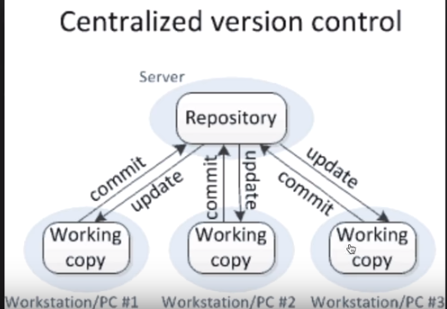
**Git & GitHub**

**Introduction:**

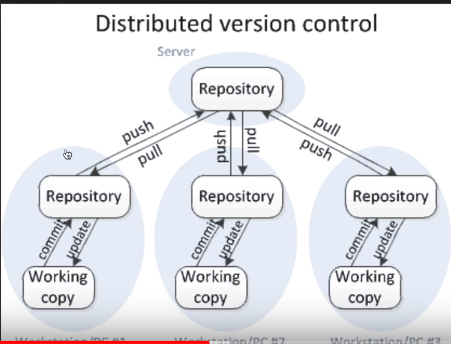
1. What is GIT?

* VCS - version control system
* To track changes in files / folders
* To collaborate in teams
* Free and open source

Git - Centralized VCS | Distributed VCS …?



Centralised VCS



Distributed VCS (DVCS)

**GIT = DVCS**

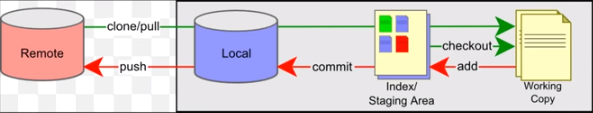
2. What is GIT HUB?

* website to upload your repositories online
* provides backup
* provides visual interface to your repo
* makes collaboration easier

3. Is GIT related to GIT HUB?

**GIT != GIT HUB**

4. Simple work flow of Git………..!!!



**Install git on windows:**

**Step - 1:** Check if git installed or not

**Step - 2:** Download and install git (while installing > if windows prompt is selected, we can use both bash & cmd; can use only bash if it is selected)

**Step - 3:** Add your project to git (goto project location which is to be added to git > right click > git bash here)

**Step - 4:** Check the following commands & can be used from git bash or from command prompt.

* git config --global user.email "abcd@gmail.com"
* git config --global user.name "usrnme"
* git --version
* git init
* git status
* git add (adding to staging area)
* git commit -m "any message"

Create an account for git hub & create a repository > **Repo1**:

Adding project to remote repository (i.e., github)

* git remote add origin https://github.com/doddam/Repo1.git > copy url from git hub
* git push -u origin master > pushes data into repository
* git log
* git --help

**Branching and Merging:**

1. What are branches?

2. How to create branch?

3. How to checkout branch?

4. How to merge branch to master?

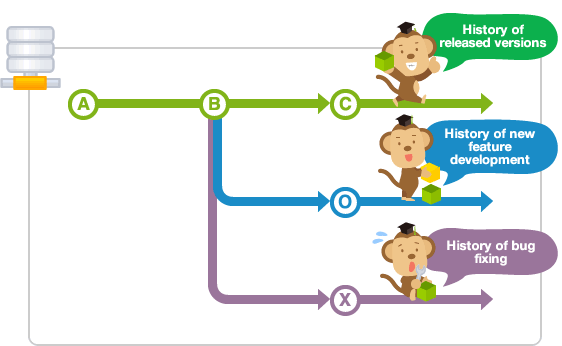
5. How to delete branch (local and remote)?

**About Branching and Merging:**

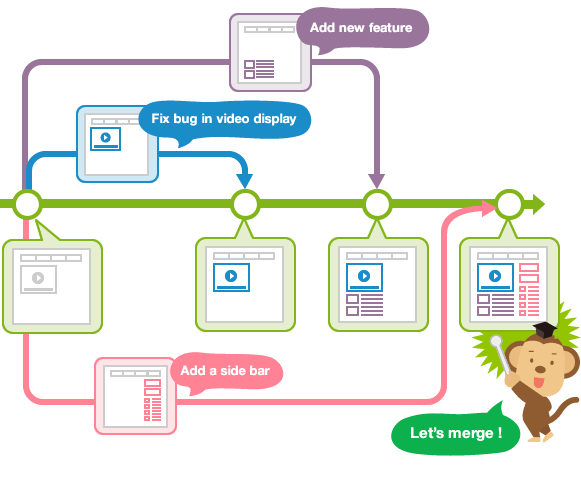
In a collaborative environment, it is common for several developers to share and work on the same source code. Some developers will be fixing bugs while others would be implementing new features. Therefore, there has got to be a manageable way to maintain different versions of the same code base.

This is where the branch function comes to the rescue. Branch allows each developer to branch out from the original code base and isolate their work from others. Another good thing about branch is that it helps Git to easily merge the versions later on.

Simply, it can be said as an **independent line of development** as one can take advantage of branch when working on new features or bug fixes as it helps to isolate your work from that of other team members.

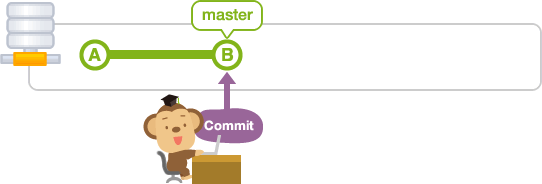


* Different branches can be merged into any one branch provided that they belong to the same repository.
* The diagram below illustrates how development can take place in parallel using branches.
* Changes in the primary branch or other branches will not affect your branch, unless you decide to pull the latest changes from those branches.
* It is a common practice to create a new branch for each task (eg. bug fixing, new features etc.), which is a good practice because it allows others to easily identify what changes to expect, and also for backtracking purposes to understand why a particular code change is implemented.



**Master branch:**

Upon making the first commit in a repository, Git will automatically create a master branch by default. Subsequent commits will go under the master branch until you decide to create and switch over to another branch.



**Create branch:**

Let's create a new branch with the name "issue1".

Use the branch command with a name to create a new branch with that name.

$ git branch <branchname>

Create a new branch named issue1.

$ git branch issue1

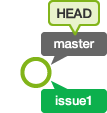
If you do not specify any parameters, the branch command will list all branches that correspond to this repository. The asterisk indicates the current active branch.

$ git branch

issue1

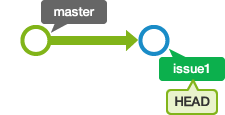
\* master

At this point, the history tree should look like this,



**Merge branches:**

Let's merge "issue1" with "master"



Use the merge command to merge branches.

$ git merge <commit>

By running the command above, the specified commit will be merged to the current active branch. Most of the time, you will want to merge a branch with the current active branch and you can do so by passing in the branch name in <commit>.

To merge commits into the master branch, let's now switch over to the master branch.

$ git checkout master

Switched to branch 'master'

# Delete branch:

Now that "issue1" has been successfully merged with "master", we can delete it.

We can delete a branch by calling the branch command and passing in the -d option, followed by the branch name.

$ git branch -d <branch-name>

Run the following command to delete "issue1".

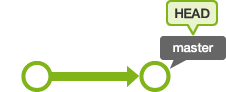
$ git branch -d issue1

Deleted branch issue1 (was b2b23c4).

We can verify that "issue1" has been deleted by calling "git branch". Only the master branch should be listed.

$ git branch

\* master



**Steps to be followed for branching & merging:**



Do not to modify the code in master branch, any changes need to be done to the code from git follow the below steps:

Create other branch > push the code or make changes required > test it and validate it > if ok > then merge this branch to master branch

git branch "branch name" > cmd to create a new branch

* git checkout "branch name" > to start working out with the branch
* touch test2.txt > create a new file
* git status
* git add . | git add “file name” | git add \*.\* (with all extensions)
* git commit -m "test2.txt is added"
* git push -u origin MyNewBranch

[test2.txt file is added to the new branch but not to the master branch,

open github and check that the branch is created,

open the project path in local system and check you can see test2.txt is added]

* git checkout master > test2.txt file is invisible
* git merge MyNewBranch > when you are merging new branch to master branch we need to check out to master branch and then merge it
* git push -u origin master

**Faced an error:**

! [rejected] master -> master (fetch first)

error: failed to push some refs to ''

hint: Updates were rejected because the remote contains work that you do

hint: not have locally. This is usually caused by another repository pushing

hint: to the same ref. You may want to first integrate the remote changes

hint: (e.g., 'git pull ...') before pushing again.

hint: See the 'Note about fast-forwards' in 'git push --help' for details.

