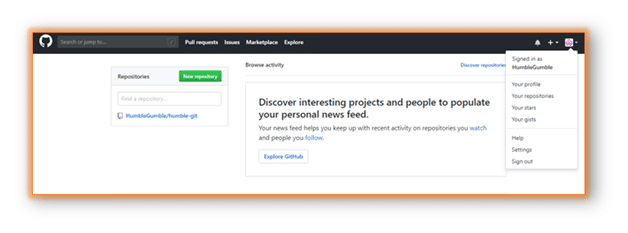
**Environment Setup**

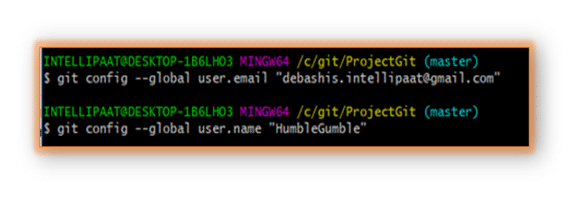
* Create a GitHub account
* Configure Git
* Create a local repository

**Creating a GitHub account:**

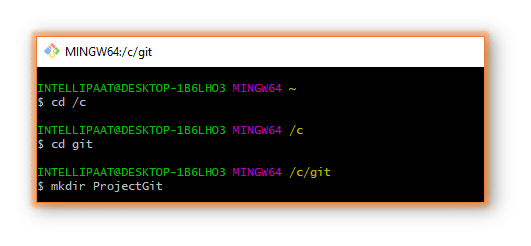
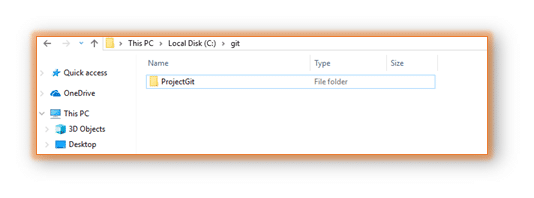
* Go to https://Github.com
* Create a GitHub account
* Login

After accomplishing these steps, our GitHub page should look like this:  


**Configuring Git:**

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


**Creating a local repository:**

To begin with, open a terminal and move to where we want to place the project on our local machine using the cd (change directory) command. For example, if we have a ‘projects’ folder on our desktop, we would do something like below:  
We can go to the Git folder and see the new directory that we have just created.  
Finally, we are all set to start working on [Git commands](https://intellipaat.com/blog/tutorial/devops-tutorial/git-commands/).

***Learn more about DevOps in this***[***DevOps Training in Sydney***](https://intellipaat.com/devops-certification-training-sydney/)***to get ahead in your career!***

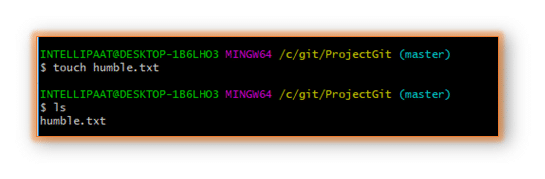
**Common Git Commands**

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

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

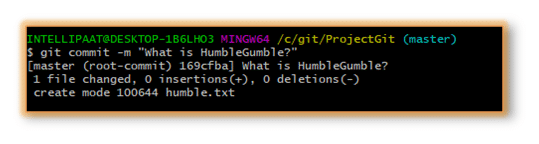
Next in Git and GitHub tutorial, we will discuss the local Git commands.

**Local Git Commands**

* **git init**: We use the git init command to initialize a Git repository in the root of a folder.
* **git touch**: To add files to a project, we can use git touch. Let us see how we can add a file to the Git repository we have just created  
  **Step 1**: Create a new file with the command touch  
  **Step 2**: See the files present in our master branch  
  
* **git status**: After creating a new file, we can use the git status command and see the status of the files in the master branch.  
  
* **git status**: Now, we will add the humble.txt file to the staging environment by using git add, before going ahead to the staging to see the change using git status.



* **git commit**: Now we will use the git commit command as shown below:

Thus, we have successfully created our first commit.

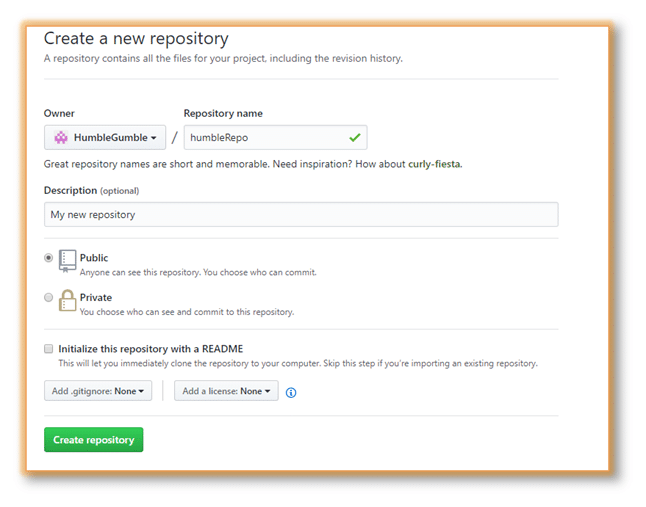
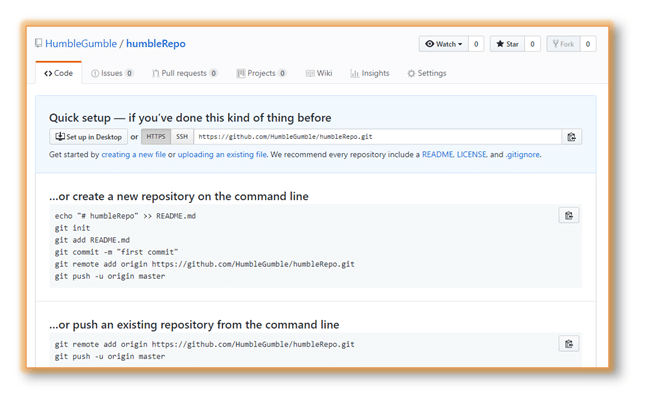
* **Git clone:**



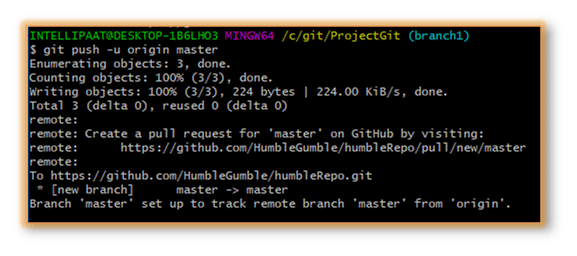
**Remote Commands**

Let us now discuss the remote commands such as remote, push, and pull in Git. For that, we will create a new repository in our GitHub account. How do we do that? Follow the below steps:

**Step 1**: Go to the GitHub account  
**Step 2**: Create a new repository

Once we are done with filling up the new repository form, we should land on a page like the following:  
Now, we are ready to operate remote commands in our repository that we have just created.

