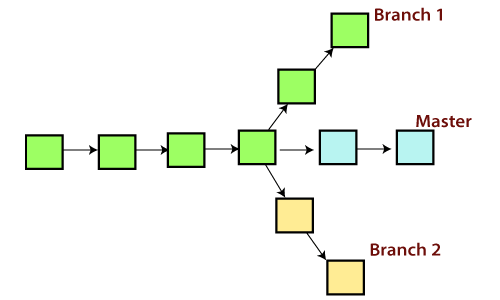
# Git Branch

A branch is a version of the repository that diverges from the main working project. It is a feature available in most modern version control systems. A Git project can have more than one branch. These branches are a pointer to a snapshot of your changes. When you want to add a new feature or fix a bug, you spawn a new branch to summarize your changes. So, it is complex to merge the unstable code with the main code base and also facilitates you to clean up your future history before merging with the main branch.



## Git Master Branch

The master branch is a default branch in Git. It is instantiated when first commit made on the project. When you make the first commit, you're given a master branch to the starting commit point. When you start making a commit, then master branch pointer automatically moves forward. A repository can have only one master branch.

Master branch is the branch in which all the changes eventually get merged back. It can be called as an official working version of your project.

## Operations on Branches

We can perform various operations on Git branches. The **git branch command** allows you to **create**, **list**, **rename** and **delete** branches. Many operations on branches are applied by git checkout and git merge command. So, the git branch is tightly integrated with the **git checkout** and **git merge commands**.

**The Operations that can be performed on a branch:**

### Create Branch

You can create a new branch with the help of the **git branch** command. This command will be used as:

**Syntax:**

1. $ git branch  <branch name>

**Output:**

IMG_257

This command will create the **branch B1** locally in Git directory.

### List Branch

You can List all of the available branches in your repository by using the following command.

Either we can use **git branch - list** or **git branch** command to list the available branches in the repository.

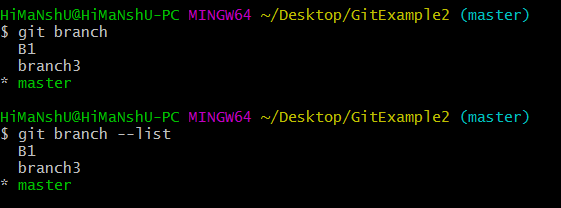
**Syntax:**

1. $ git branch --list

**or**

1. $ git branch

**Output:**



Here, both commands are listing the available branches in the repository. The symbol \* is representing currently active branch.

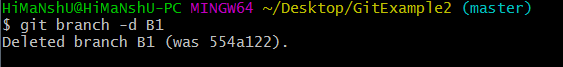
### Delete Branch

You can delete the specified branch. It is a safe operation. In this command, Git prevents you from deleting the branch if it has unmerged changes. Below is the command to do this.

**Syntax:**

1. $ git branch -d<branch name>

**Output:**



This command will delete the existing branch B1 from the repository.

The **git branch d** command can be used in two formats. Another format of this command is **git branch D**. The '**git branch D**' command is used to delete the specified branch.

1. $ git branch -D <branch name>

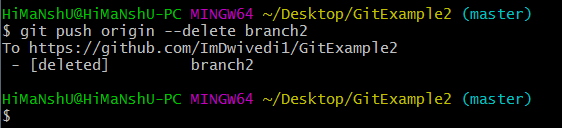
### Delete a Remote Branch

You can delete a remote branch from Git desktop application. Below command is used to delete a remote branch:

**Syntax:**

1. $ git push origin -delete <branch name>

**Output:**



As you can see in the above output, the remote branch named **branch2** from my GitHub account is deleted.

### Switch Branch

Git allows you to switch between the branches without making a commit. You can switch between two branches with the **git checkout** command. To switch between the branches, below command is used:

1. $ git checkout<branch name>

**Switch from master Branch**

You can switch from master to any other branch available on your repository without making any commit.

**Syntax:**

1. $ git checkout <branch name>

**Output:**

IMG_261

As you can see in the output, branches are switched from **master** to **branch4** without making any commit.

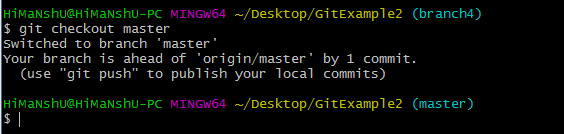
**Switch to master branch**

You can switch to the master branch from any other branch with the help of below command.

**Syntax:**

1. $ git branch -m master

**Output:**



As you can see in the above output, branches are switched from **branch1 to master** without making any commit.

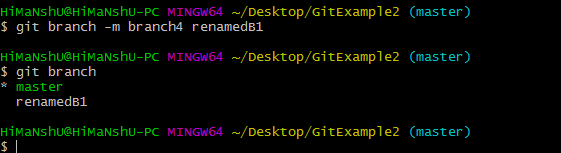
### Rename Branch

We can rename the branch with the help of the **git branch** command. To rename a branch, use the below command:

**Syntax:**

1. $ git branch -m <old branch name><new branch name>

**Output:**



As you can see in the above output, **branch4** renamed as **renamedB1**.