**Solved it using the git command:**

* “git push origin master - -force | git push origin master –f”
* git branch -d "branch name" --> branch from local (system)
* git push origin --delete "branch name" --> delete from remote (github)

**Scenario -** *How to send email from GitHub*

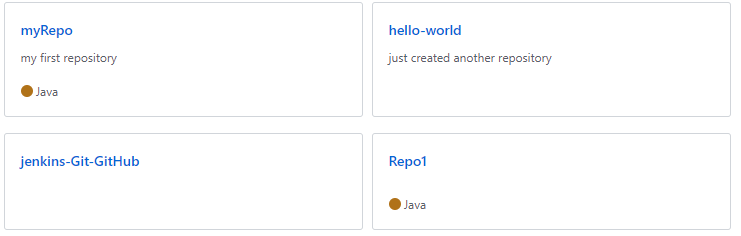
How to trigger notification email from GitHub whenever there is any change/commit in the project?

**Step-1:** GitHub > Repository > Settings > integration & services > add email

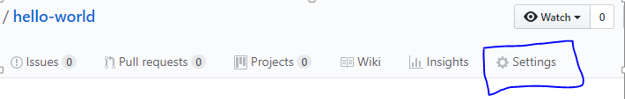
**Step-2:** Test and validate by making some change in the project

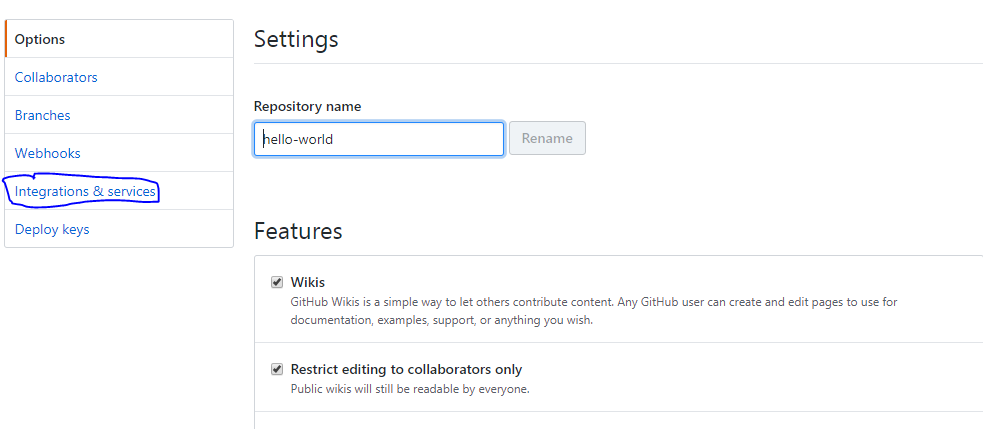
**Check the below screenshots:**

I have the following repositories in git hub,

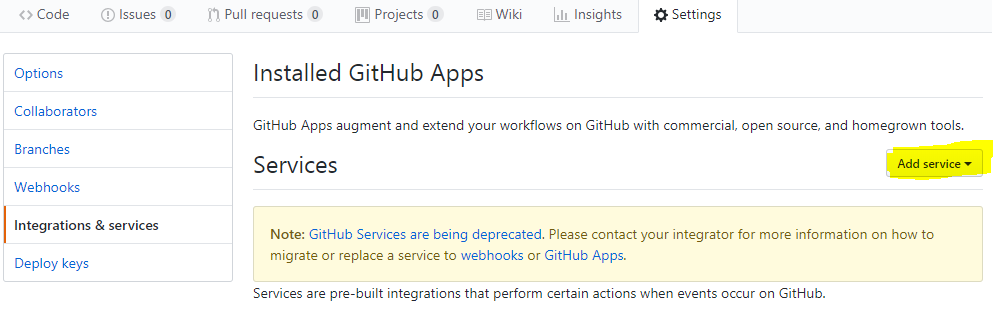


Click on any of the repository... I have done with “hello-world” repo> go to settings

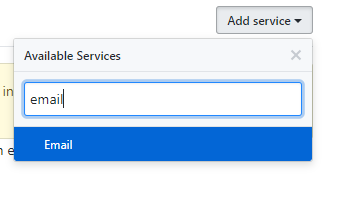


Goto options tab > select integrations & services 

Click on “Add service”

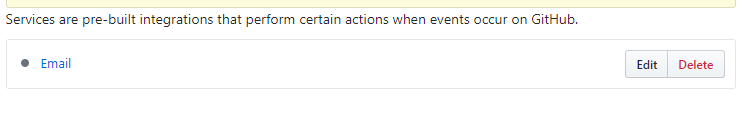


Search for the service “Email” in text box



Once you select the Email service > you need to fill the following with Email address in Address tab > Tick the checkbox - Send from author > click on Add service

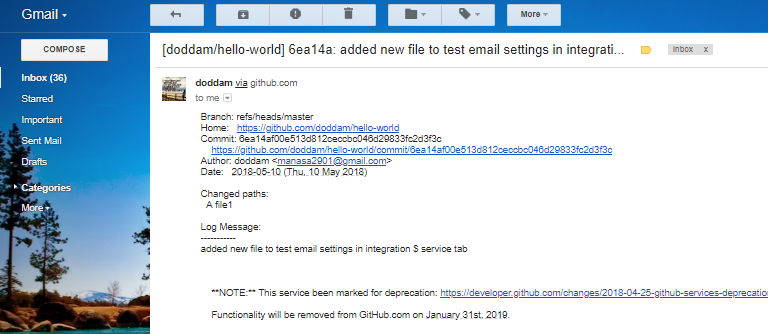
Now you can use the Email service once the service is added.



**Test the scenario whether service is running successfully or not:**

Make any changes to your repository just to test and validate > “hello-world”

I have added a new file in the repo > committed > check email that you have configured in settings > a notification email is triggered from GitHub as we have made a change in the project.



**Branches versus Tags**

The workspace is (almost always) associated with a branch, called master by default. When it is, a commit will automatically update the master reference to point to that new commit; in other words, branches are **mutable references** (changeable object).

**Tag:**

A tag, on the other hand, is created “to point to a specific commit” and thereafter does not change, even if the branch moves on. In other words, tags are **immutable references** (unchangeable object).

Tags are created using git tag are the base for the commit identifiers git describe creates. In another words, in Git you don't tag branches. You are tagging commits. It is correct to say that tag is just an annotated pointer to a commit.

**Annotated Tags:**

Git has two flavors of tags:

1. Annotated : lightweight tag
2. Non-annotated : can give a message and all notes available in description below completely stored as a git object in git repository

When using them, there is little difference between the two; both will allow you to refer to a specific commit in a repository.

An annotated tag creates an additional tag object in the Git repository, which allows you to store information associated with the tag itself. This may include release notes, the meta-information about the release, and optionally a signature to verify the authenticity of the commit to which it points.

**Step-1:** Open git bash and go to local directory, checkout the branch where you want to create the tag

git checkout <branch name>

git checkout master

**Step-2:** Create tag with some name

git tag <tag name>

git tag v1.0

**Step-3:** Check whether tag is created or not



**Step-4:** Creating annotated tag and check for git tag command.

git tag -a <tag name> -m “message”

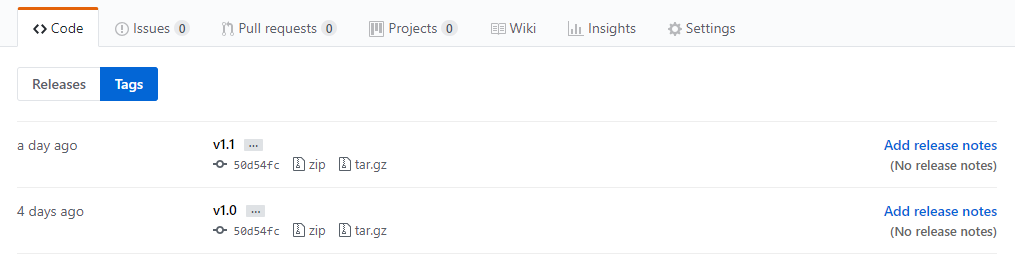
git tag -a v1.1 -m "tag for release version 1.1"



Where, v1.0 – a lightweight tag v1.1 – annotated tag

Once tags are created, you can view them in the repository.