To push an existing repository to a remote repository from a command line, follow the commands given below.

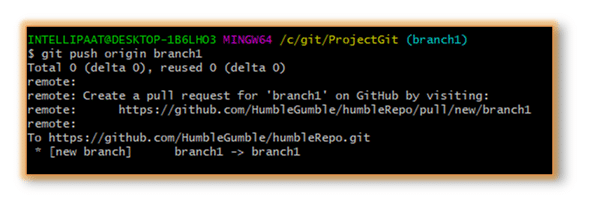
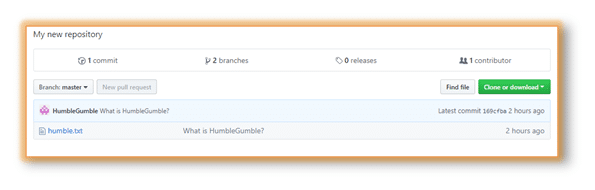
* **git remote:**After this, we will provide the following command:
* **git push**

git push origin EnterBranchName

Don’t get confused by the word ‘origin’. What exactly is this command referring to?

Instead of writing the whole Git URL, like **git push Git@Github.com:Git/Git.Git ourbranchname**, we can just use the following command, assuming that our brach name is ‘branch1’:

git push origin branch1

Now, let us look at our repository:  


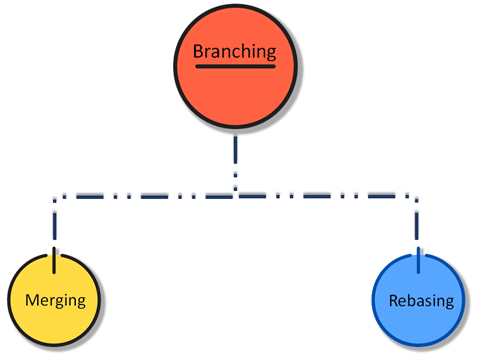
* **git pull**

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

* remote origin
* pull master

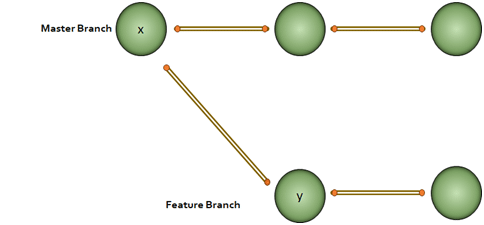
We will use these as shown below:  
***Interested in getting an industry-recognized certification in DevOps? Enroll in Intellipaat’s***[***DevOps Course in Bangalore***](https://intellipaat.com/devops-certification-training-bangalore/)***now!***

Now that we have learned the workflow of Git with the help of the common Git commands, let us take a look at branching, merging, and rebasing and also at how and when to use them.

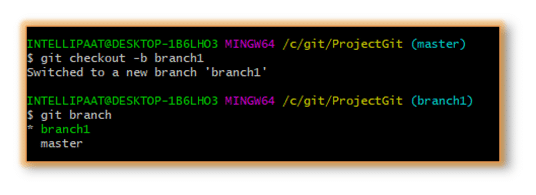


**Branching in Git**

We are almost familiar with various Git commands now. Let’s take a step further and discuss one of the most important operations in Git, namely, branching. In branching, we mainly make use of the **git checkout** and **git branch** commands.

  
Every node in the figure above represents commits. We have to note that for the ‘y’ node in the feature branch, ‘x’ is the base.

Why would we do branching in the first place? Say, we want to modify something but don’t want to make any change 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>**  
**Step 2**: Then, we will use the git branch command to confirm that our branch is created  
We can see from the above image that we have created a branch called ‘branch1’ and we automatically got landed in the new branch. Again, just to see which branch we are currently in, we run another command, git branch.

**Note:** The **\*** mark before the branch name shows that it is the current branch.

Now, how to change a branch to the master branch? For that, we use the following command:

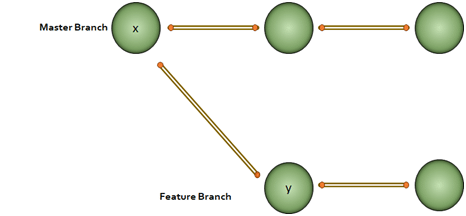
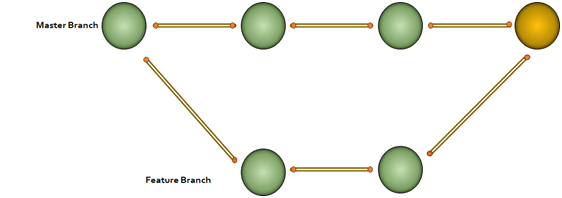
git checkout master

***DevOps Engineers are among the highest paid professionals in the technology domain. Enroll in***[***DevOps Training in Hyderabad***](https://intellipaat.com/devops-certification-training-hyderabad/)***today!***

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

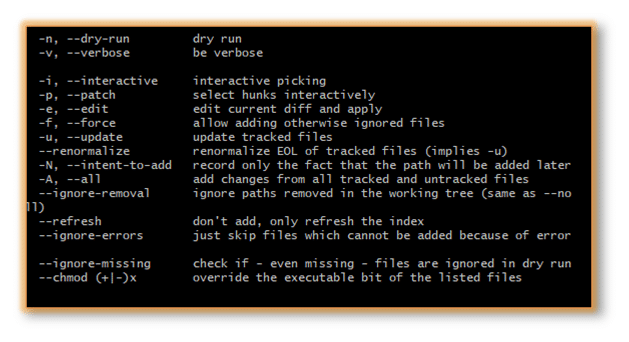
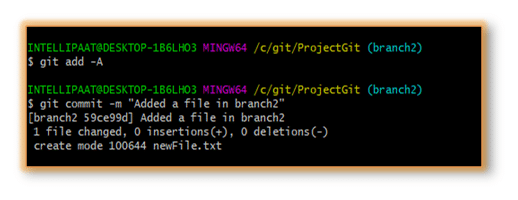
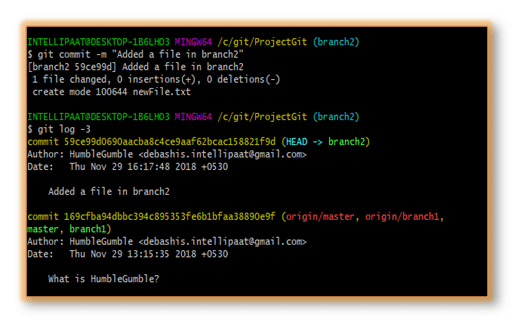
Now that we have learned how to create a branch and work on it, let us take a look at the merge feature in Git by merging the branch we created to the master branch.