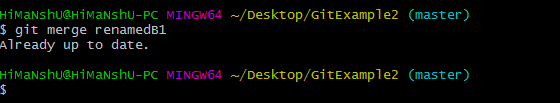
### Merge Branch

Git allows you to merge the other branch with the currently active branch. You can merge two branches with the help of **git merge** command. Below command is used to merge the branches:

**Syntax:**

1. $ git merge <branch name>

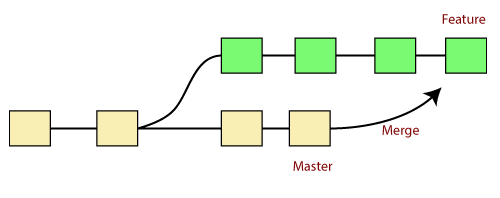
**Output:**



From the above output, you can see that **the master** branch **merged** with **renamedB1**. Since I have made no-commit before merging, so the output is showing as already up to date.

# Git Merge and Merge Conflict

In Git, the merging is a procedure to connect the forked history. It joins two or more development history together. The git merge command facilitates you to take the data created by git branch and integrate them into a single branch. Git merge will associate a series of commits into one unified history. Generally, git merge is used to combine two branches.



It is used to maintain distinct lines of development; at some stage, you want to merge the changes in one branch. It is essential to understand how merging works in Git.

In the above figure, there are two branches **master** and **feature**. We can see that we made some commits in both functionality and master branch, and merge them. It works as a pointer. It will find a common base commit between branches. Once Git finds a shared base commit, it will create a new "merge commit." It combines the changes of each queued merge commit sequence.

## The "git merge" command

The git merge command is used to merge the branches.

The syntax for the git merge command is as:

1. $ git merge <query>

It can be used in various context. Some are as follows:

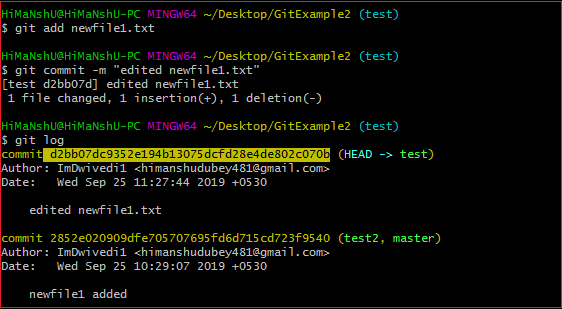
**Scenario1: To merge the specified commit to currently active branch:**

Use the below command to merge the specified commit to currently active branch.

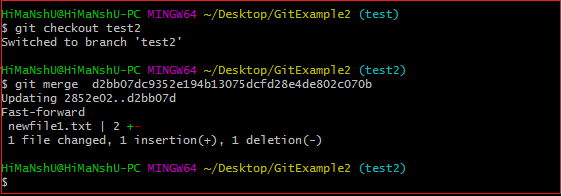
1. $ git merge <commit>

The above command will merge the specified commit to the currently active branch. You can also merge the specified commit to a specified branch by passing in the branch name in <commit>. Let's see how to commit to a currently active branch.

See the below example. I have made some changes in my project's file **newfile1.txt** and committed it in my **test** branch.



Copy the particular commit you want to merge on an active branch and perform the merge operation. See the below output:



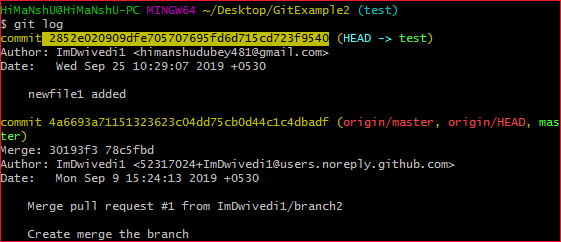
In the above output, we have merged the previous commit in the active branch test2.

**Scenario2: To merge commits into the master branch:**

To merge a specified commit into master, first discover its commit id. Use the log command to find the particular commit id.

1. $git log

See the below output:



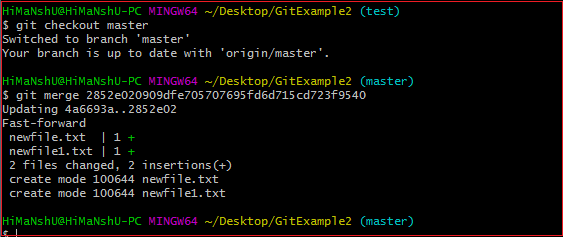
To merge the commits into the master branch, switch over to the master branch.

1. $ git checkout master

Now, Switch to branch 'master' to perform merging operation on a commit. Use the git merge command along with master branch name. The syntax for this is as follows:

1. $ git merge master

See the below output:



As shown in the above output, the commit for the commit id 2852e020909dfe705707695fd6d715cd723f9540 has merged into the master branch. Two files have changed in master branch. However, we have made this commit in the **test** branch. So, it is possible to merge any commit in any of the branches.

