**What’s a repository?**

A repository, or **Git project**, encompasses the entire collection of files and folders associated with a project, along with each file’s revision history. The file history appears as snapshots in time called commits, and the commits exist as a linked list relationship, and can be organized into multiple lines of development called branches. Because Git is a DVCS, repositories are self-contained units and anyone who owns a copy of the repository can access the entire codebase and its history. Using the command line or other ease-of-use interfaces, a git repository also allows for: interaction with the history, cloning, creating branches, committing, merging, comparing changes across versions of code, and more.

**Basic Git commands**

To use Git, developers use specific commands to copy, create, change, and combine code. These commands can be executed directly from the command line or by using an application like GitHub Desktop or Git Kraken. Here are some common commands for using Git:

* **git init** initializes a brand new Git repository and begins tracking an existing directory. It adds a hidden subfolder within the existing directory that houses the internal data structure required for version control.
* **git clone** creates a local copy of a project that already exists remotely. The clone includes all the project’s files, history, and branches.
* **git add** stages a change. Git tracks changes to a developer’s codebase, but it’s necessary to stage and take a snapshot of the changes to include them in the project’s history. This command performs staging, the first part of that two-step process. Any changes that are staged will become a part of the next snapshot and a part of the project’s history. Staging and committing separately gives developers complete control over the history of their project without changing how they code and work.
* **git commit** saves the snapshot to the project history and completes the change-tracking process. In short, a commit functions like taking a photo. Anything that’s been staged with git add will become a part of the snapshot with git commit.
* **git status** shows the status of changes as untracked, modified, or staged.
* **git branch** shows the branches being worked on locally.
* **git merge** merges lines of development together. This command is typically used to combine changes made on two distinct branches. For example, a developer would merge when they want to combine changes from a feature branch into the master branch for deployment.
* **git pull** updates the local line of development with updates from its remote counterpart. Developers use this command if a teammate has made commits to a branch on a remote, and they would like to reflect those changes in their local environment.
* **git push** updates the remote repository with any commits made locally to a branch.

**How GitHub works**

GitHub builds collaboration directly into the development process. Work is organized into repositories, where developers can outline requirements or direction and set expectations for team members. Then, using the GitHub flow, developers simply create a branch to work on updates, commit changes to save them, open a pull request to propose and discuss changes, and merge pull requests once everyone is on the same page.



**The GitHub flow**

The GitHub flow is a lightweight, branch-based workflow built around core Git commands used by teams around the globe—including ours.

The GitHub flow has six steps, each with distinct benefits when implemented:

**Step 1 : Create a branch:** Topic branches created from the canonical deployment branch (usually master) allow teams to contribute to many parallel efforts. Short-lived topic branches, in particular, keep teams focused and results in quick ships.

**Step 2 : Add commits:** Snapshots of development efforts within a branch create safe, revertible points in the project’s history.

**Step 3 : Open a pull request:** Pull requests publicize a project’s ongoing efforts and set the tone for a transparent development process.

**Step 4 : Discuss and review code:** Teams participate in code reviews by commenting, testing, and reviewing open pull requests. Code review is at the core of an open and participatory culture.

**Step 5 : Merge:** Upon clicking merge, GitHub automatically performs the equivalent of a local ‘git merge’ operation. GitHub also keeps the entire branch development history on the merged pull request.

**Step 6 : Deploy:** Teams can choose the best release cycles or incorporate continuous integration tools and operate with the assurance that code on the deployment branch has gone through a robust workflow.

**Models for collaborative development**

There are two primary ways people collaborate on GitHub:

* Shared repository
* Fork and Pull

With a shared repository, individuals and teams are explicitly designated as contributors with read, write, or administrator access. This simple permission structure, combined with features like protected branches and Marketplace, helps teams progress quickly when they adopt GitHub.

For an open source project, or for projects to which anyone can contribute, managing individual permissions can be challenging, but a fork and pull model allows anyone who can view the project to contribute. A fork is a copy of a project under an developer’s personal account. Every developer has full control of their fork and is free to implement a fix or new feature. Work completed in forks is either kept separate, or is surfaced back to the original project via a pull request. There, maintainers can review the suggested changes before they’re merged. See the [**Forking Projects Guide**](https://help.github.com/en/articles/about-pull-requests)for more information.