Let’s take the above example. Say, we have a master branch and a feature branch.

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.

Let us see how to perform merging:

**Step 1**: Create a new branch called ‘branch2’  
**Step 2**: Create a new file in the branch  
**Step 3**: Add changes from all tracked and untracked files

**Note**: Refer to the following **git add** attributes  
In our case, we have given the command as **git add -A** and after that, we will commit one sentence as shown below:  
**Step 4**: Check the last three logs by running the command: **git log -3**  
We have created another branch on our master branch. Now, we will see how to perform **merging**.

Let us get inside the master branch using the following command:

git checkout master

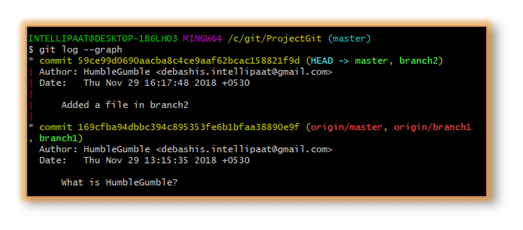
After that, we will perform merging with the help of the below command:

git merge branch2

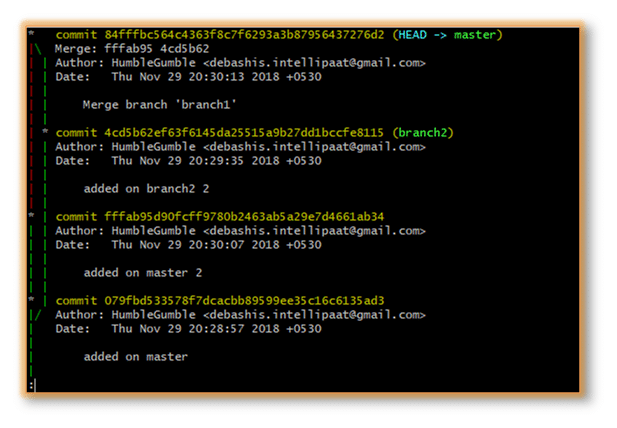
  
***Get in touch with Intellipaat for a comprehensive***[***DevOps Training***](https://intellipaat.com/devops-certification-training/)***and be a certified DevOps Engineer!***

Now, we have successfully merged the feature branch into the master branch. Let us take a look at how it looks inside the master branch using the following command:

git log --graph

We can see the graph in the above image on the left. But **why the graph is linear?**

In our ProjectGit master branch, we did branching and committing. But after the branching, the master branch had not encountered any more commits. Hence, after merging, we have a linear graph.

Let us perform**git log –graph** with more than one commits in both master and feature branches that we have created.  
Here, we can see the graph with commits on both master and feature branches. The red part of the graph indicates the merging operation.

**Advantages of merging:**

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

**The disadvantage of merging:**

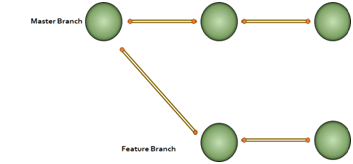
* Merging conflicts may occur while merging branches.

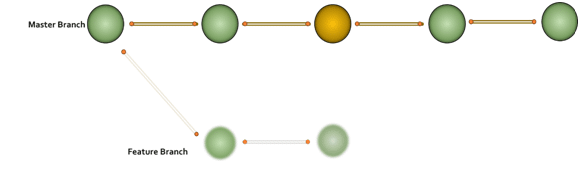
Now that we have successfully learned to branch and merge with Git and GitHub, further in this best Git tutorial, let us look at yet another important Git operation, i.e., rebasing.

**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.

Let us take the same example. Here, we will rebase the master branch and see what happens.

**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.

Now, let us perform rebasing in Git Bash.



**The advantage of rebasing:**

* Rebasing provides a cleaner project history.

**The disadvantage of rebasing:**

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

Now that we understood what branching, merging, and rebasing are, next in this Git tutorial, we will see where to use merging and rebasing.

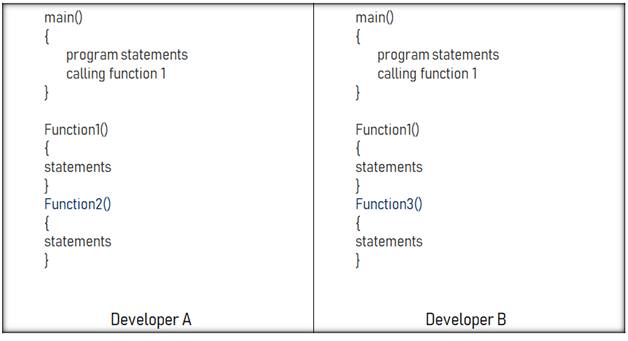
**The golden rule of Git merging and rebasing:**

* **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.

As we have already mentioned earlier in this Git commands tutorial, one major disadvantage of merging is merge conflicts, which can be backbreaking for a team project. Let us understand how merge conflicts occur and how to resolve them.

**Git Merge Conflict and Rebase Conflict**

This drawback of the merging operation in Git can be explained with a simple example shown below:

Say, we have two branches of the master branch, branch1 and branch2, and two developers are working on these two branches independently but on the same code file.

As we have seen in the above image, the developers have made the below-mentioned modifications:

* 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

[](https://intellipaat.com/course-cat/big-data-analytics-courses/)

**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.

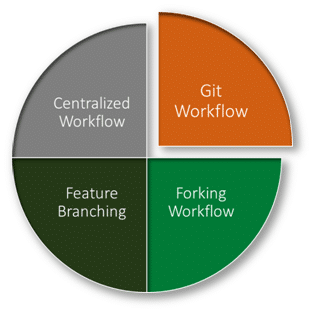
Now, let us look at a few Git workflows briefly before we end this Git tutorial so that we get an idea of which Git workflow to choose for our team project.

**Git Workflows**

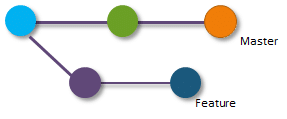
In this section, we will get introduced to different workflow options available in Git. Once we get an idea of these workflows, we can choose the right one for our team project.