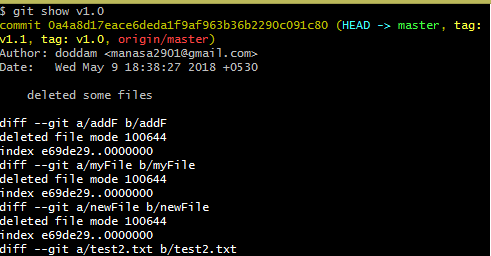
Go torepository > click on releases > check the below screenshot



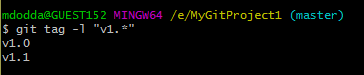
Click on that zip file > download > open and check the code that is taken backup.

**Commands to display or show tags:**

1. git tag
2. git show v1.0
3. git tag - - l “v1.\*”



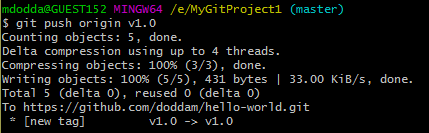
Displaying all tags using wild cards:



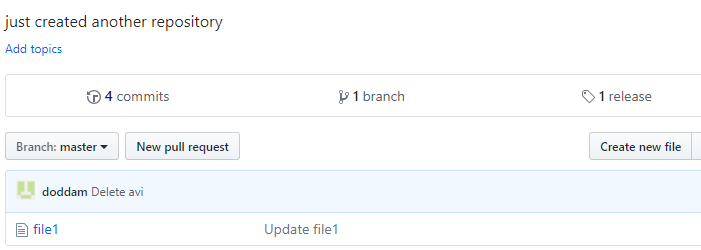
**How to push tags to remote?**

**Commands to push tags to remote:**

1. git push origin v1.0
2. git push origin - -tags
3. git push - - tags



Goto the repository > refresh it > check in releases section



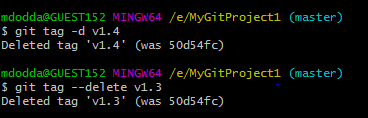
Can use second and third option to push all tags at a time to remote.



**Delete tags from local:**

git tag –d v1.4 (or)

git tag - -delete v1.3



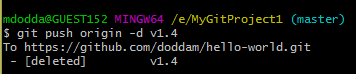
Tags are deleted from local but still tags existed on remote repository.

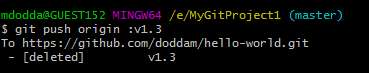
**Delete tags from remote:**

git push origin –d v1.4

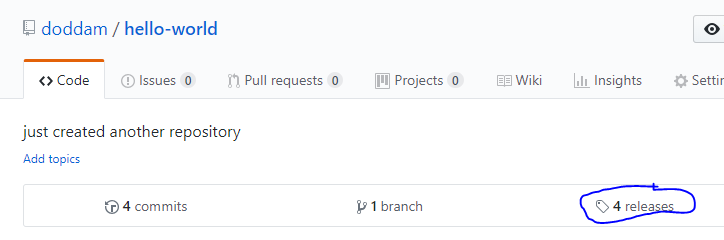
git push origin - -delete v1.4

git push origin :v1.4





Now check the repository tags are deleted on remote:



**To delete multiple tags:**

git tag –d v1.0 v1.1 (local)

git push origin –d v1.0 v1.1 (remote)

**­Let’s understand some basic terms:**

**SHA-1 (short for *Secure Hash Algorithm 1*):** is one of several [cryptographic hash functions](https://www.lifewire.com/cryptographic-hash-function-2625832).

SHA-1 is most often used to verify that a [file](https://www.lifewire.com/what-is-a-file-2625878) has been unaltered. This is done by producing a [checksum](https://www.lifewire.com/what-does-checksum-mean-2625825) before the file has been transmitted, and then again once it reaches its destination.

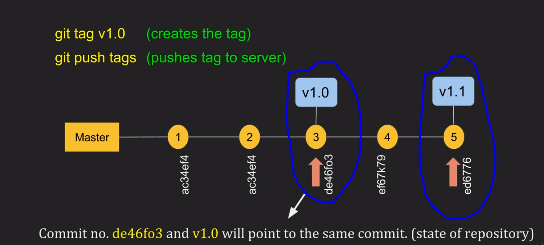
The transmitted file can be considered genuine only if **both checksums are identical**.

C**ryptographic hash function:** It is a kind of algorithm that can be run on a piece of data, like an individual [file](https://www.lifewire.com/what-is-a-file-2625878) or a password, to produce a value called a checksum.

The main use of a cryptographic hash function is to verify the authenticity of a piece of data. Two files can be assured to be identical only if the checksums generated from each file, using the same cryptographic hash function, are identical.

**Checksum:** is the outcome of running an algorithm, called a [cryptographic hash function](https://www.lifewire.com/cryptographic-hash-function-2625832), on a piece of data, usually a single [file](https://www.lifewire.com/what-is-a-file-2625878).

**Understanding the overall scenario:**



Tagging scenario is explained here,

I checked out my master branch, whenever I make a change in my project I will commit,

Made two changes 1, 2 and when I think my repository is stable i.e., at third change I will take a backup or create a historic point > created a tag v1.0

Still I have changes in my project, changed and committed 4 and after 5th commit, I need to take a backup so at that point I have created a tag > created a tag v1.1

Every commit tag has a number is said to be a checksum (40-digit number)

**Creating a tag for some past commit:**

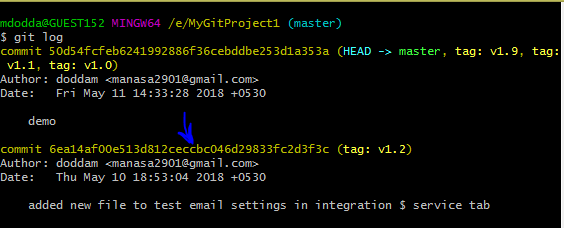
We can take backup for the past commits too.

**Step-1:** Use the following command to get the reference of commit (=checksum number)

Arrow indicates a checksum number. Checksum number is generated by a SHA-1 algorithm

Whenever a change is made in the project (i.e., committed then a unique checksum number is generated).

Command:  **git log** (to check all the commits)



**Step-2:** Take the checksum number of the commit or part of it is also enough.

**Step-3:** To create a past commit use the command,

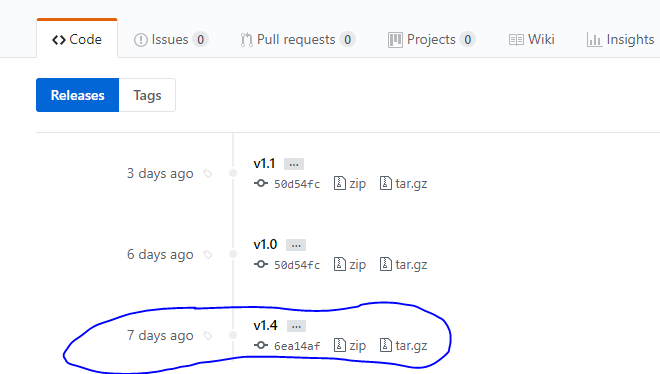
**git tag <tag name> <reference of the commit>**

For suppose, I need to take a backup for the commit made on May 10th, 2018. Use the following commands shown in the below screenshot.

1. Create tag using checksum number
2. View it whether created or not
3. Push the tag to the remote repository.



**Step-4:** Open remote repository > click releases > check it you can see v1.4 tag is created and at that point back up is taken.



By following above steps we can create a tag for some past commit and check the backup in zip file which can be downloaded.

**Issue -** I had a problem that is after exiting git log in windows git bash, text is disappeared on git bash and unable to access it properly > press ctrl+c two times(press **:q**)

**Note:**

git init > This command creates a hidden directory called .git

git uses .git folder to track changes



git looks our project as working directory



**Quick basic navigation and keyboard shortcuts for using GitBash:**

Move to a directory - cd e:/(forward slash)

To bring cursor to the beginning - Ctrl+A

To bring cursor to the ending - Ctrl+E

To delete content to beginning from the place of cursor - Ctrl+U

To delete content till ending from the place of cursor - Ctrl+K

To clear the screen – Ctrl+L/clear

To exit the git log interface you must type “**:q**”

Listing files within directory – ls (horizontal)

ls –l > long format

ls –l r\* > displays all the files starts with letter r

ls –a > shows hidden files

ls –la or ls -al> long format with hidden files

**Scenario:** *Pull changes from your Git repository on Git hub Cloud*

(Pulling a new file from git to local repository)

**Step-1:** Create a file in Git repo (in git hub).

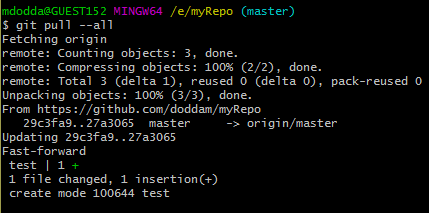
**Step-2:** Pull changes from a remote repository

We need to get that new file into your local repository.

