**WORKFLOW**

The word init means initialize. The command sets up all the tools Git needs to begin tracking changes made to the project.

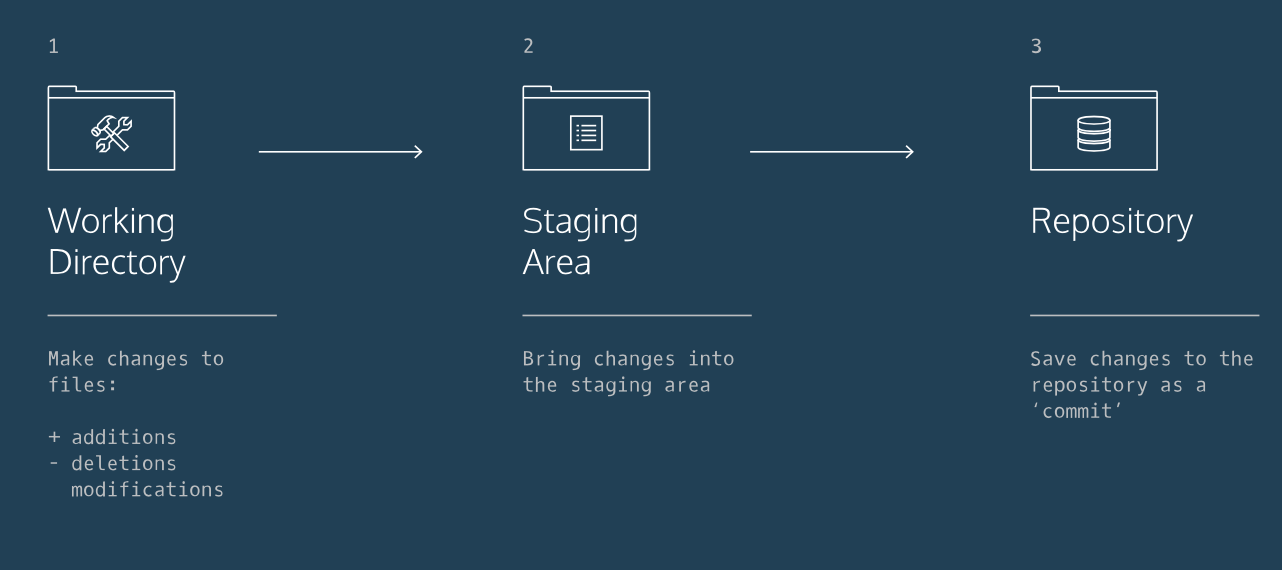
A Git project can be thought of as having three parts:

A Working Directory: where you'll be doing all the work: creating, editing, deleting and organizing files

A Staging Area: where you'll list changes you make to the working directory

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, and saving changes to a Git repository. In Git, we save changes with a commit, which we will learn more about in this lesson.



Use Git commands to help keep track of changes made to a project:

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

However, one more bit of code is needed for a commit: the option -m followed by a message. Here's an example:

git commit -m "Complete first line of dialogue"

git log shows a list of all previous commits

-> git log --pretty=oneline

**BACKTRACK**

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.**

To see the HEAD commit, enter:

git show HEAD

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

git checkout HEAD filename

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

In Git, it's common to change many files, add those files to the staging area, and commit them to Git in a single commit.

After you change the name in both files, you could add the changed files to the staging area with:

git add filename\_1 filename\_2

Unthinkingly, you add scene-2.txt **to the staging a**rea. The file change is unrelated to the Larry/Laertes swap and you **don't want to include it in the commit**.

We can **unstage** that file from the staging area using

git reset HEAD filename

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

Git enables you to rewind to the part before you made the wrong turn and create a new destiny for the project. You can do this with:

git reset SHA

This command works by using the first 7 characters of the SHA of a previous commit.

The HEAD commit will be reassigned.

Let's take a moment to review the new commands:

git checkout HEAD filename: Discards changes in the working directory.

git reset HEAD filename: Unstages file changes in the staging area.

git reset SHA: Can be used to reset to a previous commit in your commit history.

Additionally, you learned a way to add multiple files to the staging area with a single command:

git add filename\_1 filename\_2

**BRACNCHING**

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 a new branch, use:

git branch new\_branch

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

You will be now able to make commits on the fencing branch that have no impact on master.

Use git branch to verify that you have switched branches.

In the output, notice the \* is now over the fencing branch.

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 branch\_name

In a moment, you'll merge branches. Keep in mind:

Your goal is to update master with changes you made to fencing.

fencing is the giver branch, since it provides the changes.

master is the receiver branch, since it accepts those changes.

Steps:

1. You are currently on the fencing branch. Switch over to the master branch.

2. merge the fencing branch into the master branch.

Notice the output: The merge is a "fast forward" because **Git recognizes that fencing contains the most recent commit.** **Git fast forwards master to be up to date with fencing.**

Conflict:

The merge was successful because master had not changed since we made a commit on fencing. Git knew to simply update master with changes on fencing.

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 fencing? 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.

Solution:

Keeping the side branch:

From the code editor:

Delete the content of the line as it appears in the master branch

Delete all of Git's special markings including the words HEAD and fencing. If any of Git's markings remain, for example, >>>>>>> and =======, the conflict remains.

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

summary:

Let's take a moment to review the main concepts and commands from the lesson before moving on.

Git branching allows users to experiment with different versions of a project by checking out separate branches to work on.

The following commands are useful in the Git branch workflow.

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 branch\_name: Deletes the branch specified.

**TEAMWORK:**

So far, we've learned how to work on Git as a single user. Git offers a suite of collaboration tools to make working with others on a project easier.

Imagine that you're a science teacher, developing some quizzes with Sally, another teacher in the school. You are using Git to manage the project.

In order to collaborate, you and Sally need:

A complete replica of the project on your own computers

A way to keep track of and review each other's work

Access to a definitive project version

You can accomplish all of this by using **remotes**. A remote is a shared Git repository that allows multiple collaborators to work on the same Git project from different locations. Collaborators work on the project independently, and merge changes together when they are ready to do so.

Enter git remote -v to list the remotes.

Notice the output:

origin /home/ccuser/workspace/curriculum/science-quizzes (fetch)

origin /home/ccuser/workspace/curriculum/science-quizzes (push)

Git lists the name of the remote, origin, as well as its location.

Git automatically names this remote origin, because it refers to the remote repository of origin. However, it is possible to safely change its name.

The remote is listed twice: once for (fetch) and once for (push). We'll learn about these later in the lesson.

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.

Even though Sally's 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 braches. 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.

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.

Now it's time to share our work with Sally.

The command:

git push origin your\_branch\_name

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

Congratulations, you now know enough to start collaborating on Git projects! Let's review.

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.

We also learned the following commands

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.

Git projects are usually managed on Github, a website that hosts Git projects for millions of users. With Github you can access your projects from anywhere in the world by using the basic workflow you learned here.