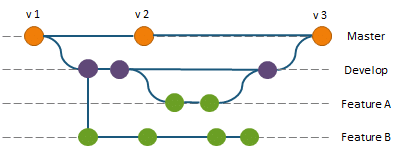
  
**But, why is it important to choose the right Git workflow?**

Depending upon the team size, choosing the right Git workflow is important for a team project to increase its productivity.

Now in this Git tutorial, let us look at the different 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.

**GIT 18 COMMANDS:**

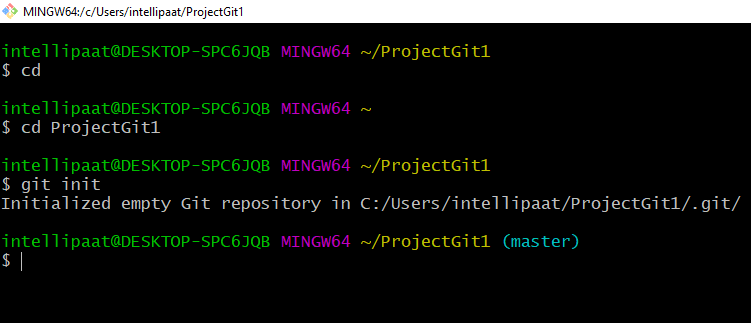
## Git Commands

[**Git**](https://intellipaat.com/blog/what-is-git/)is a free, open-source distributed version control system tool designed to handle small to very large projects with speed and efficiency. It has steadily grown from just being a preferred skill to a must-have skill for multiple job roles today. Git has become an essential part of our everyday development process.

### 1. git init

**Usage: git init [repository name]**

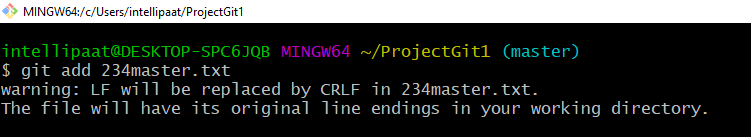
We have to navigate to our project directory and type the command **git init** to initialize a Git repository for our local project folder. Git will create a hidden **.git** directory and use it for keeping its files organized in other subdirectories.

[](https://intellipaat.com/mediaFiles/2019/07/GItCommand1.png)

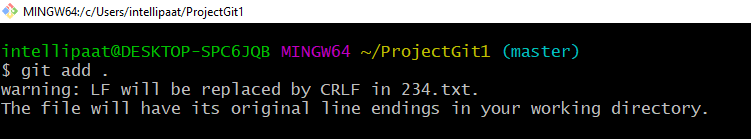
### 2. git add

**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.

[](https://intellipaat.com/mediaFiles/2019/07/GItCommand2.png)  
**Usage (ii): git add . or git add \***

This will take all our files into the Git repository, i.e., into the staging area.

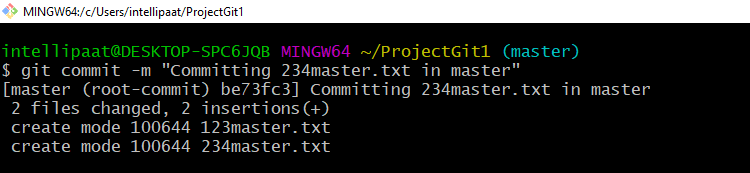
[](https://intellipaat.com/mediaFiles/2019/07/GItCommand3.png)  
We can use this command as **git add -A** as well.

**Note**: We will have to commit our files after we add them to the staging area.

### 3. git commit

**Usage: git commit -m “message”**

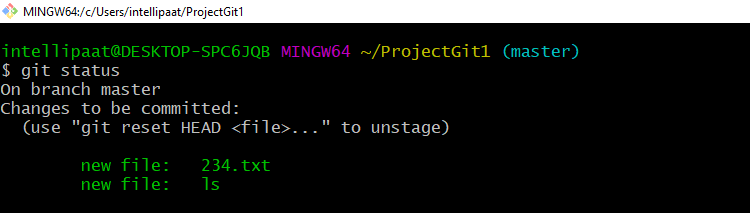
This command records or snapshots files permanently in the version history. All the files, which are there in the directory right now, are being saved in the Git file system.

**[](https://intellipaat.com/mediaFiles/2019/07/GItCommand4.png)**

### 4. 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 has to be committed.

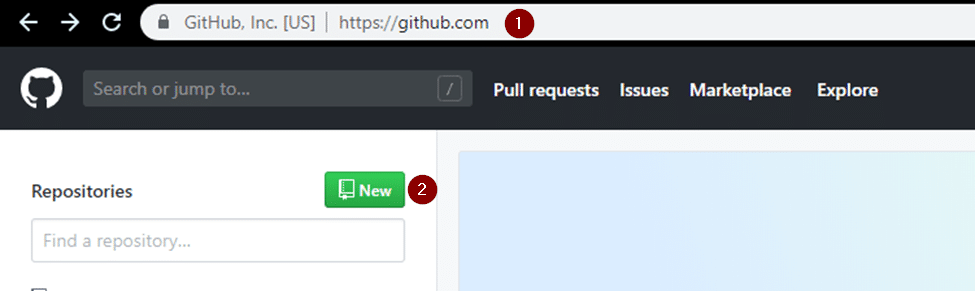
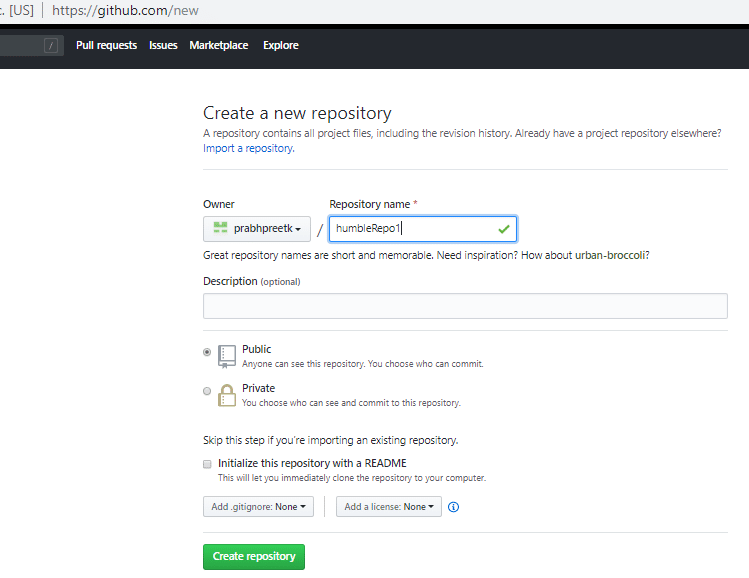
[](https://intellipaat.com/mediaFiles/2019/07/GItCommand5.png)