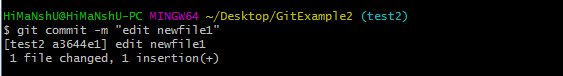
Open new files, and you will notice that the new line that we have committed to the test branch is now copied on the master branch.

**Scenario 3: Git merge branch.**

Git allows merging the whole branch in another branch. Suppose you have made many changes on a branch and want to merge all of that at a time. Git allows you to do so. See the below example:

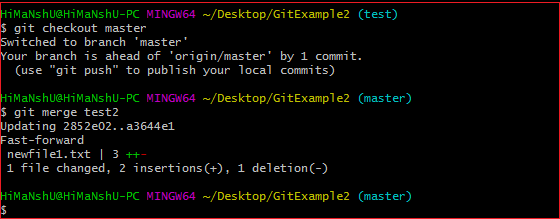
IMG_261

In the given output, I have made changes in newfile1 on the test branch. Now, I have committed this change in the test branch.



Now, switch to the desired branch you want to merge. In the given example, I have switched to the master branch. Perform the below command to merge the whole branch in the active branch.

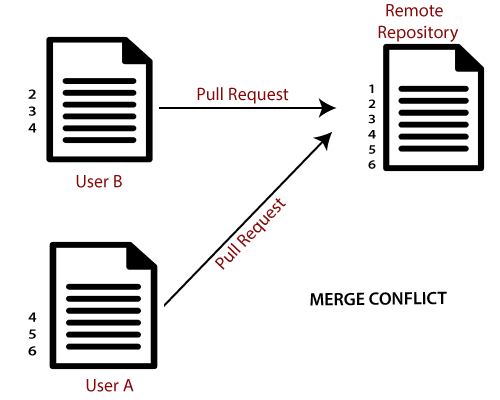
1. $ git merge <branchname>



As you can see from the given output, the whole commits of branch test2 have merged to branch master.

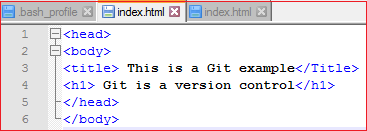
## Git Merge Conflict

When two branches are trying to merge, and both are edited at the same time and in the same file, Git won't be able to identify which version is to take for changes. Such a situation is called merge conflict. If such a situation occurs, it stops just before the merge commit so that you can resolve the conflicts manually.



Let's understand it by an example.

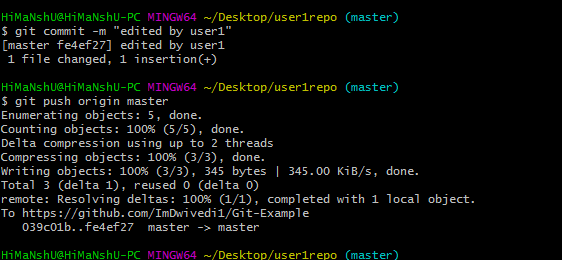
Suppose my remote repository has cloned by two of my team member **user1** and **user2**. The user1 made changes as below in my projects index file.



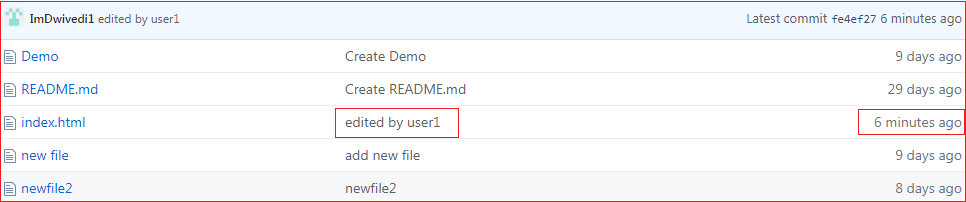
Update it in the local repository with the help of git add command.

IMG_266

Now commit the changes and update it with the remote repository. See the below output:

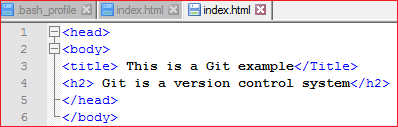


Now, my remote repository will look like this:

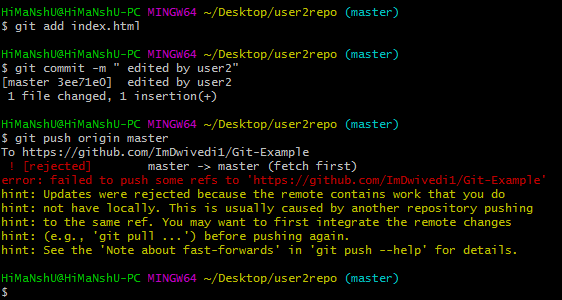


It will show the status of the file like edited by whom and when.

