned project to a Git repository or initialize a new, empty repository. Most Git commands are not available outside of an initialized repository.

Executing *“$ git init”* creates a .git subdirectory in the current working directory, which contains all of the necessary Git metadata for the new repository. A HEAD file is also created which points to the currently checked out commit.

The .git folder is contained within the working directory (the folder) and it is the local directory.

**$ git add** – adds a change in the working directory to the staging area. It tells Git that you want to include updates to a particular file in the next commit. However, Git add doesn’t really affect the repository in any significant way – changes are not actually recorded until you run *“$ git commit”*.

Developing a project resolves around the basic edit/stage/commit pattern. First you edit your files in the working directory. When you’re ready to save a copy of the current stage of the project, you stage changes with *“$ git add”.* After you are happy with the staged snapshot, you commit it to the project history with *“$git commit”.*

Basically the staging part is optional (because there is a command that allows you to commit all files without staging first), BUT the staging area gives you the control to make smaller commits.

In other words, instead of committing all of the changes you’ve made since the last commit, the stage lets you group related changes into highly focused snapshots before actually committing it to the project history. This means you can make all sorts of edits to unrelated files, then go back and split them up into logical commits by adding related changes to the stage and commit them piece-by-piece. As in any revision control system, it’s important to create atomic commits so that it’s easy to track down bugs and revert changes with minimal impact on the rest of the project.

Let’s say you have a big project. You have 10 files that you have made changes on. You don’t want to commit all of them at once, because if there is a bug, it is harder to track it down. So you want to make commits in smaller portions, called snapshots. *“$ git add*” and the staging area are there to help you. You group the files by logic, add them to the staging area, and then commit. You repeat the process for every group of files.

**$ git commit** – instead of making a change and committing it directly to the central repo, Git developers have the opportunity to accumulate commits in their local repo. This lets developers work in an isolated environment, deferring until they are at a convenient point to merge with other users.

**$ git pull** – the command is used to fetch and download content from a remote repository and immediately update the local repository to match that content. You can think of “*$ git pull*” as an easy way to synchronize your local repository with upstream changes.

The *”$ git pull”* command is actually a combination of two other commands, *“$git fetch”* followed by *“$ git merge”*.

The *“$ git pull”* script is meant as a convenience method for invoking *“$ git fetch*” followed by *“$ git merge”.* The first extra argument to *“$ git pull”* tells it which remote directory to give to the *“$ git fetch*” operation, for example:

$ git pull origin

means to fetch from origin. If you leave this out, Git uses the current branch’s remote:

$ git branch

\* master

$ git config --get branch.master.remote

origin

The second (additional) argument to *“$ git pull”* tells it which branch or branches to merge in. This is the name of the branch as found on the remote. For instance, if you want to fetch from origin to pick up new commits added to their ‘login’ branch, and merge them with your local master branch:

$ git pull origin login

If *“$ git pull”* says that there is not tracking information for the current branch, you need to set it up so that your branch tracks the GitHub branch as an upstream:

$ git branch --set-upstream-to-origin/master “Name of Branch”

$ git pull

*“$ git pull”* s job is to fetch new commits and merge them into the current branch. If the current branch is not outdated compared to the one you pull from, that is that no new pushes have been made by other users, ‘pull’ will say ‘Already-up-to-date”. Even if you have local changes in your working directory, *“$ git pull*” doesn’t update.



**Branches in Git**

**Why do we have branching?**

For example, if you work with a big project, lots of files and lots of folders, then you will know that because there are so many files and so many things can go wrong you don’t really want to have to keep track of what has changed and what hasn’t; and perhaps you have one copy of your source code in which everything is working and you don’t want to mess with that, because making changes to it risks it breaking.

So what you would normally do is make a copy of what is already working and you would just work on the duplicate.

Well Git has a very effective way of dealing with this and it is called branching.

**Example way of structuring a project:**

MASTER BRANCH

FEATURE C

FEATURE B

FEATURE A

The **master branch** -> holds the stable version of the product.

The **develop branch** -> is created in the image of the **master branch.** It is regarded as the “staging area” in which future features could be added.

The **feature branches** -> they don’t need to be three, bur for this example let’s say that we want three new features to be added to our product. Each feature is worked on separately, in a different branch, and worked on by a different developer.

The workflow is as follows -> the **develop branch** is created as a mirror image of the **master branch**. Each **feature branch** gets the same copy of the **develop branch.** Once the feature is done, its branch is *merged* with the **develop branch**. Once all features are done, and a stable version is completed, the **develop branch** is merged with the **master branch**.

Side note: Any **feature branch** could be deleted after merging with the **develop branch**.