1) Git config command

This command configures the user. The Git config command is the first and necessary command used on the Git command line. This command sets the author name and email address to be used with your commits. Git config is also used in other scenarios.

**Syntax**

1. $ git config --global user.name "ImDwivedi1"
2. $ git config --global user.email "Himanshudubey481@gmail.com"

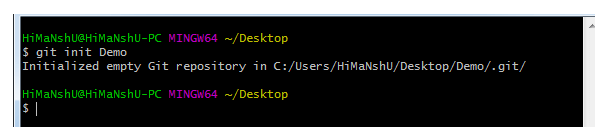
2) Git Init command

This command is used to create a local repository.

**Syntax**

1. $ git init Demo

The init command will initialize an empty repository. See the below screenshot.

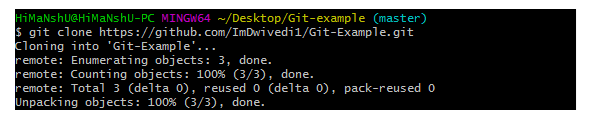


3) Git clone command

This command is used to make a copy of a repository from an existing URL. If I want a local copy of my repository from GitHub, this command allows me to create a local copy of that repository on your local directory from the repository URL.

**Syntax**

1. $ git clone URL



4) Git add command

This command is used to add one or more files to the staging (Index) area.

**Syntax**

To add one file

1. $ git add Filename

To add more than one file

1. $ git add\*

Git Commands

5) Git commit command

Commit command is used in two scenarios. They are as follows.

**Git commit -m**

This command changes the head. It records or snapshots the file permanently in the version history with a message.

**Syntax**

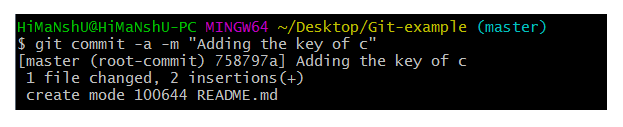
1. $ git commit -m " Commit Message"

**Git commit -a**

This command commits any files added in the repository with git add and also commits any files you've changed since then.

**Syntax**

1. $ git commit -a

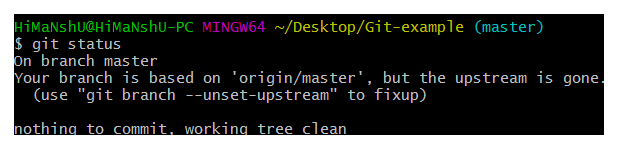


6) Git status command

The status command is used to display the state of the working directory and the staging area. It allows you to see which changes have been staged, which haven't, and which files aren?t being tracked by Git. It does not show you any information about the committed project history. For this, you need to use the git log. It also lists the files that you've changed and those you still need to add or commit.

**Syntax**

1. $ git status



### 7) Git push Command

It is used to upload local repository content to a remote repository. Pushing is an act of transfer commits from your local repository to a remote repo. It's the complement to git fetch, but whereas fetching imports commits to local branches on comparatively pushing exports commits to remote branches. Remote branches are configured by using the git remote command. Pushing is capable of overwriting changes, and caution should be taken when pushing.

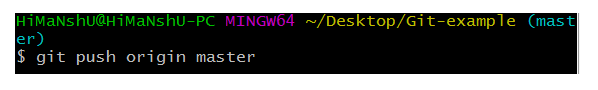
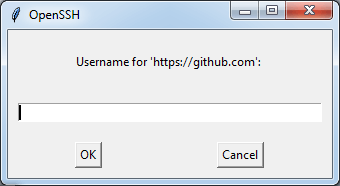
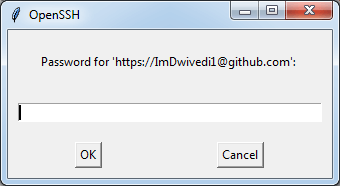
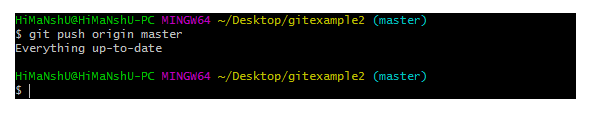
Git push command can be used as follows.

**Git push origin master**

This command sends the changes made on the master branch, to your remote repository.

**Syntax**

1. $ git push [variable name] master

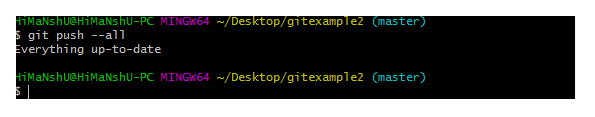
  
  
  


**Git push -all**

This command pushes all the branches to the server repository.