To pull the file into your local repository, do the following:

Open your git bash window and navigate to your local repository.

1. cd /e
2. git clone <https://github.com/doddam/myRepo.git> > clone the repository to local
3. git pull --all  > enter this command to pull all the changes from git repo



The [**git pull**](https://www.atlassian.com/git/tutorials/syncing/git-pull) command merges the file from your remote repository (github) into your local repository with a single command.

**Step-3:** Navigate to your repository folder on your local system and you'll see the file you just added.

**Scenario:** *Use a Git branch to merge a file/Merging a branch into master branch using git bash*

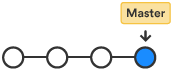
Branches are most powerful when you're working on a team. You can work on your own part of a project from your own branch, pull updates from github, and then merge all your work into the main branch when it's ready.

A branch represents an independent line of development for your repository. Think of it as a brand-new working directory, staging area, and project history. Before you create any new branches, you automatically start out with the main branch (called master).

**Step-1: Create a new branch and make a change**

Create a branch where you can add future plans that you aren't ready to commit. When you are ready to make those plans known to all, you can merge the changes into your GitHub repository and then delete the no-longer-needed branch.

It's important to understand that branches are just pointers to commits. When you create a branch, all Git needs to do is create a new pointer - it doesn’t create a whole new set of files or folders. Before you begin, your repository looks like this:



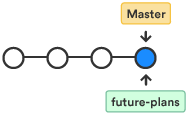
To create a branch, do the following:

1. Go to your git bash terminal window and navigate to the top level of your local repository and create a new branch using command

$ git branch <branch\_name>

$ **git branch future-plans**

This command creates a branch but does not switch you to that branch, so your repository looks something like this:



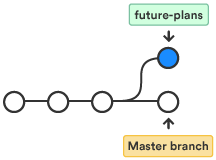
The repository history remains unchanged. All you get is a new pointer to the current branch. To begin working on the new branch, you have to check out the branch you want to use.

Checkout the new branch you just created to start using it.

$ **git checkout future-plans**

Switched to branch 'future-plans'

The git checkout command works hand-in-hand with git branch. Because you are creating a branch to work on something new, every time you create a new branch (with git branch), you want to make sure to check it out (with  git checkout ) if you're going to use it. Now that you’ve checked out the new branch, your Git workflow looks something like this:



1. Open git bash, navigate to the local repository in the system.
2. Add a file to the new branch or just make any modifications in the existing file.
3. Enter git status in the terminal window. You will see something like this:

$ **git status**  
On branch future-plans  
Changes not staged for commit:  
  (use "git add <file>..." to update what will be committed)  
  (use "git checkout -- <file>..." to discard changes in working directory)  
    modified: f1.txt  
no changes added to commit (use "git add" and/or "git commit -a")

Notice the On branch future-plans line? If you entered git status previously, the line was “On branch master” because you only had the one master branch. Before you stage or commit a change, always check this line to make sure the branch where you want to add the change is checked out.

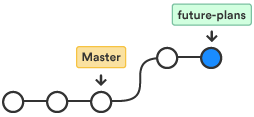
1. Stage your file.

$ **git add f1.txt**

1. Enter the git commit  command in the terminal window, as shown with the following:

$ git commit f1.txt -m “making a change in a branch”  
[future-plans e3b7732] making a change in a branch  
 1 file changed, 4 insertions(+)

With this recent commit, your repository looks something like this:

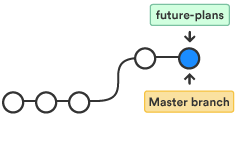


Now it's time to merge the change that you just made back into the master branch.

## **Step-2: Merge your branch > “fast-forward merging”**

You can merge your future-plans branch into the main branch on your local system.

Because you created only one branch and made one change, use the fast-forward branch method to merge.  You can do a fast-forward merge because you have a linear path from the current branch tip to the target branch. Instead of “actually” merging the branches, all Git has to do to integrate the histories is move (i.e., “fast-forward”) the current branch tip up to the target branch tip. This effectively combines the histories, since all of the commits reachable from the target branch are now available through the current one.



To complete a **fast-forward merge** do the following:

1. Go to your terminal window and navigate to the top level of your local repository.
2. Enter the git status command to be sure you have all your changes committed and find out what branch you have checked out.

$ **git status**   
On branch future-plans  
nothing to commit, working directory clean

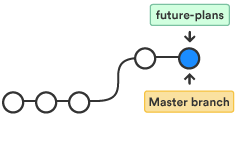
1. Switch to the master branch.

$ **git checkout master**   
Switched to branch 'master'  
Your branch is up-to-date with 'origin/master'.

1. Merge changes from the future-plans branch into the master branch. It will look something like this:

$ **git merge future-plans**  
Updating fcbeeb0..e3b7732  
Fast-forward  
 f1.txt | 4 ++++  
 1 file changed, 4 insertions(+)

You've essentially moved the pointer for the master branch forward to the current head and your repository looks something like this:



1. Because you don't plan on using future-plans anymore, you can delete the branch.

**$ git branch -d future-plans**Deleted branch future-plans (was e3b7732).

When you delete future-plans, you can still access the branch from master using a commit id. For example, if you want to undo the changes added from future-plans, use the commit id you just received to go back to that branch.

1. Enter git status to see the results of your merge, which show that your local repository is one ahead of your remote repository.   
   It will look something like this:

$ git status   
On branch master  
Your branch is ahead of 'origin/master' by 1 commit.  
  (use "git push" to publish your local commits)  
nothing to commit, working directory clean

Here's what you've done so far:

* Created a branch and checked it out
* Made a change in the new branch
* Committed the change to the new branch
* Integrated that change back into the main branch
* Deleted the branch you are no longer using.

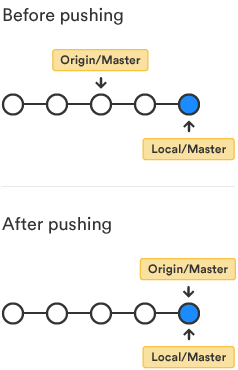
Next, we need to push all this work back up to Bitbucket, your remote repository. 

**Step-3: Push your change to GitHub**

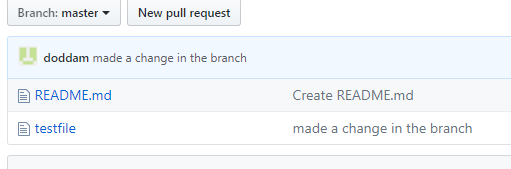
Here's how to push your change to the remote repository:

1. From the repository directory in your terminal window, enter following command to push the changes. It will result in something like this:

$ **git push origin master**



1. Now open your git hub account > click on master branch > changes made are seen as shown in the following screenshot,



Check the following commands from the git bash terminal for easy understanding:

mdodda@GUEST152 MINGW64 /e

$ git clone https://github.com/doddam/Test.git

Cloning into 'Test'...

remote: Counting objects: 3, done.

remote: Total 3 (delta 0), reused 0 (delta 0), pack-reused 0

Unpacking objects: 100% (3/3), done.

mdodda@GUEST152 MINGW64 /e

$ cd Test

mdodda@GUEST152 MINGW64 /e/Test (master)

$ ll

total 1

-rw-r--r-- 1 mdodda 1049089 8 May 22 15:22 README.md

mdodda@GUEST152 MINGW64 /e/Test (master)

$ git branch future-plans

mdodda@GUEST152 MINGW64 /e/Test (master)

$ git checkout future-plans

Switched to branch 'future-plans'

mdodda@GUEST152 MINGW64 /e/Test (future-plans)

$ vi f1.txt

mdodda@GUEST152 MINGW64 /e/Test (future-plans)

$ git status

On branch future-plans

Untracked files:

(use "git add <file>..." to include in what will be committed)

f1.txt

nothing added to commit but untracked files present (use "git add" to track)

mdodda@GUEST152 MINGW64 /e/Test (future-plans)

$ git add f1.txt

warning: LF will be replaced by CRLF in f1.txt.

