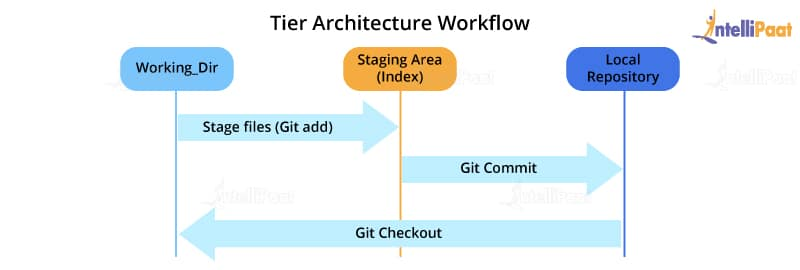
**GIT & GITHUB**

* **Git** is a **distributed version control system (DVCS)** for tracking changes to files.
  + But what does that mean?
  + **Git** is an open-source VCS, which is not file-based, unlike other systems. Rather, it stores information as **snapshots**. Being a VCS, helps coders revert to their previous code when they hit a rollback in the newer version, without affecting the original source code.
  + On the other hand, what makes it different from other VCS is the way it sees data, which is more like a series of snapshots. ***It basically clicks a picture of how all your files look at the moment and saves the changes made to them over time.***

## **Features of Git**

* **Works on a distributed system:** A distributed system is one that allows collaborators to access the central repository using a VCS, even from the remotest corner of the world. As Git maintains a snapshot every time a user pulls a file, the risk of data loss due to system failure or lack of Internet connection is mitigated. Users are allowed to work on the same bit of code simultaneously without getting interfered with by others.
* **Compatible with all operating systems:** Git is compatible with almost all operating systems that are available today. Even the repositories created by other version control systems can be accessed by the Git repository.
* **Allows for non-linear development:** As users from remote parts of the world can access the Git repository, work on it, and update the project at any time they want, Git allows for development in a non-linear fashion. Git supports such a kind of development by providing its branching and merging features, and it uses specific tools for navigating through them. The projects are viewed in a tree form.
* **Branches like a tree:** While users are working on their projects, branches parallel to the main project file are created, so the original code is not affected. There is no restriction upon the number of branches created.
* **Light as a cotton ball:** One might think that making multiple copies of data from a central repository to a local one will eventually lead the system to crash due to overload. But, Git has got it covered. It compresses the data in such a way that it takes up minimal space, and whenever you need to retrieve data, the reverse technique is used. This helps save a lot of memory.
* **Fast as a flash:** Unlike other version control systems, Git is written in a language known to be the closest to the machine language, that is, C. Hence, it processes information much faster.
* **Reliable:** There will never be an issue of data loss as long as the copies of data in the central repositories are available in the local repositories of different collaborators.

## **Git Architecture**

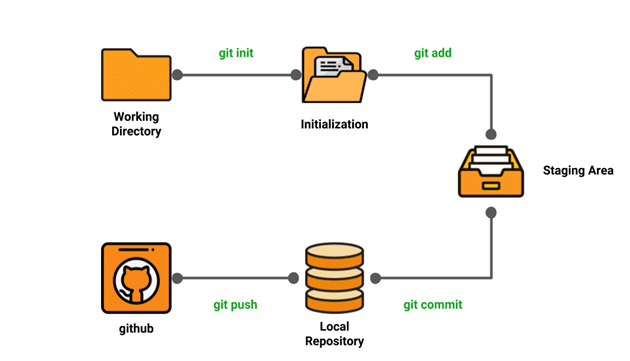


Most of the version control systems have a **two-tier architecture**. However, **Git** has a layer more, making it a **three-tier architecture**. But, why are there three layers of Git? Let’s find out.

The three layers are:

* **Working directory:** This is created when a Git project is initialized onto your local machine and allows you to edit the source code copied.
* **Staging area:** Post the edits, the code is staged in the staging area by applying the command, **git add**. This displays a preview for the next stage. In case further modifications are made in the working directory, the snapshots for these two layers will be different. However, these can be synced by using the same ‘git add’ command.
* **Local repository:** If no further edits are required to be done, then you can go ahead and apply the **git commit command**. This replicates the latest snapshots in all three stages, making them in sync with each other.

## **Git Life Cycle**



* **Local working directory:** The first stage of a Git project life cycle is the local working directory where our project resides, which may or may not be tracked.
* **Initialization**: To initialize a repository, we give the command **git init**. With this command, we will make Git aware of the project file in our repository.
* **Staging area**: Now that our source code files, data files, and configuration files are being tracked by Git, we will add the files that we want to commit to the staging area by the **git add** command. This process can also be called indexing. The index consists of files added to the staging area.
* **Commit**: Now, we will commit our files using the **git commit -m ‘our message’** command.