**Syntax**

1. $ git push --all

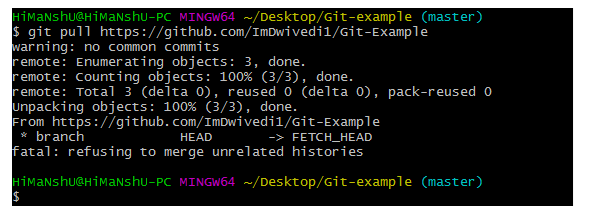


8) Git pull command

Pull command is used to receive data from GitHub. It fetches and merges changes on the remote server to your working directory.

**Syntax**

1. $ git pull URL

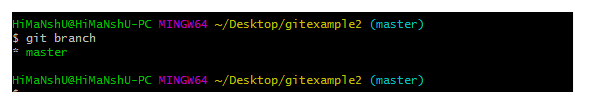


9) Git Branch Command

This command lists all the branches available in the repository.

**Syntax**

1. $ git branch



10) Git Merge Command

This command is used to merge the specified branch?s history into the current branch.

**Syntax**

1. $ git merge BranchName

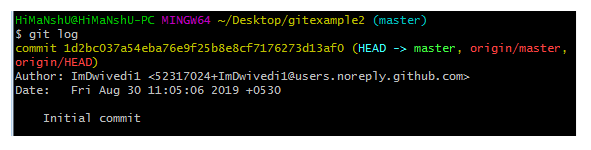


11) Git log Command

This command is used to check the commit history.

**Syntax**

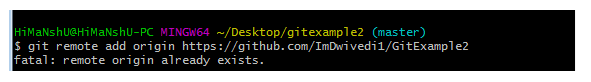
1. $ git log



By default, if no argument passed, Git log shows the most recent commits first. We can limit the number of log entries displayed by passing a number as an option, such as -3 to show only the last three entries.

12) Git remote Command

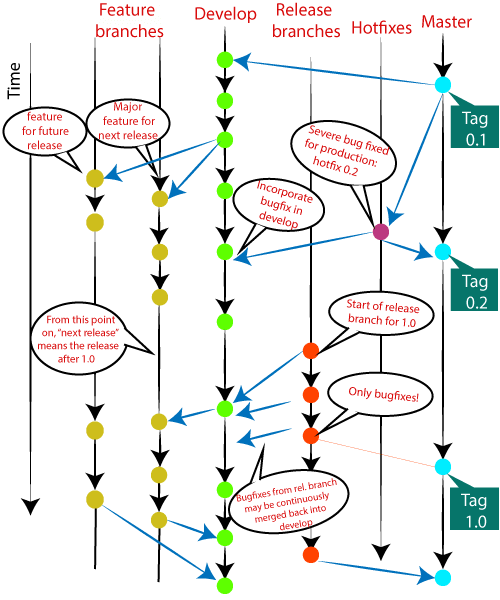
Git Remote command is used to connect your local repository to the remote server. This command allows you to create, view, and delete connections to other repositories. These connections are more like bookmarks rather than direct links into other repositories. This command doesn't provide real-time access to repositories.



Git Flow / Git Branching Model

Git flow is the set of guidelines that developers can follow when using Git. We cannot say these guidelines as rules. These are not the rules; it is a standard for an ideal project. So that a developer would easily understand the things.

It is referred to as **Branching Model** by the developers and works as a central repository for a project. Developers work and push their work to different branches of the main repository.



There are different types of branches in a project. According to the standard branching strategy and release management, there can be following types of branches:

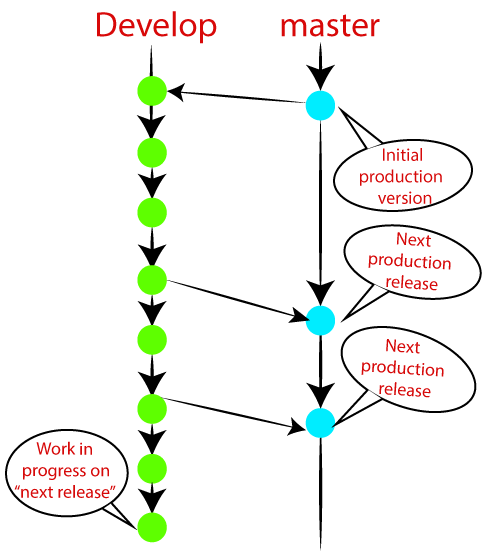
* **Master**
* **Develop**
* **Hotfixes**
* **Release branches**
* **Feature branches**

Every branch has its meaning and standard. Let's understand each branch and its usage.

## The Main Branches

Two of the branching model's branches are considered as main branches of the project. These branches are as follows:

* **master**
* **develop**



Master Branch

The master branch is the main branch of the project that contains all the history of final changes. Every developer must be used to the master branch. The master branch contains the source code of HEAD that always reflects a final version of the project.