The file will have its original line endings in your working directory.

mdodda@GUEST152 MINGW64 /e/Test (future-plans)

$ git commit f1.txt -m "made a change in the branch"

warning: LF will be replaced by CRLF in f1.txt.

The file will have its original line endings in your working directory.

[future-plans 10cda5b] made a change in the branch

1 file changed, 2 insertions(+)

create mode 100644 f1.txt

mdodda@GUEST152 MINGW64 /e/Test (future-plans)

$ git status

On branch future-plans

nothing to commit, working tree clean

mdodda@GUEST152 MINGW64 /e/Test (future-plans)

$ git checkout master

Switched to branch 'master'

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

mdodda@GUEST152 MINGW64 /e/Test (master)

$ git branch

future-plans

\* master

mdodda@GUEST152 MINGW64 /e/Test (master)

$ git merge future-plans

Updating 5238da6..10cda5b

Fast-forward

f1.txt | 2 ++

1 file changed, 2 insertions(+)

create mode 100644 f1.txt

mdodda@GUEST152 MINGW64 /e/Test (master)

$ git branch -d future-plans

Deleted branch future-plans (was 10cda5b).

mdodda@GUEST152 MINGW64 /e/Test (master)

$ git status

On branch master

Your branch is ahead of 'origin/master' by 1 commit.

(use "git push" to publish your local commits)

nothing to commit, working tree clean

mdodda@GUEST152 MINGW64 /e/Test (master)

$ git push origin master

Counting objects: 3, done.

Delta compression using up to 4 threads.

Compressing objects: 100% (2/2), done.

Writing objects: 100% (3/3), 299 bytes | 149.00 KiB/s, done.

Total 3 (delta 0), reused 0 (delta 0)

To https://github.com/doddam/Test.git

5238da6..10cda5b master -> master

**Fork:**

A fork is a copy of a repository that you manage.

Forks let you make changes to a project without affecting the original repository.

You can fetch updates from or submit changes to the original repository with pull requests.

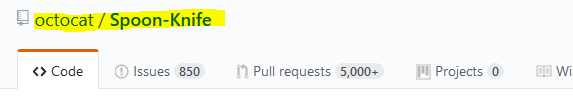
Forking a repository is similar to copying another repository, with two major differences:

1. You can use a pull request to suggest changes from your fork to the original repository, also known as the upstream repository.
2. You can bring changes from the upstream repository to your local fork by synchronizing your fork with the upstream repository.

**Fork an example repository:**

Forking a repository is a simple two-step process.

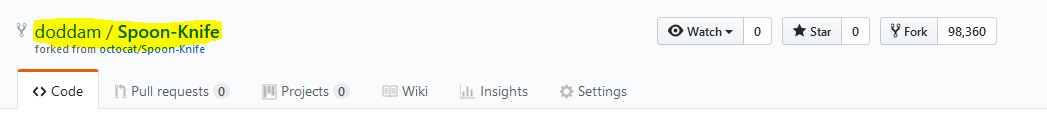
1. On GitHub, navigate to the link [**octocat/Spoon-Knife**](https://github.com/octocat/Spoon-Knife) repository.



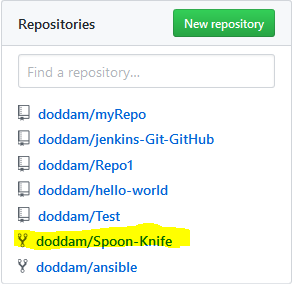
1. In the top-right corner of the page, click **Fork**.



Once you fork the Spoon-Knife repo, asks for credentials of GitHub – so give the credentials and the repo is cloned from octocat account to your account as shown below.



Now, you have a *fork* of the original octocat/Spoon-Knife repository, can check your repositories list now



**Keep your fork synced:**

Sync a fork of a repository to keep it up-to-date with the upstream repository.

Before you can sync your fork with an upstream repository, you must [configure a remote that points to the upstream repository](https://help.github.com/articles/configuring-a-remote-for-a-fork) in Git.

It's good practice to regularly sync your fork with the upstream repository. To do this, you'll need to use Git on the command line. You can practice setting the upstream repository using the same [octocat/Spoon-Knife](https://github.com/octocat/Spoon-Knife) repository you just forked!

**Step-1:** Set up Git

If you haven't yet, you should first [set up Git](https://help.github.com/articles/set-up-git). Don't forget to [set up authentication to GitHub from Git](https://help.github.com/articles/set-up-git#next-steps-authenticating-with-github-from-git) as well.

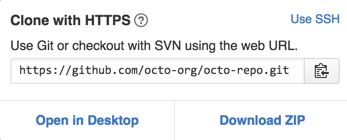
**Step-2:** Create a local clone of your fork (copying all the files from fork to the local system)

Right now, you have a fork of the Spoon-Knife repository, but you don't have the files in that repository on your computer. Let's create a clone of your fork locally on your computer.

1. On GitHub, navigate to **your fork** of the Spoon-Knife repository.
2. Under the repository name, click **Clone or download**.

Clone or download button

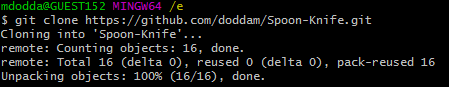
1. In the Clone with HTTPs section, click  to copy the clone URL for the repository.



1. Open Git Bash.
2. Type git clone, and then paste the URL you copied in Step2. It will look like this, with your GitHub username instead of YOUR-USERNAME:

$ git clone <https://github.com/YOUR-USERNAME/Spoon-Knife>

$ git clone <https://github.com/doddam/Spoon-Knife> - sample link to tell “doddam” is my github username.

1. Press **Enter**. Your local clone will be created.  
   

Now, you have a local copy of your fork of the Spoon-Knife repository!

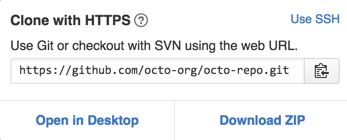
**Step-3:** Configure Git to sync your fork with the original Spoon-Knife repository

When you fork a project in order to propose changes to the original repository, you can configure Git to pull changes from the original, or *upstream*, repository into the local clone of your fork.

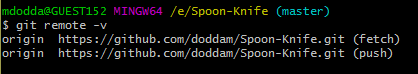
1. On GitHub, navigate to the [octocat/Spoon-Knife](https://github.com/octocat/Spoon-Knife) repository.
2. Under the repository name, click **Clone or download**.

Clone or download button

1. In the Clone with HTTPs section, click to copy the clone URL for the repository.



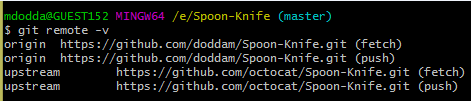
1. Open Git Bash.
2. Change directories to the location of the fork you cloned in [Step 2: Create a local clone of your fork](https://help.github.com/articles/fork-a-repo/#step-2-create-a-local-clone-of-your-fork).
3. To go to your home directory, type just cd with no other text.
4. To list the files and folders in your current directory, type ls.
5. To go into one of your listed directories, type cd your\_listed\_directory.
6. To go up one directory, type cd ...
7. Type git remote -v and press **Enter**. You'll see the current configured remote repository for your fork.



1. Type git remote add upstream, and then paste the URL you copied in Step 2 and press **Enter**. It will look like this:



1. To verify the new upstream repository you've specified for your fork, type git remote –v again. You should see the URL for your fork as origin, and the URL for the original repository as upstream.



Now, you can keep your fork synced with the upstream repository.

**What is a pull request exactly?**

Pull requests let you tell others about changes you've pushed to a GitHub repository. Once a pull request is sent, interested parties can review the set of changes, discuss potential modifications, and even push follow-up commits if necessary.

If you have [distributed version control](https://en.wikipedia.org/wiki/Distributed_revision_control) systems, every developer has a copy of the full repository. If you change something to the software, you commit your changes to your local repository. If different repositories should have these changes, you can push the changes (moving changes to another repository you have the right to write to) or pull the changes (copy revision from other repositories to your own). As many project have a main repository, a pull-request is the request that the maintainer pulls your changes.