### **Environment Setup**

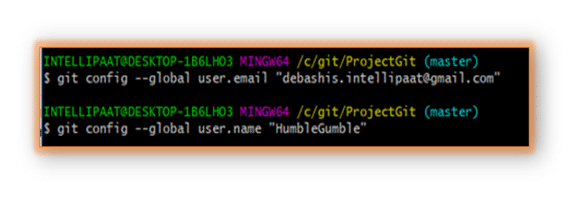
* Create a GitHub account
* Configure Git
* Create a local repository

**Creating a GitHub account:**

* Go to https://Github.com
* Create a GitHub account
* Login

**Configuring Git:**

To tell Git who we are, run the following two commands:



## **Common Git Commands**

We will divide the common Git commands into two primary categories:

* **Local (offline)** : **git init, git touch, git status, git add, git commit, and git rm**
* **Remote (online)** : **git remote, git pull, and git push**

**Steps for Creating and Pushing code to Remote Repository (GitHub)**

1. Git init
2. Git add . (Staging Area)
3. Git status (to know the file which are tracked or not)
4. Git commit -m “my Message” (Local Repository)
5. Git clone “full path” → Optional

### **Remote Commands**

**Step 1**: Go to the GitHub account

**Step 2**: Create a new repository

After Creating NEW github Repository please follow below commands :

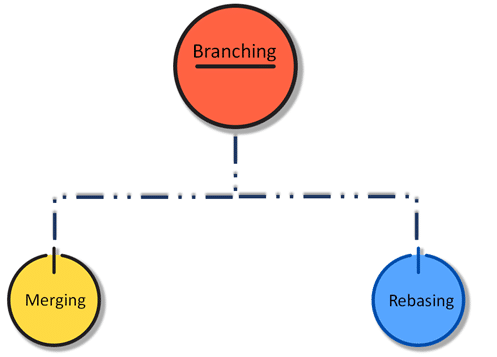
1. **Git remote add origin “your github repo URL”**
2. **Git push origin “branchName” -- eg: git push origin master**
3. **git pull**

When it comes to syncing a remote repository, the pull command comes in handy. Let us take a look at the commands that can lead to the pulling operation in Git.

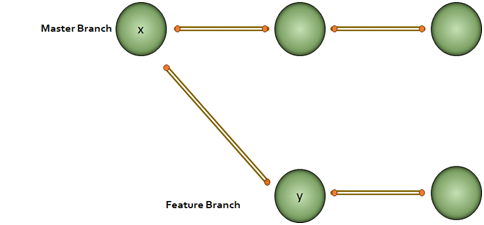
* **remote origin** : **git remote set-url origin “repository URL”**
* **pull master** : **git pull origin master**



## **Branching in Git**



* In branching, we mainly make use of the **git checkout** and **git branch** commands.



**Why would we do branching in the first place ?**

We want to modify something but don’t want to make any changes in the main project. This is when we make a branch out of the master branch, i.e., if we want to create a new branch to add a feature to the main project, we will make a branch out of it with the help of the following steps:

**Step 1**: We will run **git checkout -b <new branch name>**

**(To create a New Branch)**

**Step 2:** Then, we will use the **git branch** command to confirm that our branch is created

If we want to switch again to Master Branch then please follow below command :

**git checkout master**

## 

## 

## 

## 

## 

## **Merging in Git (git checkout, git add, git log, git merge, merging conflicts, and rebasing)**

## The merge commit represents every change that has occurred on the feature branch since it got branched out from the master.

**Note**: Even after merging, we can go on with our work on both the master and the feature branches independently.

* To check Git Logs : **git log / git log -5** (to check last 5 logs/history)

Steps to Merge from branch 1 to Master Branch :

* **Git checkout master**
* **git merge branch2**

**NOTE** :

**git log --graph**

**Advantages of merging:**

* Merging allows parallel working in collaborative projects.
* It saves the time that is consumed by manual merging.

**Disadvantage of merging:**

* Merging conflicts may occur while merging branches.

## 

## 

## **Git Rebase**

When our project becomes relatively large, the commit log and the history of the repository become messy. Here, we use **rebasing**.

**Rebasing** will take a set of commits, copy them, and store them outside our repository. This helps us maintain a linear sequence of commits in our repository.

**Note**: In rebasing, the base of the feature branch gets changed and the last commit of the master branch becomes the new base of the feature branch.

→ **Git rebase Master**

**Advantage of rebasing:**

Rebasing provides a cleaner project history.