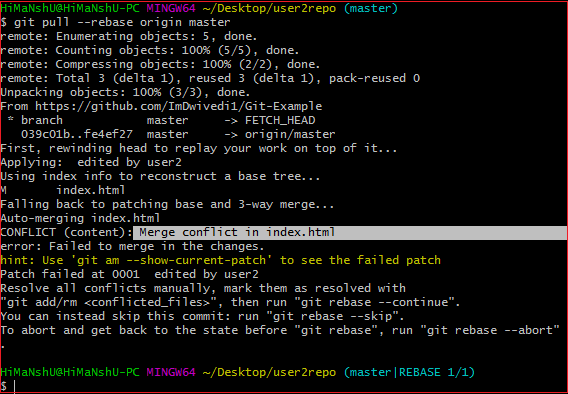
Now, at the same time, **user2** also update the index file as follows.



User2 has added and committed the changes in the local repository. But when he tries to push it to remote server, it will throw errors. See the below output:



In the above output, the server knows that the file is already updated and not merged with other branches. So, the push request was rejected by the remote server. It will throw an error message like **[rejected] failed to push some refs to <remote URL>**. It will suggest you to pull the repository first before the push. See the below command:



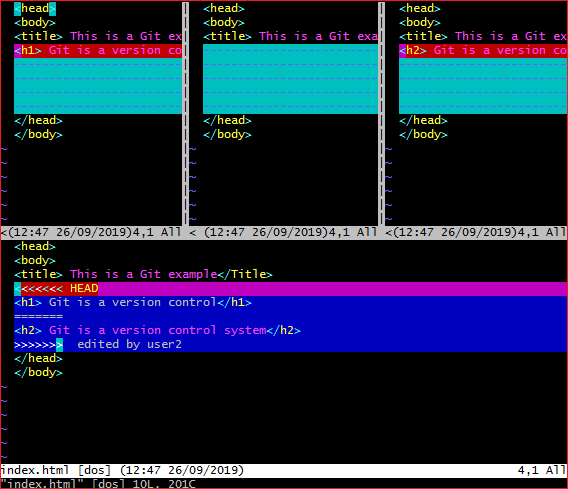
In the given output, git rebase command is used to pull the repository from the remote URL. Here, it will show the error message like **merge conflict in <filename>**.

## Resolve Conflict:

To resolve the conflict, it is necessary to know whether the conflict occurs and why it occurs. Git merge tool command is used to resolve the conflict. The merge command is used as follows:

1. $ git mergetool

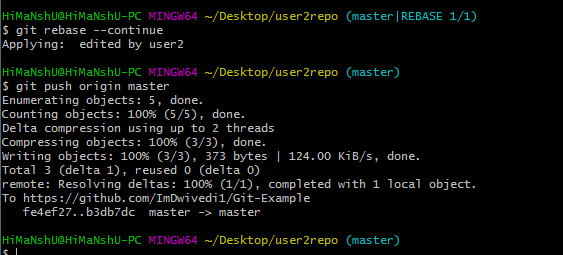
In my repository, it will result in:



The above output shows the status of the conflicted file. To resolve the conflict, enter in the insert mode by merely pressing **I key** and make changes as you want. Press the **Esc key**, to come out from insert mode. Type the: **w!** at the bottom of the editor to save and exit the changes. To accept the changes, use the rebase command. It will be used as follows:

1. $ git rebase --continue

Hence, the conflict has resolved. See the below output:



In the above output, the conflict has resolved, and the local repository is synchronized with a remote repository.

To see that which is the first edited text of the merge conflict in your file, search the file attached with conflict marker **<<<<<<<**. You can see the changes from the **HEAD** or base branch after the line **<<<<<<< HEAD** in your text editor. Next, you can see the divider like **=======**. It divides your changes from the changes in the other branch, **followed by >>>>>>> BRANCH-NAME**. In the above example, user1 wrote "<h1> Git is a version control</h1>" in the base or HEAD branch and user2 wrote "<h2> Git is a version control</h2>".

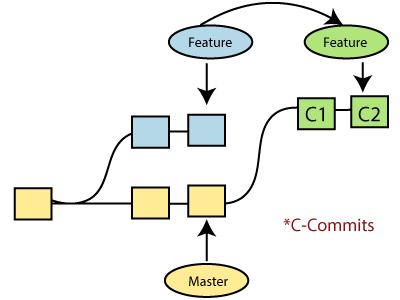
Decide whether you want to keep only your branch's changes or the other branch's changes, or create a new change. Delete the conflict markers **<<<<<<<, =======, >>>>>>>** and create final changes you want to merge.

# Git Rebase

Rebasing is a process to reapply commits on top of another base trip. It is used to apply a sequence of commits from distinct branches into a final commit. It is an alternative of git merge command. It is a linear process of merging.

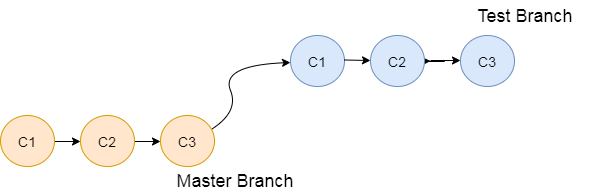
In Git, the term rebase is referred to as the process of moving or combining a sequence of commits to a new base commit. Rebasing is very beneficial and it visualized the process in the environment of a feature branching workflow.