### 5. git remote

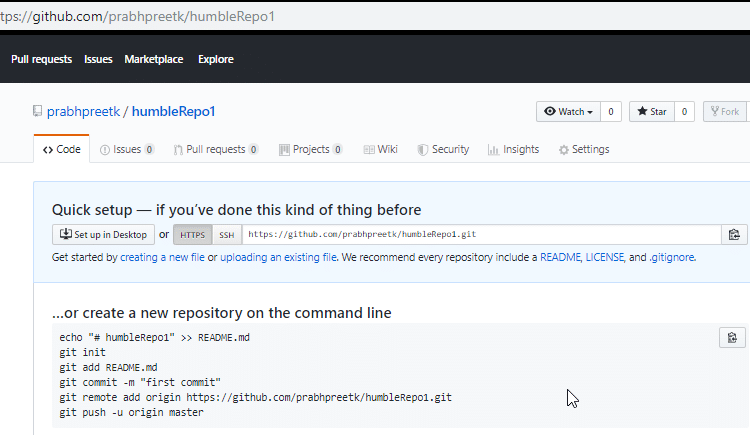
**Usage: git remote add origin “[URL]”**

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

**Step 1:**  
**(1)**Login to the **GitHub account** if the account already exists (If not, sign up on github.com)  
**(2)** Click on**New**

[](https://intellipaat.com/mediaFiles/2019/07/GItCommand6.png)  
**Step 2:**Now, we have to create a new repository. Provide a **name**to our**repository**, select the **privacy**of the repository as**Public**, and then click on **Create repository**  
**[](https://intellipaat.com/mediaFiles/2019/07/GItCommand7.png)This**[***GIT Cheat Sheet***](https://intellipaat.com/blog/tutorial/devops-tutorial/git-cheat-sheet/)**by Intellipaat is very useful and handy. Give it a read.**

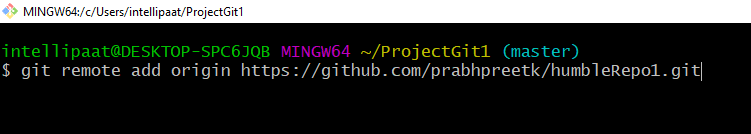
Once we are done with filling up the new repository form, we would land on a page as follows:

[](https://intellipaat.com/mediaFiles/2019/07/GItCommand8.png)

**Step 3:** Click on the Copy icon on the right side of the URL box of the Github repository to copy the link and paste it as shown below:

git remote add origin “URL”

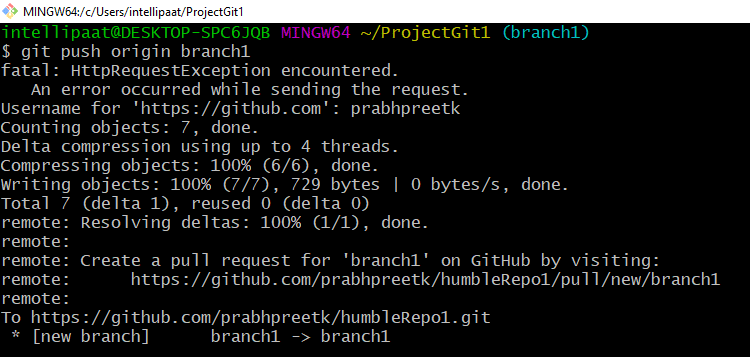
Now, we are ready to operate the remote commands in our repository that we have just created.

[](https://intellipaat.com/mediaFiles/2019/07/GItCommand9.png)

### 6. git push

**Usage: git push origin [branch name]**

Suppose, we have made some changes in the file and want to push the changes to our remote repository on a particular branch. By using the command ‘git push,’ the local repository’s files can be synced with the remote repository on Github.

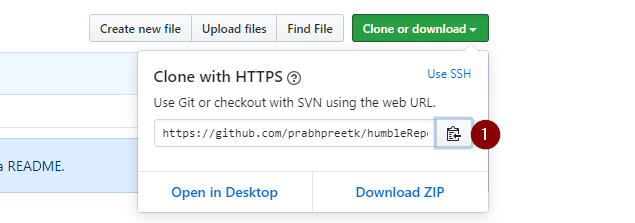
[](https://intellipaat.com/mediaFiles/2019/07/GItCommand10.png)

### 7. 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.

**(1)** The below screenshot shows from where to copy the link:

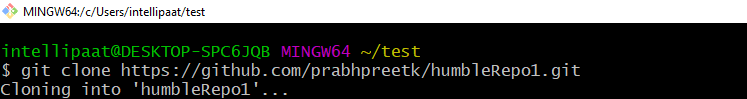
[](https://intellipaat.com/mediaFiles/2019/07/GItCommand11.png)To create a local folder, we have to use the following command:

mkdir [directory- name]

cd [directory- name]

git clone [URL]

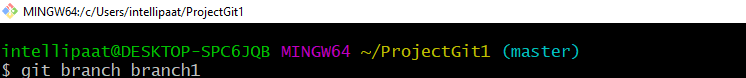
Now, paste the copied link along with the git clone command as shown below:

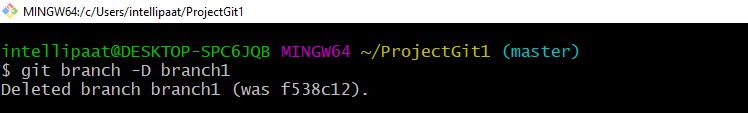
[](https://intellipaat.com/mediaFiles/2019/07/GItCommand12.png)**Note**: Here, we don’t have to use the **git remote add origin** command because we have already cloned the remote repository in the local directory. Now, if we push any new file, it knows where it has to go.