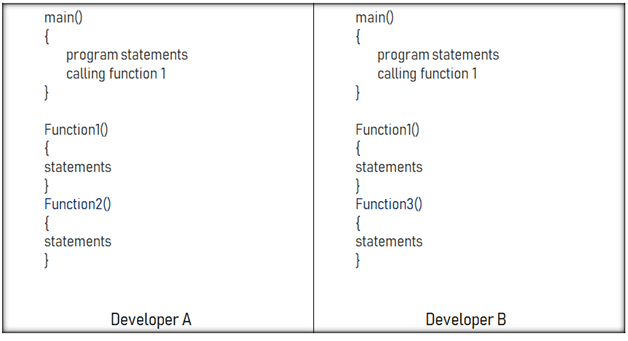
**Disadvantage of rebasing:**

In a collaborative workflow, re-writing the project history can be potentially catastrophic.

**The golden rule of Git merging and rebasing:**

* **When to use git merge**: We can use git merge while working on the **public branches**.
* **When to use git rebase**: We will use git rebase while working on the **local branches.**

**Git Merge Conflict and Rebase Conflict**



* **Developer A added a function called ‘function2’ to the main code.**
* **Developer B added a different function called ‘function3’ to the same code file.**

**How to merge these two modifications? That is where the merging conflict occurs.**

**Similarly, git rebase also exhibits conflicts**

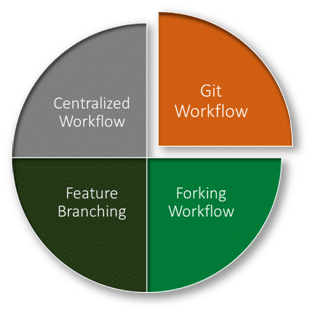
### **Solving the Conflicts**

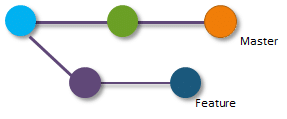
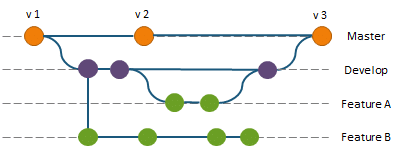
We can manually resolve the merge conflict using the merging tool.

We can use **file locking** which doesn’t allow different developers to work on the same piece of code simultaneously. It helps avoid merge conflicts but slows down the development process.

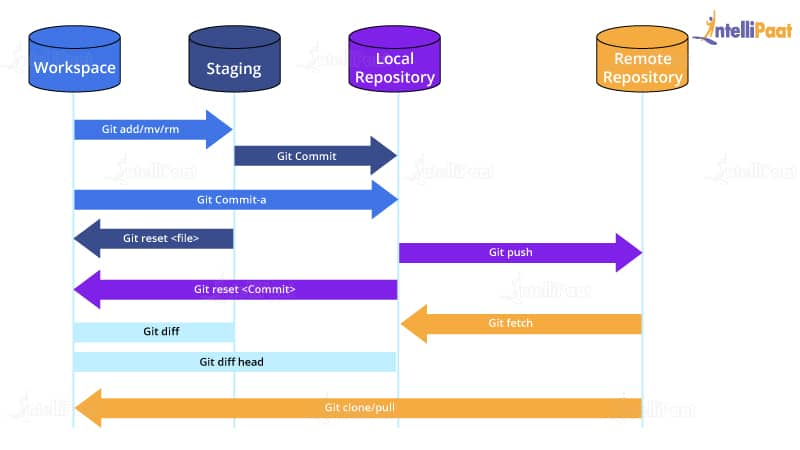
The rebasing conflict can also be solved with the help of the Git merging tool.

**Git Workflows**

****

* **Centralized workflow:** In the Git centralized workflow, only one development branch is there, which is called ‘master’ and all changes are committed into this single branch.
* **Feature branching workflow:** In the feature branching workflow, feature development takes place only in a dedicated feature branch. The below-given image in this Git tutorial depicts the feature branching workflow:
* 
* **Git workflow:** Instead of a single master branch, the Git workflow uses two branches. Here, the master branch stores the official release history, whereas the second ‘develop’ branch acts as an integration branch for features. The below-given image depicts the Git workflow:
* 
* **Forking workflow:** In the case of the forking workflow, the contributor has two Git repositories: one private local repository and the other public server-side repository.
* ======================================================================

**SUMMARY NOTES:**

* 

### 

**Git Commands When Working with Local Repositories**

* **git init**: This Git command converts a directory into an empty repository. This is the initial step you need to take to build a repository. Once you run git init, you will be able to add and commit files and directories.
* **git add:** This command allows you to add files in the Git staging area. The file must be added to the index of Git before being available to commit to any particular repository. You can use this command to add directories, files, etc.
* **git commit:** The commit command in Git allows you to track the changes in the files in a local repository. Each commit has its own unique ID for reference.
* **git status:** The git status command returns the present state of a repository, like if the file is in the staging area but has not been committed.
* **git config:** There are numerous configurations and settings possible in Git, and this command allows you to assign these settings. User.name and user.email are the two significant settings that set the name and email address of a user.
* **git branch:** This command determines the branch of the local repository and allows you to add or delete a branch.
* **git checkout:** You can use this command to switch to another branch.
* **git merge:** The merge command allows you to integrate two or more branches together. It combines the changes made in the branches.