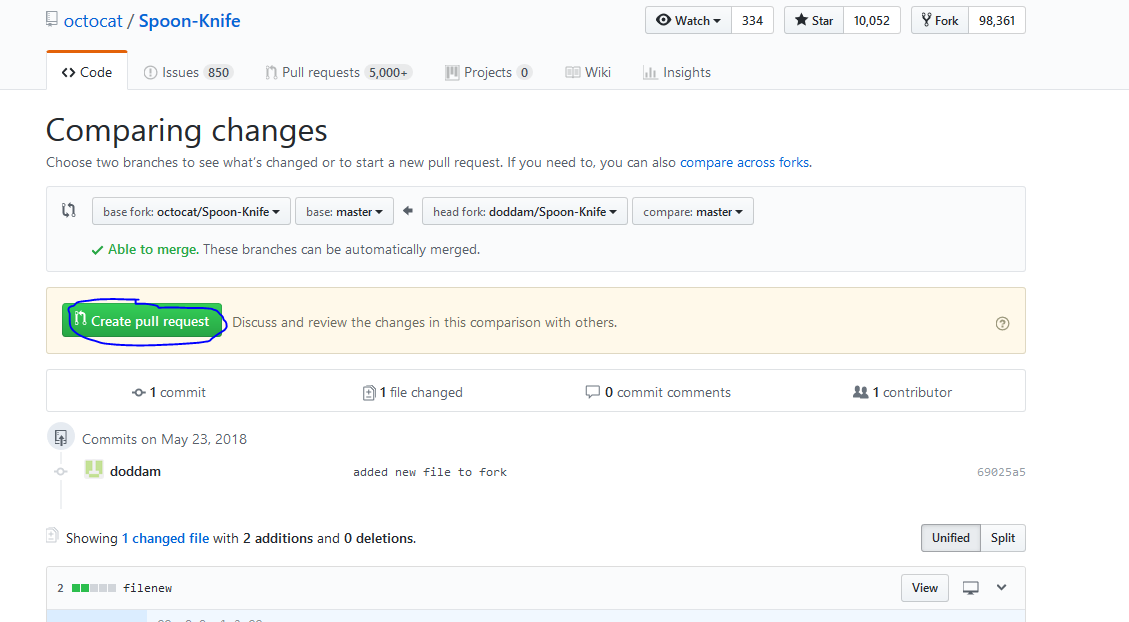
Simple example is when you clone/fork the open source repository, do some changes/commits and in order to merge the changes into the main repository, you are required to send the pull request which consist all your commits grouped into one logical piece so it can be reviewed by other developers.

**Making changes in forked repo:**

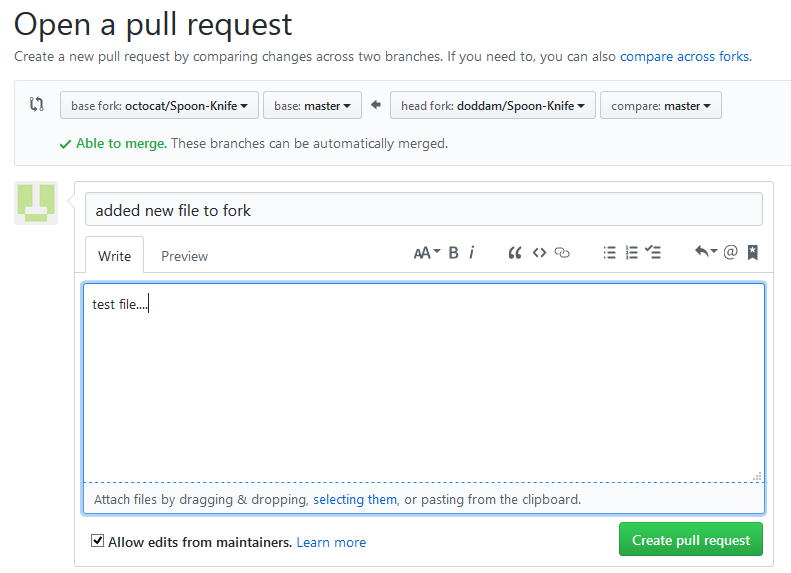
Added a new file in the Spoon-Knife repo that is forked from my account.



Click on new pull request > navigates to the original repo and compares for the changes.



Click on create pull request and a pull request is opened as shown in the below screenshot > add the comment and again click on pull request.



Thus the new file is added to the original repository



We can see here, my file is added below



Thus the changes made in the forked repo are pulled to the main repo…. :-)

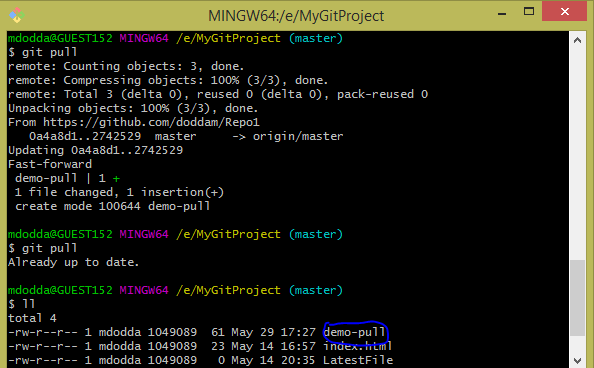
**git pull vs pull request:**

git pull is used to pull changes/files to the local repository from remote repository.

**Eg:** A file say “demo-pull” is added in remote > committed

Now check in your local you can’t find the file so type the command

“**git pull**” in order to get the new files from remote repository



Now you can check that file is in your local repository,



So finally,

If you use **git pull**, you pull the changes from the remote repository into yours.

If you send a **pull request** to another repository, you ask their maintainers to pull your changes into theirs (you more or less ask them to use a git pull from your repository).

Have seen that this is related to a **Fork and Pull** collaborative development model and is used for **code reviews**

**git pull:**

git pull fetches the latest changes of the current branch from a remote and applies those changes to your local copy of the branch. Generally this is done by merging, i.e., the local changes are merged into the remote changes.

**git pull** is shorthand for **git fetch** followed by **git merge FETCH\_HEAD.**

working directory

|=>.git

| |=>objects <= contains data for each commit

| |=>refs

| |=>heads

| |-master <= file containing current commit of local master branch

| |=>remotes

| |=>origin

| |-master <= file containing current commit of remote origin's master branch

|-FETCH\_HEAD <= file updated by `git fetch`, contains info of what was fetched

https://i.stack.imgur.com/zUInQ.png

**fig:** git fetch, git merge and git pull

**git fetch** just "downloads" the changes from the remote to your local repository. git pull downloads the changes and merges them into your current branch.

**git pull** vs **git fetch**:

**git pull** - pulls from a remote branch and merges it.

**git fetch** - only fetches from the remote branch but it does not merge.

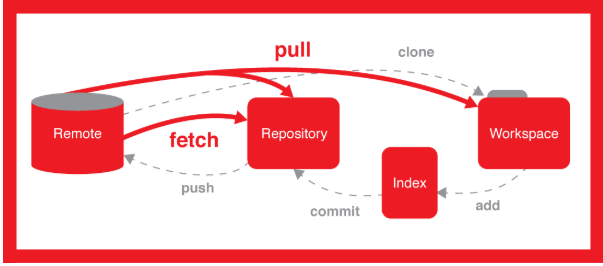
i.e., git pull = (git fetch + git merge)

If you run **git pull**, you do not need to merge the data to local.

If you run **git fetch**, it means you must run **git merge** for getting the latest code to your local machine. Otherwise, the local machine code would not be changed without merge.

The command git fetch makes your local copy up to date by getting data from remote repository. The reason we need this is because somebody else might have made some changes to the code and you want to keep yourself updated.

Check the below image to know how **git fetch** and **git pull** working together:



**Stashing:**

Often, when you’ve been working on part of your project, things are in a messy state and you want to switch branches for a bit to work on something else. The problem is, you don’t want to do a commit of half-done work just so you can get back to this point later. The answer to this issue is the git stash command.

Stashing takes the dirty state of your working directory — that is, your modified tracked files and staged changes — and saves it on a stack of unfinished changes that you can reapply at any time.

Think of the Stash as a clipboard on steroids: it takes all the changes in your working copy and saves them for you on a new clipboard. You're left with a clean working copy, i.e. you have no more local changes.

Later, at any time, you can restore the changes from that clipboard in your working copy - and continue working where you left off.