Your local repository has its master branch that always up to date with the master of a remote repository.

It is suggested not to mess with the master. If you edited the master branch of a group project, your changes would affect everyone else, and very quickly, there will be merge conflicts.

Develop Branch

It is parallel to the master branch. It is also considered as the main branch of the project. This branch contains the latest delivered development changes for the next release. It has the final source code for the release. It is also called as a "**integration branch**."

When the develop branch reaches a stable point and is ready to release, it should be merged with master and tagged with a release version.

Supportive Branches

The development model needs a variety of supporting branches for the parallel development, tracking of features, assist in quick fixing and release, and other problems. These branches have a limited lifetime and are removed after the uses.

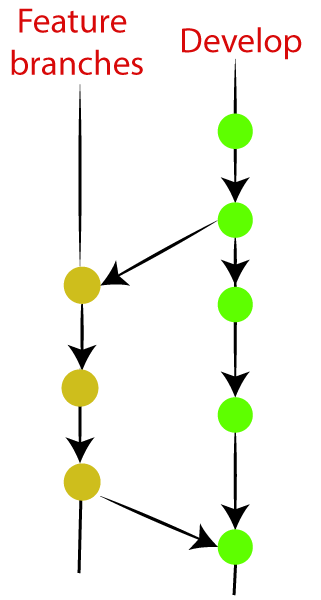
The different types of supportive branches, we may use are as follows:

* **Feature branches**
* **Release branches**
* **Hotfix branches**

Each of these branches is made for a specific purpose and have some merge targets. These branches are significant for a technical perspective.

Feature Branches

Feature branches can be considered as topic branches. It is used to develop a new feature for the next version of the project. The existence of this branch is limited; it is deleted after its feature has been merged with develop branch.



To learn how to create a Feature Branch [**Visit Here**](https://www.javatpoint.com/git-branch).

Release Branches

The release branch is created for the support of a new version release. Senior developers will create a release branch. The release branch will contain the predetermined amount of the feature branch. The release branch should be deployed to a staging server for testing.

Developers are allowed for minor bug fixing and preparing meta-data for a release on this branch. After all these tasks, it can be merged with the develop branch.

When all the targeted features are created, then it can be merged with the develop branch. Some usual standard of the release branch are as follows:

* Generally, senior developers will create a release branch.
* The release branch will contain the predetermined amount of the feature branch.
* The release branch should be deployed to a staging server for testing.
* Any bugs that need to be improved must be addressed at the release branch.
* The release branch must have to be merged back into developing as well as the master branch.
* After merging, the release branch with the develop branch must be tagged with a version number.

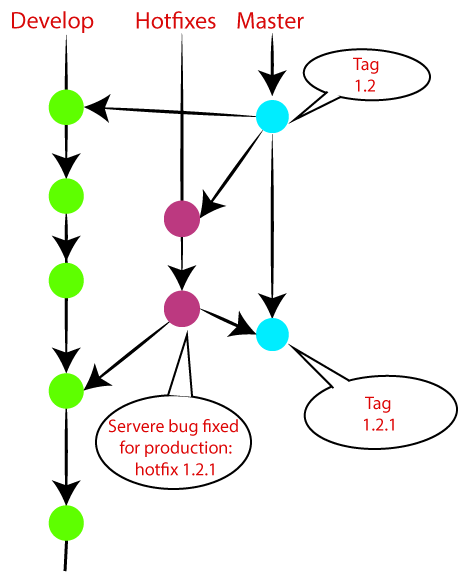
To create a release branch, visit [**Git Branching**](https://www.javatpoint.com/git-branch).

To tag branch after merging the release branch, Visit [**Git tag**](https://www.javatpoint.com/git-tags).

Hotfix Branches

Hotfix branches are similar to Release branches; both are created for a new production release.

The hotfix branches arise due to immediate action on the project. In case of a critical bug in a production version, a hotfix branch may branch off in your project. After fixing the bug, this branch can be merged with the master branch with a tag.



# **Git Cheat Sheet**

### 1. Git configuration

* **Git config**  
  Get and set configuration variables that control all facets of how Git looks and operates.  
  **Set the name:**  
  $ git config --global user.name "User name"  
  **Set the email:**  
  $ git config --global user.email "himanshudubey481@gmail.com"  
  **Set the default editor:**  
  $ git config --global core.editor Vim  
  **Check the setting:**  
  $ git config -list
* **Git alias**  
  **Set up an alias** for each command:  
  $ git config --global alias.co checkout  
  $ git config --global alias.br branch  
  $ git config --global alias.ci commit  
  $ git config --global alias.st status

2. Starting a project

* **Git init**  
  **Create a local repository:**  
  $ git init
* **Git clone**  
  **Make a local copy** of the server repository.  
  $ git clone