It is good to rebase your branch before merging it.



Generally, it is an alternative of git merge command. Merge is always a forward changing record. Comparatively, rebase is a compelling history rewriting tool in git. It merges the different commits one by one.

Suppose you have made three commits in your master branch and three in your other branch named test. If you merge this, then it will merge all commits in a time. But if you rebase it, then it will be merged in a linear manner. Consider the below image:



The above image describes how git rebase works. The three commits of the master branch are merged linearly with the commits of the test branch.

Merging is the most straightforward way to integrate the branches. It performs a three-way merge between the two latest branch commits.

## How to Rebase

When you made some commits on a feature branch (test branch) and some in the master branch. You can rebase any of these branches. Use the git log command to track the changes (commit history). Checkout to the desired branch you want to rebase. Now perform the rebase command as follows:

**Syntax:**

1. $git rebase <branch name>

If there are some conflicts in the branch, resolve them, and perform below commands to continue changes:

1. $ git status

It is used to check the status,

1. $git rebase --continue

The above command is used to continue with the changes you made. If you want to skip the change, you can skip as follows:

1. $ git rebase --skip

When the rebasing is completed. Push the repository to the origin. Consider the below example to understand the git merge command.

Suppose that you have a branch say **test2** on which you are working. You are now on the test2 branch and made some changes in the project's file **newfile1.txt**.

Add this file to repository:

1. $ git add newfile1.txt

Now, commit the changes. Use the below command:

1. $ git commit -m "new commit for test2 branch."

The output will look like:

[test2 a835504] new commitfor test2 branch

1 file changed, 1 insertion(+)

Switch the branch to master:

1. $ git checkout master

**Output:**

Switched to branch 'master.'

Your branch is up to date with 'origin/master.'

Now you are on the master branch. I have added the changes to my file, says **newfile.txt**. The below command is used to add the file in the repository.

1. $ git add newfile.txt

Now commit the file for changes:

1. $ git commit -m " new commit made on the master branch."

**Output:**

[master 7fe5e7a] new commit made on master

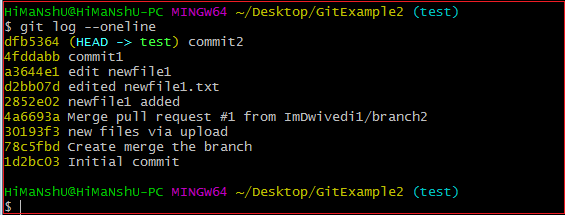
1 file changed, 1 insertion(+)

HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/GitExample2 (master)

To check the log history, perform the below command.

1. $ git log --oneline

**Output:**



As we can see in the log history, there is a new commit in the master branch. If I want to rebase my test2 branch, what should I do? See the below rebase branch scenario:

## Rebase Branch

If we have many commits from distinct branches and want to merge it in one. To do so, we have two choices either we can merge it or rebase it. It is good to rebase your branch.

From the above example, we have committed to the master branch and want to rebase it on the test2 branch. Let's see the below commands:

1. $ git checkout test2

This command will switch you on the test2 branch from the master.

**Output:**

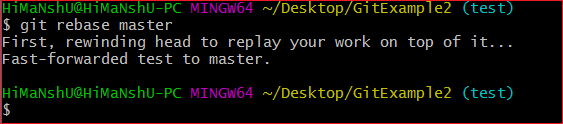
Switched to branch 'test2.'

Now you are on the test2 branch. Hence, you can rebase the test2 branch with the master branch. See the below command:

1. $ git rebase master

This command will rebase the test2 branch and will show as **Applying: new commit on test2 branch**. Consider the below output:

**Output:**



## Git Interactive Rebase

Git facilitates with Interactive Rebase; it is a potent tool that allows various operations like **edit, rewrite, reorder,** and more on existing commits. Interactive Rebase can only be operated on the currently checked out branch. Therefore, set your local HEAD branch at the sidebar.

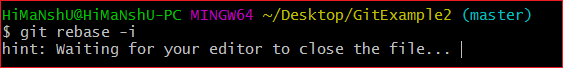
Git interactive rebase can be invoked with rebase command, just type **-i** along with rebase command. Here '**i**' stands for interactive. Syntax of this command is given below:

**Syntax:**

1. $ git rebase -i

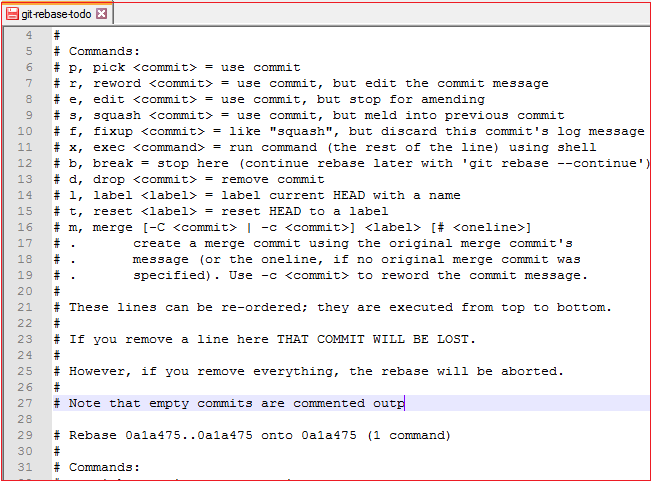
It will list all the available interactive options.