### 8. git branch

**Usage (i): git branch [name-of-the-branch]**

So far, we saw how we can work on Git. Now, 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:

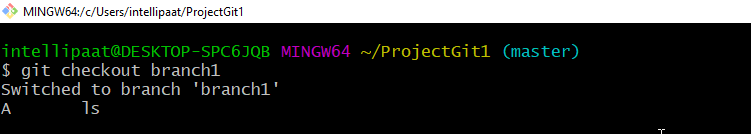
[](https://intellipaat.com/mediaFiles/2019/07/GItCommand13.png)  
**Usage (ii): git branch -D [name -of-the-branch]**

Similarly, to delete a branch, we use the **git branch -D**command:  
**[](https://intellipaat.com/mediaFiles/2019/07/GItCommand14.png)**

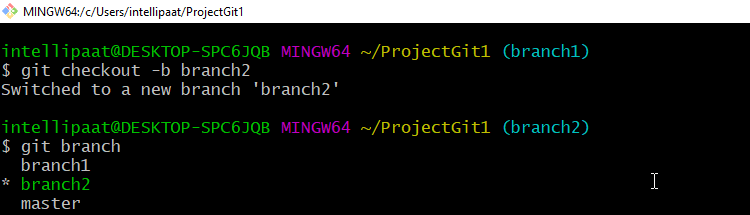
### 9. git checkout

**Usage (i): git checkout [name-of-the-new-branch]**

We use this command to navigate to an existing branch, add new files, and commit the files:

**[](https://intellipaat.com/mediaFiles/2019/07/GItCommand15.png)Usage (ii): git checkout -b [name-of-the-new-branch]**

We use this command to create a branch and navigate to that particular branch (say, the ‘name-of-the-new-branch’ is ‘branch2’) at the same time:

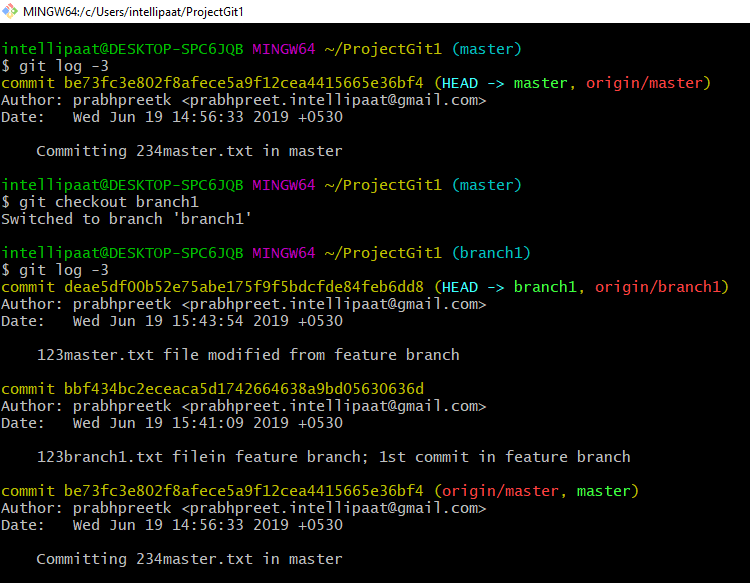
[](https://intellipaat.com/mediaFiles/2019/07/GItCommand16.png)

**10. git log**

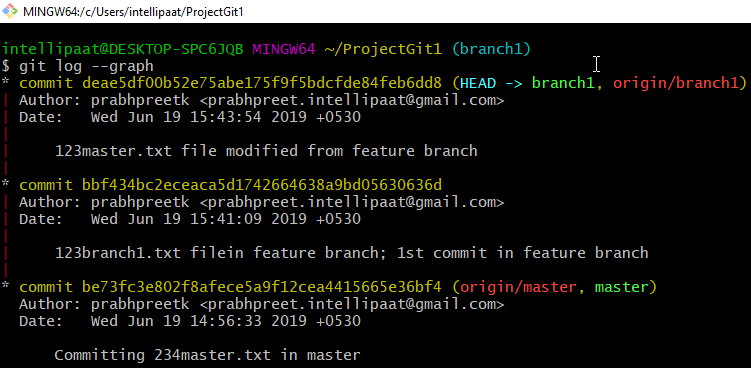
**Usage (i): git log**

This command is used when we want to check the log for every commit in detail in our repository.

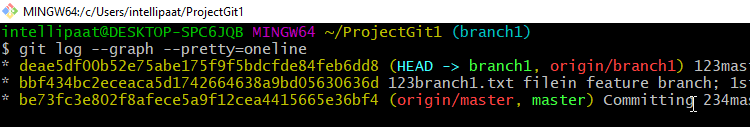
**Note**: It will show the log of the branch we are in. We can check the last three logs by giving the command: **git log -3**

[](https://intellipaat.com/mediaFiles/2019/07/GItCommand17.png)

**Usage (ii): git log –graph**

[](https://intellipaat.com/mediaFiles/2019/07/GItCommand18.png)

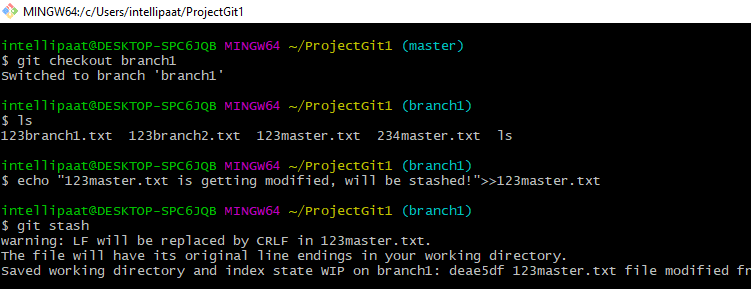
**Usage (iii): git log –graph –pretty=oneline**

**[](https://intellipaat.com/mediaFiles/2019/07/GItCommand19.png)**

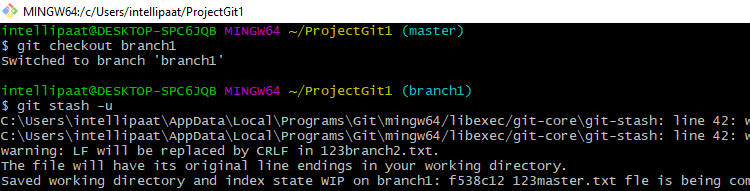
**11. git stash**

**Usage (i): git stash**

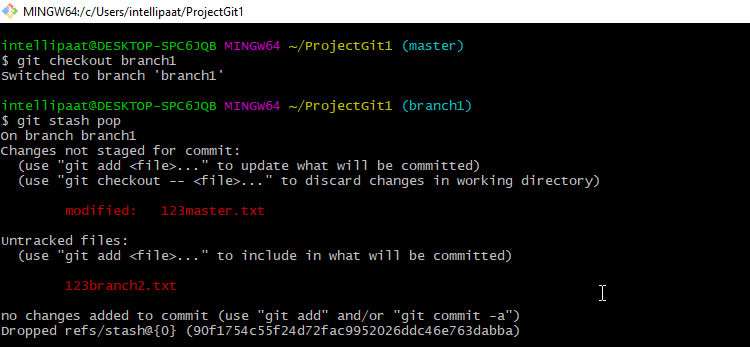
This command can be used when we want to save our work without staging or committing the code to our Git repository and want to switch between branches.

[](https://intellipaat.com/mediaFiles/2019/07/GItCommand20.png)  
**Usage (ii): git stash -u**

This command is used when we want to stash the untracked files.