3. Local changes

* **Git add**  
  **Add a file** to staging (Index) area:  
  $ git add Filename  
  **Add all files** of a repo to staging (Index) area:  
  $ git add\*
* **Git commit**  
  **Record** or snapshots the file permanently in the version history **with a message**.  
  $ git commit -m " Commit Message"

4. Track changes

* **Git diff**  
  Track the changes that have not been staged: $ git diff  
  Track the changes that have staged but not committed:  
  $ git diff --staged  
  Track the changes after committing a file:  
  $ git diff HEAD  
  Track the changes between two commits:  
  $ git diff Git Diff Branches:  
  $ git diff < branch 2>
* **Git status**  
  Display the state of the working directory and the staging area.  
  $ git status
* **Git show Shows objects:**  
  $ git show

5. Commit History

* **Git log**  
  Display the most recent commits and the status of the head:  
  $ git log  
  Display the output as one commit per line:  
  $ git log -oneline  
  Displays the files that have been modified:  
  $ git log -stat  
  Display the modified files with location:  
  $ git log -p
* **Git blame**  
  Display the modification on each line of a file:  
  $ git blame <file name>

6. Ignoring files

* **.gitignore**  
  Specify intentionally untracked files that Git should ignore. Create .gitignore:  
  $ touch .gitignore List the ignored files:  
  $ git ls-files -i --exclude-standard

7. Branching

* **Git branch Create branch:**  
  $ git branch List Branch:  
  $ git branch --list Delete a Branch:  
  $ git branch -d Delete a remote Branch:  
  $ git push origin -delete Rename Branch:  
  $ git branch -m
* **Git checkout**  
  Switch between branches in a repository.  
  Switch to a particular branch:  
  $ git checkout  
  Create a new branch and switch to it:  
  $ git checkout -b Checkout a Remote branch:  
  $ git checkout
* **Git stash**  
  Switch branches without committing the current branch. Stash current work:  
  $ git stash  
  Saving stashes with a message:  
  $ git stash save ""  
  Check the stored stashes:  
  $ git stash list  
  Re-apply the changes that you just stashed:  
  $ git stash apply  
  Track the stashes and their changes:  
  $ git stash show  
  Re-apply the previous commits:  
  $ git stash pop  
  Delete a most recent stash from the queue:  
  $ git stash drop  
  Delete all the available stashes at once:  
  $ git stash clear  
  Stash work on a separate branch:  
  $ git stash branch
* **Git cherry pic**  
  Apply the changes introduced by some existing commit:  
  $ git cherry-pick

8. Merging

* **Git merge**  
  Merge the branches:  
  $ git merge  
  Merge the specified commit to currently active branch:  
  $ git merge
* **Git rebase**  
  Apply a sequence of commits from distinct branches into a final commit.  
  $ git rebase  
  Continue the rebasing process:  
  $ git rebase -continue Abort the rebasing process:  
  $ git rebase --skip
* **Git interactive rebase**  
  Allow various operations like edit, rewrite, reorder, and more on existing commits.  
  $ git rebase -i

9. Remote

* **Git remote**  
  Check the configuration of the remote server:  
  $ git remote -v  
  Add a remote for the repository:  
  $ git remote add Fetch the data from the remote server:  
  $ git fetch  
  Remove a remote connection from the repository:  
  $ git remote rm  
  Rename remote server:  
  $ git remote rename  
  Show additional information about a particular remote:  
  $ git remote show  
  Change remote:  
  $ git remote set-url
* **Git origin master**  
  Push data to the remote server:  
  $ git push origin master Pull data from remote server:  
  $ git pull origin master

10. Pushing Updates

* **Git push**  
  Transfer the commits from your local repository to a remote server. Push data to the remote server:  
  $ git push origin master Force push data:  
  $ git push -f  
  Delete a remote branch by push command:  
  $ git push origin -delete edited

11. Pulling updates

* **Git pull**  
  Pull the data from the server:  
  $ git pull origin master  
  Pull a remote branch:  
  $ git pull
* **Git fetch**  
  Download branches and tags from one or more repositories. Fetch the remote repository:  
  $ git fetch< repository Url> Fetch a specific branch:  
  $ git fetch  
  Fetch all the branches simultaneously:  
  $ git fetch -all  
  Synchronize the local repository:  
  $ git fetch origin

12. Undo changes

* **Git revert**  
  Undo the changes:  
  $ git revert  
  Revert a particular commit:  
  $ git revert
* **Git reset**  
  Reset the changes:  
  $ git reset -hard  
  $ git reset -soft:  
  $ git reset --mixed

13. Removing files

* **Git rm**  
  Remove the files from the working tree and from the index:  
  $ git rm <file Name>  
  Remove files from the Git But keep the files in your local repository:  
  $ git rm --cached