**Output:**



After the given output, it will open an editor with available options. Consider the below output:

**Output:**



When we perform the git interactive rebase command, it will open your default text editor with the above output.

The options it contains are listed below:

* Pick
* Reword
* Edit
* Squash
* Fixup
* Exec
* Break
* Drop
* Label
* Reset
* Merge

The above options perform their specific tasks with git-rebase. Let's understand each of these options in brief.

**Pick (-p):**

Pick stands here that the commit is included. Order of the commits depends upon the order of the pick commands during rebase. If you do not want to add a commit, you have to delete the entire line.

**Reword (-r):**

The reword is quite similar to pick command. The reword option paused the rebase process and provides a chance to alter the commit message. It does not affect any changes made by the commit.

**Edit (-e):**

The edit option allows for amending the commit. The amending means, commits can be added or changed entirely. We can also make additional commits before rebase continue command. It allows us to split a large commit into the smaller commit; moreover, we can remove erroneous changes made in a commit.

**Squash (-s):**

The squash option allows you to combine two or more commits into a single commit. It also allows us to write a new commit message for describing the changes.

**Fixup (-f):**

It is quite similar to the squash command. It discarded the message of the commit to be merged. The older commit message is used to describe both changes.

**Exec (-x):**

The exec option allows you to run arbitrary shell commands against a commit.

**Break (-b):**

The break option stops the rebasing at just position. It will continue rebasing later with '**git rebase --continue**' command.

**Drop (-d):**

The drop option is used to remove the commit.

**Label (-l):**

The label option is used to mark the current head position with a name.

**Reset (-t):**

The reset option is used to reset head to a label.

## GitMerge vs. Rebase

It is a most common puzzling question for the git user's that when to use merge command and when to use rebase. Both commands are similar, and both are used to merge the commits made by the different branches of a repository.

Rebasing is not recommended in a shared branch because the rebasing process will create inconsistent repositories. For individuals, rebasing can be more useful than merging. If you want to see the complete history, you should use the merge. Merge tracks the entire history of commits, while rebase rewrites a new one.

Git rebase commands said as an alternative of git merge. However, they have some key differences:

|  |  |
| --- | --- |
| **Git Merge** | **Git Rebase** |
| Merging creates a final commit at merging. | Git rebase does not create any commit at rebasing. |
| It merges all commits as a single commit. | It creates a linear track of commits. |
| It creates a graphical history that might be a bit complex to understand. | It creates a linear history that can be easily understood. |
| It is safe to merge two branches. | Git "rebase" deals with the severe operation. |
| Merging can be performed on both public and private branches. | It is the wrong choice to use rebasing on public branches. |
| Merging integrates the content of the feature branch with the master branch. So, the master branch is changed, and feature branch history remains consistence. | Rebasing of the master branch may affect the feature branch. |
| Merging preserves history. | Rebasing rewrites history. |
| Git merge presents all conflicts at once. | Git rebase presents conflicts one by one. |

# Git Squash

In Git, the term squash is used to squash the previous commits into one. It is not a command; instead, it is a keyword. The squash is an excellent technique for group-specific changes before forwarding them to others. You can merge several commits into a single commit with the compelling interactive rebase command.

If you are a Git user, then you must have realized the importance of squashing a commit. Especially if you are an open-source contributor, then many times, you have to create a PR (pull request) with squashed commit. You can also squash commits if you have already created a PR.

Let's understand how to squash commits?

## Git Squash Commits

Being a responsible contributor to Git, it is necessary to make the collaboration process efficient and meaningful. Git allows some powerful collaboration tools in different ways. Git squash is one of the powerful tools that facilitate efficient and less painful collaboration.

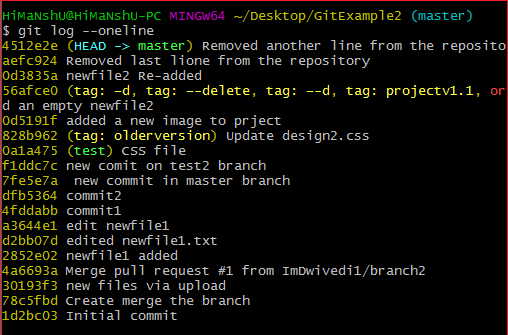
The squash is not any command; instead, it's one of many options available to you under git interactive rebases. The squash allows us to rewrite history. Suppose we have made many commits during the project work, squashing all the commits into a large commit is the right choice than pushing. Let's understand how to squash two commits.