[](https://intellipaat.com/mediaFiles/2019/07/GItCommand21.png)  
**Usage (iii): git stash pop**

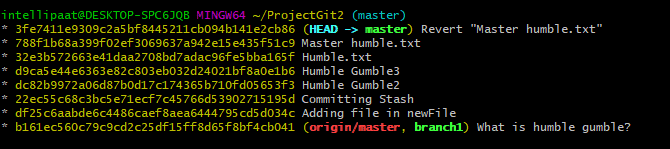
This command is used when we are back on our branch and want to retrieve the code.

[](https://intellipaat.com/mediaFiles/2019/07/GItCommand22.png)

### 12. git revert

**Usage: git revert [commit id]**

The git revert command can be considered as an ‘undo’ command. However, it does not work as the traditional ‘undo’ operation. It figures out how to invert the changes introduced by the commit and appends a new commit with the resulting inverse content.

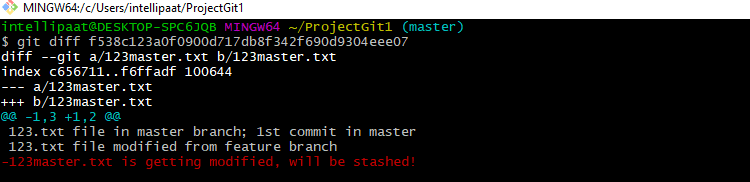
[](https://intellipaat.com/mediaFiles/2019/07/GItCommand23.png)

**13. git diff**

**Usage: git diff [commit-id-of-version-x] [commit-id-of-version-y]**

Diffing is a function that takes two input datasets and outputs the changes between them. The git diff command is a multi-use Git command which, when executed, runs a diff function on Git data sources. These data sources can be commits, branches, files, and more. The git diff command is often used along with the git status and git log commands to analyze the current state of our Git repository. We use **git log** to get the details of commit IDs.

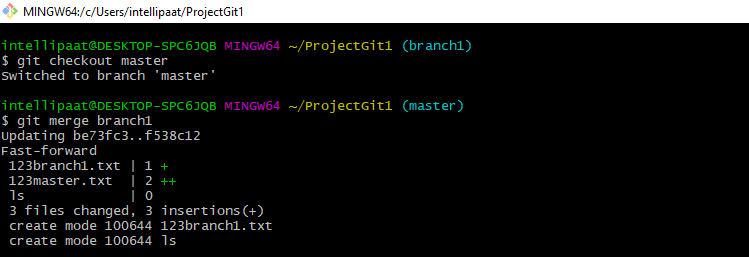
Let’s compare the working directory with the index as shown below:

[](https://intellipaat.com/mediaFiles/2019/07/GItCommand24.png)

**14. git merge**

**Usage: git merge [another-file-name]**

This command will combine multiple sequences of commits into one unified history. In the most frequent use cases, git merge is used to combine two branches. The git merge command takes two commit pointers, usually the branch tips, and finds a common base commit between them. Once it finds a common base commit, it will create a commit sequence.

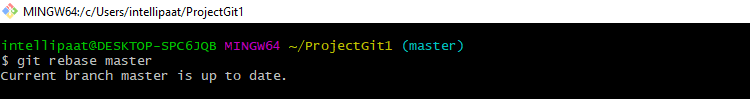
[](https://intellipaat.com/mediaFiles/2019/07/GItCommand25.png)

**15. git rebase**

**Usage: git rebase [base]**

Rebase is the process of moving and combining a sequence of commits to a new base commit. Rebasing is changing the base of our branch from one commit to another, making it appear as if we’ve created our branch from a different commit. Internally, Git accomplishes this by creating new commits and applying them to the specified base. It’s very important to understand that even though the branch looks the same, it is composed of entirely new commits.

The git rebase command performs an automatic git checkout <branch> before doing anything else. Otherwise, it remains on the current branch.

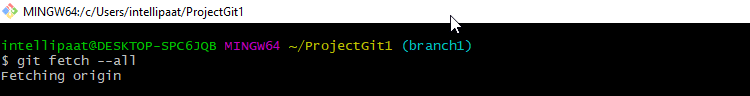
[](https://intellipaat.com/mediaFiles/2019/07/GItCommand26.png)  
Consider a situation where we have branched off from the master and have created a feature branch, but the master branch is still having more commits. We want to get the updated version of the master branch in our feature branch, keeping our branch’s history clean, so that it appears as if we are working on the latest version of the master branch.

**Note**: We don’t rebase public history. We should never rebase commits once they are pushed to a public repository. Why because the rebase would replace the old commits with the new ones, and it would appear that a part of our project history got abruptly vanished.

**16. git fetch**

**Usage: git fetch**

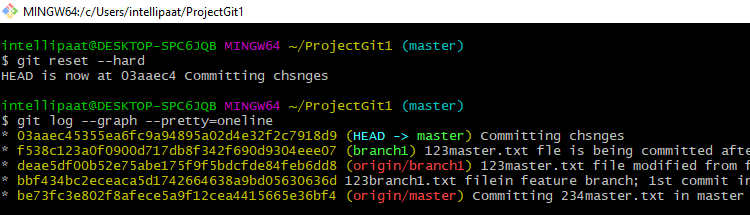
When we use the command git fetch, Git gathers any commit from the target branch that does not exist in our current branch and stores it in our local repository. However, **it does not merge it with our current branch**.

[](https://intellipaat.com/mediaFiles/2019/07/GItCommand27.png)  
This is particularly useful when we need to keep our repository up to date but are working on something that might break if we updated our files. To integrate the commits into our master branch, we use merge. It fetches all of the branches from the repository. It also downloads all the required commits and files from another repository.

**17. git reset**

**Usage: git reset –hard [SOME-COMMIT]**

We use this command to **return** the *entire* working tree to the last committed state.

[](https://intellipaat.com/mediaFiles/2019/07/GItCommand28.png)  
This will discard commits in a private branch or throw away the uncommitted changes!

Here, we have executed a ‘hard reset’ using the **–hard** option. Git displays the output indicating that the HEAD is pointing to the latest commit. Now, when we check the state of the repo with git status, Git will indicate that there are no pending changes (if any prior addition of a new file or modification of an existing file is done before using the ‘git reset –hard’ command). Our modifications to an existing file, if not committed, and the addition of a new file, if not staged, will be destroyed. It is critical to take note that this data loss cannot be undone.

