Git init : When you use git init and initialize a repo, you’re officially making a Git project and can now make use of all the git commands in the world! If you’re not inside of a git repo, you won’t be able to use git commands and the terminal will complain to you about it.

Git Project:

A Git project can be thought of as having three parts:

1. A *Working Directory*: where you’ll be doing all the work: creating, editing, deleting and organizing files
2. A *Staging Area*: where you’ll list changes you make to the working directory
3. A *Repository*: where Git permanently stores those changes as different *versions* of the project

The Git workflow consists of editing files in the working directory, adding files to the staging area (git add), and saving changes to a Git repository (git commit).

As you work you will be making changes to your content, You can check the status of those changes with git status.

**Untracked** means that Git sees the file but has not started tracking changes yet.

Git add: Git start tracking the project

Git diff *filename*: showing the difference between the working directory and the staging area

Git commit –m : submitting the final change to git with a commit message (-m)

Git log: viewed commits history in the chronologically order –

In the output, notice:

* A 40-character code, called a SHA, that uniquely identifies the commit. This appears in orange text.
* The commit author (you!)
* The date and time of the commit
* The commit message

Summary:

* git init creates a new Git repository
* git status inspects the contents of the working directory and staging area
* git add adds files from the working directory to the staging area
* git diff shows the difference between the working directory and the staging area
* git commit permanently stores file changes from the staging area in the repository
* git log shows a list of all previous commits

In Git, the commit you are currently on is known as the HEAD commit. In many cases, the most recently made commit is the HEAD commit.

Git show HEAD : the output of this command display everything the git log command displays for the HEAD commit, plus all the file changes that were committed.

**git checkout**

What if you decide to change the ghost’s line in the working directory, but then decide you wanted to discard that change?

You could rewrite the line how it was originally, but what if you forgot the exact wording? The command

git checkout HEAD filename

will restore the file in your working directory to look exactly as it did when you last made a commit.

Here, filename again is the actual name of the file. If the file is named **changes.txt**, the command would be

git checkout HEAD changes.txt

**Git reset HEAD filename:**  resets the file in the staging area to be the same as the HEAD commit. It does not discard the file changes from the working directory, it just removes them from the staging area.

* git checkout HEAD filename: Discards changes in the working directory.
* git reset HEAD filename: Unstages file changes in the staging area.
* git reset commit\_SHA: Resets to a previous commit in your commit history.

# git branch

Up to this point, you’ve worked in a single Git branch called master. Git allows us to create branches to experiment with versions of a project. Imagine you want to create version of a story with a happy ending. You can create a new branch and make the happy ending changes to that branch only. It will have no effect on the master branch until you’re ready to merge the happy ending to the master branch.

In this lesson, we’ll be using Git branching to develop multiple versions of a resumé.

You can use the command below to answer the question: “which branch am I on?”

git branch

To create new branch: git branch new\_branch

New\_branch would be the name of the new branch you create. The name cannot have whitespace: *new-branch* and *new\_branch* are acceptable but *new branch* is not.

The master and fencing branches are identical: they share the same exact commit history. You can switch to the new branch with

git checkout branch\_name

Here, branch\_name is the name of the branch. If the branch’s name is skill

git checkout skill

Once you switch branch, be now able to make commits on the branch that have no impact on master.

You can continue your workflow, while master stays intact!

# git merge

What if you wanted include all the changes made to the fencing branch on the master branch? We can easily accomplish this by merging the branch into master with:

git merge giver\_branch\_name

For example, if I wanted to merge the skills branch to master, I would enter

git merge skills

# merge conflict I

What would happen if you made a commit on master before you merged the two branches? Furthermore, what if the commit you made on master altered the same exact text you worked on in Branch? When you switch back to master and ask Git to merge the two branches, Git doesn’t know which changes you want to keep. This is called a merge conflict.

# merge conflict II

Let’s say you decide you’d like to merge the changes from fencing into master.

Here’s where the trouble begins!

You’ve made commits on separate branches that alter the same line in conflicting ways. Now, when you try to merge fencing into master, Git will not know which version of the file to keep.

# delete branch

In Git, branches are usually a means to an end. You create them to work on a new project feature, but the end goal is to merge that feature into the master branch. After the branch has been integrated into master, it has served its purpose and can be deleted.

The command

git branch -d branch\_name

will delete the specified branch from your Git project.

Now that master contains all the file changes that were in fencing, let’s delete fencing.

* git branch: Lists all a Git project’s branches.
* git branch branch\_name: Creates a new branch.
* git checkout branch\_name: Used to switch from one branch to another.
* git merge branch\_name: Used to join file changes from one branch to another.
* git branch –d/D branch\_name: Deletes the branch specified. Capital D to delete branch that haven’t merge

# git clone

Sally has created the remote repository, **science-quizzes** in the directory **curriculum**, which teachers on the school’s shared network have access to. In order to get your own replica of **science-quizzes**, you’ll need to clone it with:

git clone remote\_location clone\_name

In this command:

* remote\_location tells Git where to go to find the remote. This could be a web address, or a filepath, such as:

/Users/teachers/Documents/some-remote

* clone\_name is the name you give to the directory in which Git will clone the repository.

# git remote -v

One thing that Git does behind the scenes when you clone remote\_location is give the remote address the name origin, ,so that you can refer to it more conveniently.

You can see a list of a Git project’s remotes with the command:

git remote -v

# git fetch

After you cloned , the recent project is changed in some way. If so, your clone will no longer be up-to-date.

An easy way to see if changes have been made to the remote and bring the changes down to your local copy is with:

git fetch

This command will not merge changes from the remote into your local repository. It brings those changes onto what’s called a remote branch (origin/master) and fetch any new changes have made to the remote.

# git merge

Even though new commits have been fetched to your local copy of the Git project, those commits are on the origin/master branch. Your local master branch has not been updated yet, so you can’t view or make changes to any of the work she has added.

In Lesson III, Git Branching we learned how to merge branches. Now we’ll use the git merge command to integrate origin/master into your local master branch. The command:

git merge origin/master

will accomplish this for us.

**Git workflow**

Now that you’ve merged origin/master into your local master branch, you’re ready to contribute some work of your own. The workflow for Git collaborations typically follows this order:

1. Fetch and merge changes from the remote
2. Create a branch to work on a new project feature
3. Develop the feature on your branch and commit your work
4. Fetch and merge from the remote again (in case new commits were made while you were working)
5. *Push* your branch up to the remote for review

Steps 1 and 4 are a safeguard against *merge conflicts*, which occur when two branches contain file changes that cannot be merged with the git merge command. Step 5 involves git push, a command you will learn in the next exercise.

# git push

Now it’s time to share our work with others.

The command:

git push origin your\_branch\_name

will push your branch up to the remote, origin, and create a new branch. From there, other can review your branch and merge your work into the master branch, making it part of the definitive project version.

* A *remote* is a Git repository that lives *outside* your Git project folder. Remotes can live on the web, on a shared network or even in a separate folder on your local computer.
* The *Git Collaborative Workflow* are steps that enable smooth project development when multiple collaborators are working on the same Git project.
* git clone: Creates a local copy of a remote.
* git remote -v: Lists a Git project’s remotes.
* git fetch: Fetches work from the remote into the local copy.
* git merge origin/master: Merges origin/master into your local branch.
* git push origin <branch\_name>: Pushes a local branch to the origin remote.