You can create as many Stashes as you want - you're not limited to storing only one set of changes. Also, a Stash is not bound to the branch where you created it: when you restore it, the changes will be applied to your current HEAD branch, whichever this may be.

**When to Stash:**

Stashing helps you get a clean working copy. While this can be helpful in many situations, it's strongly recommended...

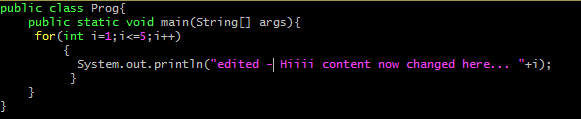
* ...before checking out a different branch.
* ...before pulling remote changes.
* ...before merging or rebasing a branch.

In short, use **git stash** to save changes temporarily in local memory.

**Stashing your work:**

To demonstrate, you’ll go into your project and start working on a couple of files and possibly stage one of the changes. If you run git status, you can see your dirty state:

I modified the file Prog.java



mdodda@GUEST152 MINGW64 /e/MyGitProject (master)

$ vi Prog.java

mdodda@GUEST152 MINGW64 /e/MyGitProject (master)

$ git status

On branch master

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

Changes not staged for commit:

(use "git add <file>..." to update what will be committed)

(use "git checkout -- <file>..." to discard changes in working directory)

modified: Prog.java

no changes added to commit (use "git add" and/or "git commit -a")

Used **git stash save “some message”** command to save my changes temporarily in local memory i.e., on stack

mdodda@GUEST152 MINGW64 /e/MyGitProject (master)

$ git stash save "content changed"

Saved working directory and index state On master: content changed

After stashing your changes, check the git status.

mdodda@GUEST152 MINGW64 /e/MyGitProject (master)

$ git status

On branch master

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

nothing to commit, working tree clean

You can easily get an overview of your current Stashes:

mdodda@GUEST152 MINGW64 /e/MyGitProject (master)

$ git stash list

stash@{0}: On master: content changed

The newest Stash will always be at the top of the list, named "stash@{0}". Older Stashes have higher numbers.

When you're ready to restore a saved Stash, you have two options:

(a) Calling "git stash pop" will apply the newest Stash *and* clear it from your Stash clipboard.

(b) Calling "git stash apply <stashname>" will also apply the specified Stash, but it will *remain saved*. You can delete it later via "git stash drop <stashname>".

You can choose to *not* specify the Stash when using any of these commands. Then, Git will simply take the newest Stash (always "stash@{0}").

mdodda@GUEST152 MINGW64 /e/MyGitProject (master)

$ git stash pop

On branch master

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

Changes not staged for commit:

(use "git add <file>..." to update what will be committed)

(use "git checkout -- <file>..." to discard changes in working directory)

modified: Prog.java

no changes added to commit (use "git add" and/or "git commit -a")

Dropped refs/stash@{0} (a8d1afae2ff12ec9d4cebe2bd5fa2babc9e8ece8)

mdodda@GUEST152 MINGW64 /e/MyGitProject (master)

$ git status

On branch master

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

Changes not staged for commit:

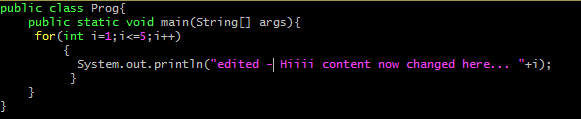
(use "git add <file>..." to update what will be committed)

(use "git checkout -- <file>..." to discard changes in working directory)

modified: Prog.java

no changes added to commit (use "git add" and/or "git commit -a")

Now you can check the file below as our temporary changes are left same before and after stashing.



*In detail:*

**Stashing your work:**

The git stash command takes your uncommitted changes (both staged and unstaged), saves them away for later use, and then reverts them from your working copy. For example:

**$ git status**

On branch master

Changes to be committed:

new file: style.css

Changes not staged for commit:

modified: index.html

**$ git stash**

Saved working directory and index state WIP on master: 5002d47 our new homepage

HEAD is now at 5002d47 our new homepage

**$ git status**

On branch master

nothing to commit, working tree clean

At this point you're free to make changes, create new commits, switch branches, and perform any other Git operations; then come back and re-apply your stash when you're ready.

Note that the stash is local to your Git repository; stashes are not transferred to the server when you push.

**Re-applying your stashed changes:**

You can reapply previously stashed changes with git stash pop:

**$ git status**

On branch master

nothing to commit, working tree clean

**$ git stash pop**

On branch master

Changes to be committed:

new file: style.css

Changes not staged for commit:

modified: index.html

Dropped refs/stash@{0} (32b3aa1d185dfe6d57b3c3cc3b32cbf3e380cc6a)

Popping your stash removes the changes from your stash and reapplies them to your working copy.

Alternatively, you can reapply the changes to your working copy and keep them in your stash with git stash apply:

**$ git stash apply**

On branch master

Changes to be committed:

new file: style.css

Changes not staged for commit:

modified: index.html

This is useful if you want to apply the same stashed changes to multiple branches.

Now that you know the basics of stashing, there is one caveat with git stash you need to be aware of: by default Git won't stash changes made to untracked or ignored files.

**Stashing untracked or ignored files:**

By default, running git stash will stash:

* Changes that have been added to your index (staged changes)
* Changes made to files that are currently tracked by Git (unstaged changes)

But it will not stash:

* New files in your working copy that have not yet been staged
* Files that have been ignored

So if we add a third file to our example above, but don't stage it (i.e. we don't run git add), git stash won't stash it.

**$ script.js**

**$ git status**

On branch master

Changes to be committed:

new file: style.css

Changes not staged for commit:

modified: index.html

Untracked files:

script.js

**$ git stash**

Saved working directory and index state WIP on master: 5002d47 our new homepage

HEAD is now at 5002d47 our new homepage

**$ git status**

On branch master

Untracked files:

script.js

Adding the -u option (or --include-untracked) tells git stash to also stash your untracked files:

**$ git status**

On branch master

Changes to be committed:

new file: style.css

Changes not staged for commit:

modified: index.html

Untracked files:

script.js

**$ git stash -u**

Saved working directory and index state WIP on master: 5002d47 our new homepage

HEAD is now at 5002d47 our new homepage

**$ git status**

On branch master

nothing to commit, working tree clean

You can include changes to ignored files as well by passing the -a option (or --all) when running git stash.



**Managing multiple stashes:**

You aren't limited to a single stash. You can run git stash several times to create multiple stashes, and then use git stash list to view them. By default, stashes are identified simply as a "WIP" – work in progress – on top of the branch and commit that you created the stash from. After a while it can be difficult to remember what each stash contains:

**$ git stash list**

stash@{0}: WIP on master: 5002d47 our new homepage

stash@{1}: WIP on master: 5002d47 our new homepage

stash@{2}: WIP on master: 5002d47 our new homepage

To provide a bit more context, it's good practice to annotate your stashes with a description, using git stash save "message":

**$ git stash save "add style to our site"**

Saved working directory and index state On master: add style to our site

HEAD is now at 5002d47 our new homepage

**$ git stash list**

stash@{0}: On master: add style to our site

stash@{1}: WIP on master: 5002d47 our new homepage

stash@{2}: WIP on master: 5002d47 our new homepage

By default, git stash pop will re-apply the most recently created stash: stash@{0}

You can choose which stash to re-apply by passing its identifier as the last argument, for example:

**$ git stash pop stash@{2}**

Stashing untracked files using different options:

$ **git stash save -u** (or)

$ **git stash save --include-untracked**

The following will give you the diff of the topmost stash item:

**$ git stash show**

*In short:*

**git stash pop**- To apply the stash and then immediately drop it from your stack.

**git stash drop** - Top of the stash from the stack will be deleted.

**git stash drop <stash\_id>** - If you no longer need a particular stash, you can delete it with the command.

**git stash clear** - You can delete all of your stashes from the stack

**git stash pop** = = (**git stash apply** && **git stash drop**)

**Scenario** - *Stashing changes from one branch to other with ease or before checking out a different branch*

* Created a new file and working with that file “welcome” on master branch using git console.

mdodda@GUEST152 MINGW64 /e/MyGitProject (master)

$ cat > welcome

welcome to the stash concept......!!!

mdodda@GUEST152 MINGW64 /e/MyGitProject (master)

$ git add welcome

warning: LF will be replaced by CRLF in welcome.

The file will have its original line endings in your working directory.