**Git Commands When Working with Remote Repositories**

1. **git remote:** This Git command allows you to connect a remote repository to a local repository.
2. **git clone:** You can use the clone command to create a local copy of an already existing remote repository. This allows you to copy and download the required repository to the system. It is similar to the init command while working with remote repositories as it allows you to build a local directory, consisting of all the necessary files and history of the repository.
3. **git pull:** The pull command is used to run the latest version of any repository. This pulls all the changes made from the remote to the local repository.
4. **git push:** This command sends local commits to the respective remote repository. It needs two parameters, i.e., the remote repository and the specific branch where it needs to be pushed.

There are numerous other Git commands that are of more advanced level, such as **git stash, git log, git rm,** etc.

===========================================================================

**Here are the top 18 Git commands list discussed in this tutorial:**

* [**git init**](https://intellipaat.com/blog/tutorial/devops-tutorial/git-commands/#git-init)

git init [repository name]

We have to navigate to our project directory and type the command git init to initialize a Git repository for our local project folder. Git will create a hidden .git directory and use it for keeping its files organized in other subdirectories.

* [**git add**](https://intellipaat.com/blog/tutorial/devops-tutorial/git-commands/#git-add)

Usage (i): git add [file(s) name]

This will add the specified file(s) into the Git repository, the staging area, where they are already being tracked by Git and now ready to be committed.

* [**git commit**](https://intellipaat.com/blog/tutorial/devops-tutorial/git-commands/#git-commit)

Usage: git commit -m “message”

This command records or snapshots files permanently in the version history. All the files, which are there in the directory right now, are being saved in the Git file system.

* [**git status**](https://intellipaat.com/blog/tutorial/devops-tutorial/git-commands/#git-status)

Usage: git status

This command will show the modified status of an existing file and the file addition status of a new file, if any, that have to be committed.

* [**git remote**](https://intellipaat.com/blog/tutorial/devops-tutorial/git-commands/#git-remote)

Usage: git remote add origin “[URL]”

Once everything is ready on our local system, we can start pushing our code to the remote (central) repository of the project.

* [**git push**](https://intellipaat.com/blog/tutorial/devops-tutorial/git-commands/#git-push)

Usage: git push origin [branch name]

By using the command ‘git push,’ the local repository’s files can be synced with the remote repository on Github.

* [**git clone**](https://intellipaat.com/blog/tutorial/devops-tutorial/git-commands/#git-clone)

Usage: git clone [URL]

Suppose, we want to work on a file that is on a remote Github repository as another developer. How can we do that? We can work on this file by clicking on Clone or Download and copying the link and pasting it on the terminal with the git clone command. This will import the files of a project from the remote repository to our local system.

**Note**: Here, we don’t have to use the git remote add origin command because we have already cloned the remote repository in the local directory. Now, if we push any new file, it knows where it has to go.

* [**git branch**](https://intellipaat.com/blog/tutorial/devops-tutorial/git-commands/#git-branch)

git branch [name-of-the-branch]

Imagine multiple developers working on the same project or repository! To handle the workspace of multiple developers, we can use branches. To create a branch (say, the ‘name-of-the-branch’ is ‘branch1’), we use this command .[**git checkout**](https://intellipaat.com/blog/tutorial/devops-tutorial/git-commands/#git-checkout)

git checkout [name-of-the-new-branch]

* [**git log**](https://intellipaat.com/blog/tutorial/devops-tutorial/git-commands/#git-log)
* [**git stash**](https://intellipaat.com/blog/tutorial/devops-tutorial/git-commands/#git-stash)
* [**git revert**](https://intellipaat.com/blog/tutorial/devops-tutorial/git-commands/#git-revert)
* [**git diff**](https://intellipaat.com/blog/tutorial/devops-tutorial/git-commands/#git-diff)
* [**git merge**](https://intellipaat.com/blog/tutorial/devops-tutorial/git-commands/#git-merge)
* [**git rebase**](https://intellipaat.com/blog/tutorial/devops-tutorial/git-commands/#git-rebase)
* [**git fetch**](https://intellipaat.com/blog/tutorial/devops-tutorial/git-commands/#git-fetch)
* [**git reset**](https://intellipaat.com/blog/tutorial/devops-tutorial/git-commands/#git-reset)
* [**git pull**](https://intellipaat.com/blog/tutorial/devops-tutorial/git-commands/#git-pull)

**REALTime Errors :**

! [rejected] master -> master (non-fast-forward)

error: failed to push some refs to 'https://github.com/abhishekpandaOfficial/EmployeeManagement.git'

1. git fetch origin master:tmp
2. git rebase tmp
3. git push origin HEAD:master
4. git branch -D tmp