**Step1: Check the commit history**

To check the commit history, run the below command:

1. $ git log --oneline

The given command will display the history in one line. We can track the history and choose the commits we want to squash. Consider the below output:

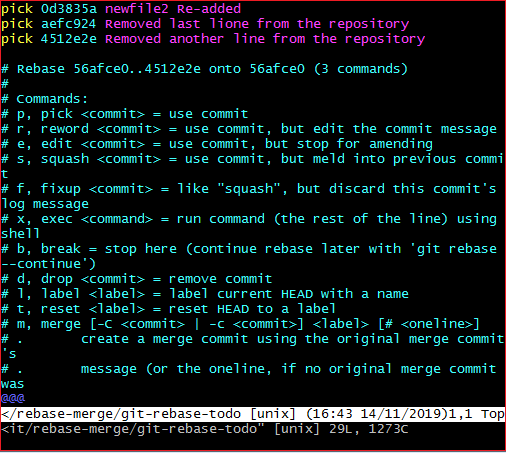


**Step 2: Choose the commits to squash.**

Suppose we want to squash the last commits. To squash commits, run the below command:

1. $ git rebase -i HEAD ~3

The above command will open your default text editor and will squash the last three commits. The editor will open as follows:

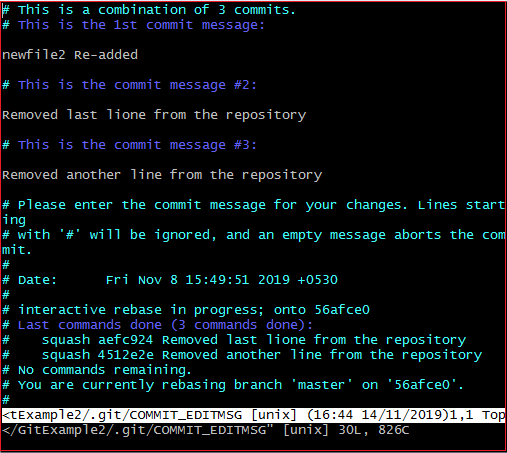


From the above image, we can see previous commits shown at the top of the editor. If we want to merge them into a single commit, then we have to **replace** the word **pick** with **the squash** on the top of the editor. To write on the editor, press '**i**' button to enter in **insert mode**. After editing the document, press the **:wq** to save and exit from the editor.

**Step 3: update the commits**

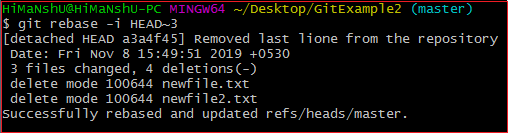
On pressing **enter** key, a new window of the text editor will be opened to confirm the commit. We can edit the commit message on this screen.

I am editing my first commit message because it will be a combination of all three commits. Consider the below image:



The above image is the editor screen to confirm the merging of commits. Here we can update the commit messages. To edit on this editor, press the '**i**' button for insert mode and edit the desired text. Press the **:wq** keys, to save and exit from the editor.

When we exit the editor, it will show the description of updates. Consider the below output:



The above output is listing the description of changes that have been made on the repository. Now, the commits have been squashed. Check the commit history for confirmation with the help of the git log. Consider the below output:



**Step 4: Push the squashed commit**

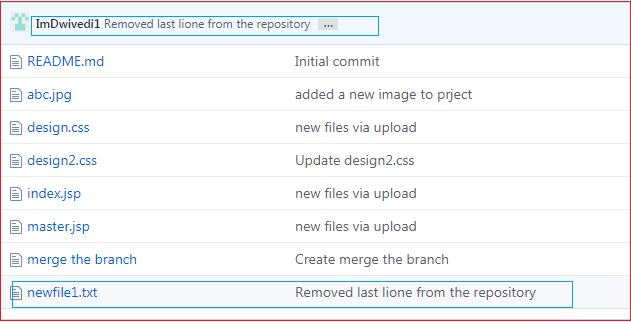
Now, we can push this squashed commit on the remote server. To push this squashed commit, run the below command:

1. $ git push origin master

Or

1. $ git push -f origin master

The above command will push the changes on the remote server. We can check this commit on our remote repository. Consider the below image:



As you can see from the above image. A new commit has been added to my remote repository.

## Drawbacks of Squashing

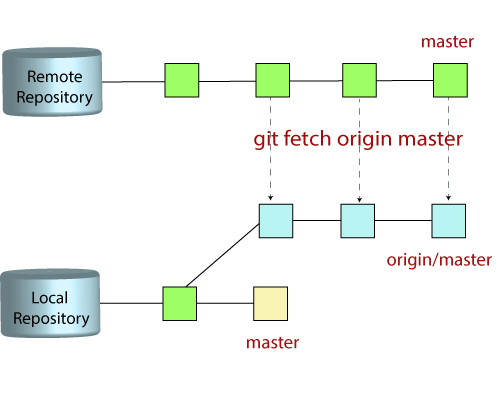
There are no significant drawbacks of squashing. But we can consider some facts that may affect the project. These facts are as follows:

The squashing commits, and rebasing changes the history of the repository. If any contributor does not pay attention to the updated history, then it may create conflict. I suggest a clean history because it is more valuable than another one. Although we can check the original history in the ref log.