For more reference Please visit the following [link!](https://www.edureka.co/blog/git-tutorial/) and watch this [**video**](https://www.youtube.com/watch?v=SWYqp7iY_Tc) for an interactive version of explanation of git and git work flow.

**Fork vs Clone**

Fork  
  
- A fork is a copy of a repository that allows you to freely experiment with changes without affecting the original project.

- A forked repository differs from a clone in that a connection exists between your fork and the original repository itself.

- Your fork acts as a bridge between the original repository and your personal copy where you can contribute back to the original project using [Pull Requests](https://help.github.com/articles/about-pull-requests).

-Forking a project is as easy as clicking the **Fork** button in the header of a repository  
  
-Once the process is complete, you'll be taken right to your the forked copy of the project so you can start collaborating!

Clone

-When you create a new repository on GitHub, Clone exists as a remote location where your project is stored. You can clone your repository to create a local copy on your computer so that you can sync between both the local and remote locations of the project.

-Unlike forking, you won't be able to pull down changes from the original repository you cloned from

- If the project is owned by someone else you won't be able to contribute back to it unless you are specifically invited as a collaborator.   
   
- Cloning is ideal for instances when you need a way to quickly get your own copy of a repository where you may not be contributing to the original project.

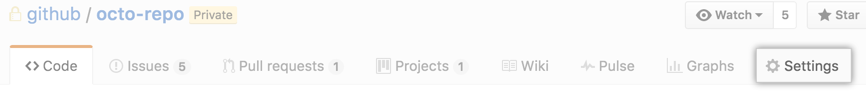
-To clone a repository, head over to the main page of a project and click the Clone or download button to get the the repository's HTTPS or SSH URL. Then, you can perform the clone using the `git clone` command in your command line interface of choice.

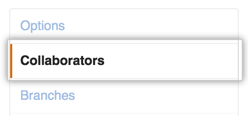
**How to Add Collaborators?**

Step 1 : Ask for the username of the person you're inviting as a collaborator

Step 2 : On GitHub Enterprise, navigate to the main page of the repository

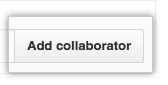
Step 3 : Under your repository name, click  ‘**Settings**’



Step 4 : In the left sidebar, click ‘**Collaborators**’   


Step 5 : Under "Collaborators", start typing the collaborator's username

Step 6 : Click **Add collaborator**



**Creating a Merge conflict**

Merge conflicts in git happen, when two branches were changed on the same line or in the same content of a file before a merge. If you just extend a file or append something, git usually just figures it out by itself.

When git doesn’t know which version of a file to keep, it will insert some lines into the respective source code file and it will end up looking somewhat like this:

A screenshot of a cell phone

Description automatically generated

**Resolving a merge conflict**

Usually there are only two scenarios when a merge conflict arises, you will either be putting changes from your branch into master, or you’ll be putting changes from master into your branch.  
The trick to avoiding merge conflicts is each time you have to put code from one branch into another, you just gotta figure out which of these two scenarios you are in.

## **Merge my-branch onto master**

Say you are on **master** (and you’ve already run git pull), and you run…

(git merge my-branch)Git is putting the changes from your branch on top of master.

## **Rebase master onto my-branch**

Say you are on **my-branch**, and you run…

(git rebase master)

Git is taking off the changes you made on my-branch, then it is putting the latest changes from master, and finally it puts your changes from my-branch back on top. It’s like the second frame of the sandwich-making GIF above, when the top slice of bread is removed. After that, the meat and cheese are added, and then the top slice is put back on top of the whole pile. That’s what git rebase does.

# The steps -First, make sure to git pull the latest changes into master.

-Second, git rebase master into your branch. This will likely be the only time you get a conflict, and it should only be because someone else has changed the same code as you since the last time you rebased master, so it’s usually very easy to fix.

Finally, whenever you want to merge your branch with master, make sure you do the first two steps right before, and then finally from master you can run git merge my-branch. Since your branch is totally up-to-date with master, Git will simply put your code on top of the master pile.

**How to execute a pull request in Git?**

The simplest way is to:

1. Find a project you want to contribute to
2. Fork it
3. Clone it to your local system
4. Make a new branch
5. Make your changes
6. Push it back to your repo
7. Click the **Compare & pull request** button
8. Click **Create pull request** to open a new pull request