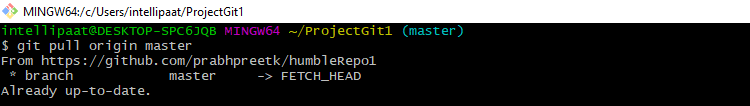
If we do**git reset –hard [SOME-COMMIT]**, then Git will**:**

* Make our current branch (typically master) back to point <SOME-COMMIT>
* Make the files in our working tree and the index (“staging area”) the same as the versions committed at <SOME-COMMIT>

**18. git pull**

**Usage: git pull origin master**

The git pull command first runs ‘git fetch’ which downloads the content from the specified remote repository and then immediately updates the local repo to match the content.

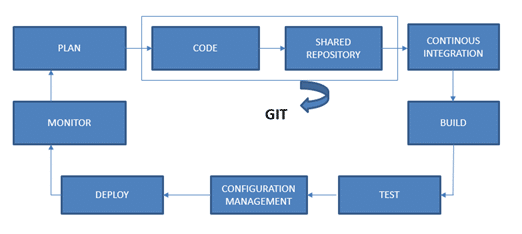
[](https://intellipaat.com/mediaFiles/2019/07/GItCommand30.png)

GIT CHEATSHEET

**Process Flow**

It shows the process flow with components used to perform certain tasks:

* **Plan**: It can be created or deleted by itself based on the updates in the source repository.
* **Code**: It is the statement written to perform a task in a repository.
* **Shared repository**: It is the repository that is shared among several members in order to perform a task.
* [**Continuous integration**](https://intellipaat.com/blog/what-is-continuous-integration/): It helps in updating a repository with the changes made in the code and provides integration with the changes, time to time.
* **Build**: It is used to manually build products and dependencies to Git.
* **Configuration management**: It is a process of tracking and controlling the changes made in a system.
* **Deploy**: Deploy is the process of pushing the code to a remote server.



**Version Control**

Version control is the management of changes made to the code, documents, programs, large sites, and other information. The changes are termed as **versions**.

A version control system (VCS) is used to perform the following:

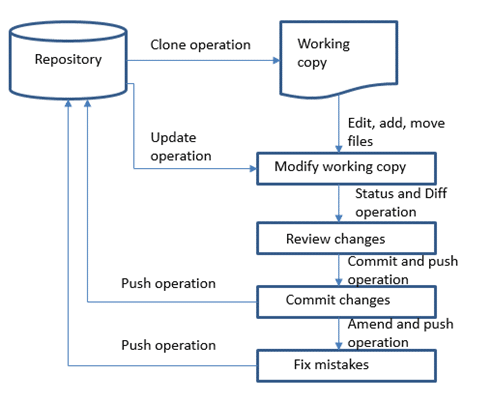
* It allows developers to work simultaneously.
* VCS does not allow overwriting each other’s changes.
* It maintains a history of every version.

There are two types of version control systems:

* Centralized VCS
* Distributed VCS

Our DevOps tool, Git falls under the category of distributed VCS.

**Life Cycle**



**Git and GitHub**

Git is a VCS that supports distributed nonlinear workflows by providing data assurance for developing quality software.

Its features are as follows:

* **Distributed**: A distributed development of code
* **Compatible**: Works with existing systems and protocols
* **Non-linear**: Allows the non-linear development of code
* **Branching**: Easy to create and merge branches
* **Lightweight**: Lossless compression
* **Speed**: Faster than the remote repository
* **Open-source**: A free tool and hence economical
* **Reliable**: Not viable to any loss of data upon crashes
* **Secure**: Uses SHA1 and checksum

**Git Operations and Commands**

**Git Configuration**

* For the initial configuration of username, email, and code highlighting (optional):

$git config -- global user.name”firstname lastname”

$git config -- global user.email” abc123@abc.com”

$git config -- global color.ui true *(enables code highlights*)

$git config --list

**Initializing Git**

* To initialize:

$git init

* To know the status:

$git status

**Adding and Removing Files**

* To add a file:

$git add<filename>

* To add multiple files:

$git add<filename> <2nd filename> <3rd filename>

* To add all the updated files:

$git add --all ( use -A instead of -all too )

* To remove files:

$git rm -r <filename>

**Committing Changes**

* To pass a message, use ‘commit’ with ‘-m’:

$git commit -m “body\_of\_message”

* To amend the last commit or the last message:

$git commit --amend -m “new\_message”

***Learn DevOps from Intellipaat’s***[***DevOps Training in Sydney***](https://intellipaat.com/devops-certification-training-sydney/)***to boost your career!***

**Pushing and Pulling**

Here, a remote repository typically represents a remote server or a Git server.

* To create a remote repository via GitHub, go to:

https://github.com/YourUsername/appname.git

* To add a link:

$git remote add origin<link>

* To push files:

$git push -u origin master

* To clone files:

$git clone <clone>

**Branching and Merging**

Following are the branching and merging [**commands for GIT**](https://intellipaat.com/blog/tutorial/devops-tutorial/git-commands/) :

|  |  |
| --- | --- |
| **Command** | **Description** |
| git branch | To list branches |
| git branch -a | To list all the branches |
| git branch [branch name] | To create a new branch |
| git branch -d [branch name] | To delete a branch |
| git push origin –delete [branchName] | To delete a remote branch |
| git checkout -b [branch name] | To create a new branch and switch to it |
| git checkout -b [branch name] origin/[branch name] | To clone a remote branch and switch to it |
| git checkout [branch name] | To switch to a branch |
| git checkout – | To switch to the branch last checked out |
| git checkout — [file-name.txt] | To discard the changes made to a file |
| git merge [branch name] | To merge a branch into an active branch |
| git stash | To stash the changes in a dirty working directory |
| git stash clear | To remove all the stashed entries |

***Learn more about the basic commands in this***[***Git Tutorial***](https://intellipaat.com/blog/tutorial/devops-tutorial/git-tutorial/)***by Intellipaat.***

**Sharing and Updating Projects**

|  |  |
| --- | --- |
| **Command** | **Description** |
| git push origin [branch name] | To push a branch to a remote repository |
| git push -u origin [branch name] | To push the changes made to a remote repository (-u remembers the branch for the next use) |
| git push origin –delete [branch name] | To delete a remote branch |
| git pull | To update a local repository to the newest commit |
| git pull origin [branch name] | To pull the changes from a remote repository |
| git remote add origin ssh://git@github.com/[username]/[repository-name].git | To add a remote repository |
| git remote set-url origin ssh://git@github.com/[username]/[repository-name].git | To set a repository’s origin branch to SSH |

**Inspection and Comparison**

|  |  |
| --- | --- |
| **Command** | **Description** |
| git log | To view the changes made |
| View changes | To view changes (in detail) |
| git diff [source branch] [target branch} | To preview changes before merging |