There is another drawback, we may lose granularity because of squashing. Try to make minimum squashes while working with Git. So, if you are new on Git, then try to stay away from squash.

# Git Fetch

Git "fetch" Downloads commits, objects and refs from another repository. It fetches branches and tags from one or more repositories. It holds repositories along with the objects that are necessary to complete their histories to keep updated remote-tracking branches.



## The "git fetch"command

The "**git fetch**" **command** is used to pull the updates from remote-tracking branches. Additionally, we can get the updates that have been pushed to our remote branches to our local machines. As we know, a branch is a variation of our repositories main code, so the remote-tracking branches are branches that have been set up to pull and push from remote repository.

## How to fetch Git Repository

We can use fetch command with many arguments for a particular data fetch. See the below scenarios to understand the uses of fetch command.

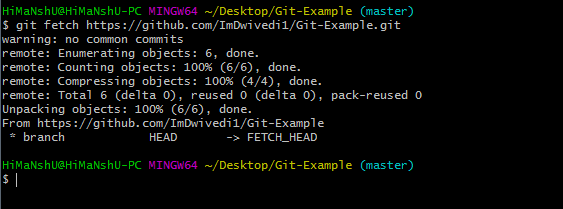
### Scenario 1: To fetch the remote repository:

We can fetch the complete repository with the help of fetch command from a repository URL like a pull command does. See the below output:

**Syntax:**

1. $ git fetch< repository Url>

**Output:**



In the above output, the complete repository has fetched from a remote URL.

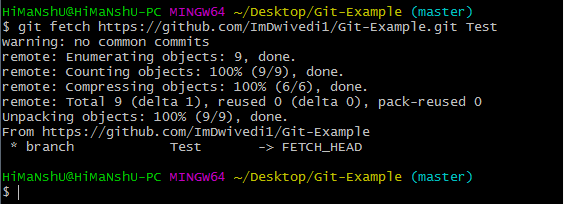
### Scenario 2: To fetch a specific branch:

We can fetch a specific branch from a repository. It will only access the element from a specific branch. See the below output:

**Syntax:**

1. $ git fetch <branch URL><branch name>

**Output:**



In the given output, the specific branch **test** has fetched from a remote URL.

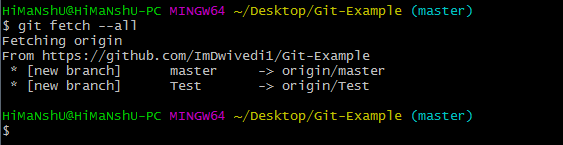
### Scenario 3: To fetch all the branches simultaneously:

The git fetch command allows to fetch all branches simultaneously from a remote repository. See the below example:

**Syntax:**

1. $ git fetch -all

**Output:**



In the above output, all the branches have fetched from the repository Git-Example.

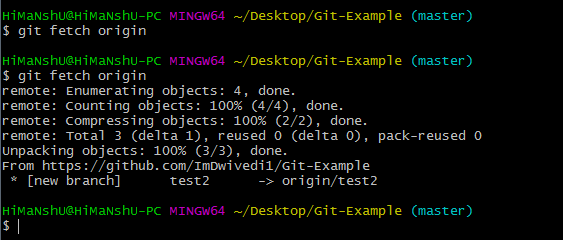
### Scenario 4: To synchronize the local repository:

Suppose, your team member has added some new features to your remote repository. So, to add these updates to your local repository, use the git fetch command. It is used as follows.

**Syntax:**

1. $ git fetch origin

**Output:**



In the above output, new features of the remote repository have updated to my local system. In this output, the branch **test2** and its objects are added to the local repository.

The git fetch can fetch from either a single named repository or URL or from several repositories at once. It can be considered as the safe version of the git pull commands.

The git fetch downloads the remote content but not update your local repo's working state. When no remote server is specified, by default, it will fetch the origin remote.

## Differences between git fetch and git pull

To understand the differences between fetch and pull, let's know the similarities between both of these commands. Both commands are used to download the data from a remote repository. But both of these commands work differently. Like when you do a git pull, it gets all the changes from the remote or central repository and makes it available to your corresponding branch in your local repository. When you do a git fetch, it fetches all the changes from the remote repository and stores it in a separate branch in your local repository. You can reflect those changes in your corresponding branches by merging.

So basically,

1. git pull = git fetch + git merge

### Git Fetch vs. Pull

Some of the key differences between both of these commands are as follows:

|  |  |
| --- | --- |
| **git fetch** | **git pull** |
| Fetch downloads only new data from a remote repository. | Pull is used to update your current HEAD branch with the latest changes from the remote server. |
| Fetch is used to get a new view of all the things that happened in a remote repository. | Pull downloads new data and directly integrates it into your current working copy files. |
| Fetch never manipulates or spoils data. | Pull downloads the data and integrates it with the current working file. |
| It protects your code from merge conflict. | In git pull, there are more chances to create the **merge conflict**. |
| It is better to use git fetch command with git merge command on a pulled repository. | It is not an excellent choice to use git pull if you already pulled any repository. |