* Now I want to move to a different branch NewB to fix a minor bug I've just found, so I need to stash my changes.
* Before checking out the master branch, I have used the command **git stash** to save file temporarily.

mdodda@GUEST152 MINGW64 /e/MyGitProject (master)

$ git stash save "stash demo"

Saved working directory and index state On master: stash demo

* You can easily get an overview of your current Stashes by using the following command:

mdodda@GUEST152 MINGW64 /e/MyGitProject (master)

$ git stash list

stash@{0}: On master: stash demo

stash@{1}: On master: added a file - newfile

stash@{2}: WIP on master: a28e114 deltd files

stash@{3}: On master: added b.txt file

* You can save a stash on one branch, switch to another branch later, and try to reapply the changes using the **git stash apply** command. (reapply changes from one branch to other branch)
* If you want to apply one of the older stashes, you can specify it by naming it, like this:

**git stash apply stash@{2}.** If you don’t specify a stash, Git assumes the most recent stash and tries to apply it:

mdodda@GUEST152 MINGW64 /e/MyGitProject (master)

$ git checkout NewB

Switched to branch 'NewB'

Your branch is ahead of 'origin/NewB' by 2 commits.

(use "git push" to publish your local commits)

mdodda@GUEST152 MINGW64 /e/MyGitProject (NewB)

$ cat welcome

cat: welcome: No such file or directory

Here I used git stash apply command to apply the stashes saved in master branch to the other branch NewB

mdodda@GUEST152 MINGW64 /e/MyGitProject (NewB)

$ git stash apply stash@{0}

On branch NewB

Your branch is ahead of 'origin/NewB' by 2 commits.

(use "git push" to publish your local commits)

Changes to be committed:

(use "git reset HEAD <file>..." to unstage)

new file: welcome

mdodda@GUEST152 MINGW64 /e/MyGitProject (NewB)

$ cat welcome

welcome to the stash concept......!!!

mdodda@GUEST152 MINGW64 /e/MyGitProject (NewB)

$ git commit -m "welcome file added"

[NewB 93a61b7] welcome file added

1 file changed, 1 insertion(+)

create mode 100644 welcome

mdodda@GUEST152 MINGW64 /e/MyGitProject (NewB)

$ git checkout master

Switched to branch 'master'

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

mdodda@GUEST152 MINGW64 /e/MyGitProject (master)

$ cat welcome

cat: welcome: No such file or directory

* Now check the file “welcome” is on the branch “NewB” where I have stashed the changes in one branch and applied it on other branch i.e., from master branch to NewB branch.

**Issue:**

"**error**: you need to resolve your current index first Prog.java: needs merge"

**Solution**: git reset - -merge

**What happens if you create a new branch?**

Well, doing so creates a new pointer for you to move around. Let’s say you create a new branch called *testing.* You do this with the git branch command:

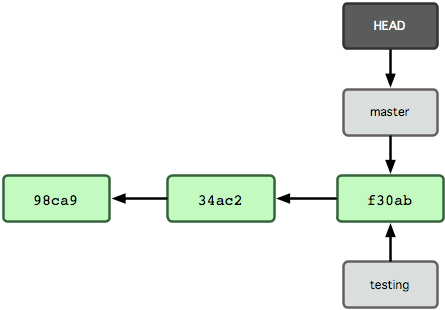
**$ git branch testing**

This creates a new pointer at the same commit you’re currently on

**How does Git know what branch you’re currently on?**

**Git: What's HEAD**

It keeps a special pointer called HEAD. Note that this is a lot different than the concept of HEAD in other VCSs you may be used to, such as Subversion or CVS. In Git, this is a pointer to the local branch you’re currently on. In this case, you’re still on master. The git branch command only created a new branch - it didn’t switch to that branch.



HEAD file pointing to the branch you’re on.

**How to find what commit is current HEAD?**

Look in the file .git/HEAD.

You can do so by the command

# show the commit id the current HEAD points to

cd my\_project\_dir > moving to the current project directory

cat .git/HEAD

Sample output:

ref: refs/heads/master

The refs/heads/master is again a reference. It is a file, you can look at .git/refs/heads/master

The content on my disk is: 4929082f12ef06321143316fca5d04ce439b96bc

**What is detached HEAD?**

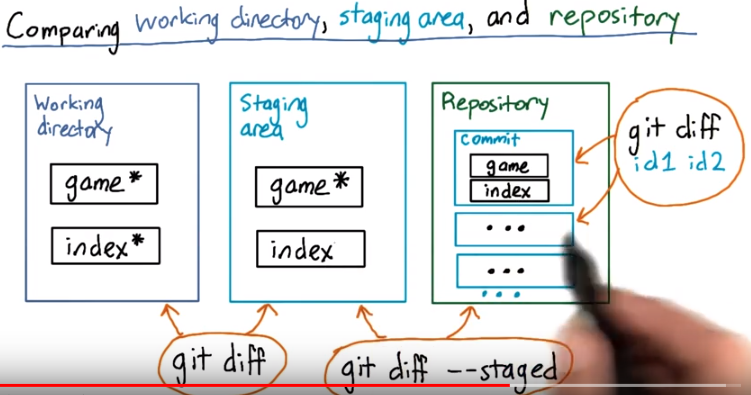
When HEAD is pointing to a commit ID directly, instead of a named branch, it's called detached HEAD.

# Git: Diff between {Working Dir, Staged Area, Last Commit}

# Shows changes between commits, commit and working tree, etc.

There are 3 major concepts of places:

* Working Directory → files in your working directory.
* Staging Area (aka cache, index) → a temp area that git add is placed into.
* HEAD → A reference to a specific commit (think of it as a variable). Normally, it points to the last commit in local repository. (that is, after you did git commit).



All are local on your disk.

One important concept is Commit ID. Every commit has a ID. The commit id is a 40 digits hexadecimal, for example: 3b6ea398cc2d69212b04c29f06b8d15c0af34e34.

**Check the following commands that shows the differences between working directory, staging area and last commit:**

**How to get commit ID?**

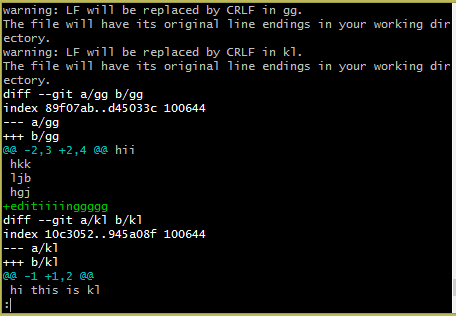
# To show the last 3 commit's commit id

git log -3

**How to diff between {working dir, staging area}?**

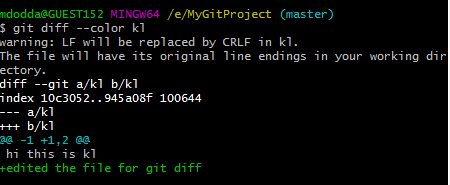
# Diff working dir, staging area

git diff –color



# Diff working dir, staging area, 1 file

git diff --color filename



**How to diff between {staging area, last commit}?**

# Diff staging area, last commit. (--staged is same as --cached)

git diff --color --staged ‹commitID›

Eg: git diff --color --staged 7a020cb76d7ac5694e602acb9448ba2ba78d26db

**How to diff between {last commit, working dir}?**

First, use git log to get a commit ID.

# Diff last commit, working dir

git diff --color ‹commitID›

git diff --color 7a020cb76d7ac5694e602acb9448ba2ba78d26db

# To show the word difference for all the files.

$ git diff --word-diff

#To show the word difference only for the given filename.

$ git diff --word-diff filename



**Git Diff between 2 Commits:**

**How to diff between 2 commits in the same branch?**

#1 First run following command

git log file\_name

#2 To find the commits IDs. Then, do

git diff commit\_ID\_1 commit\_ID\_2 file\_name

Eg:

git diff 3819bfa40adf7b2cd7063a98f4f8b9bd1fb4f639 4929082f12ef06321143316fca5d04ce439b96bc test

(or)

git diff 3819bfa40adf7b2cd7063a98f4f8b9bd1fb4f639 4929082f12ef06321143316fca5d04ce439b96bc

You can also use the first few characters of commit id instead of whole id.

**git difftool:**

*git difftool* is a Git command that allows you to compare and edit files between revisions using common diff tools. *git difftool* is a frontend to *git diff*.

Diffing is a function that takes two input data sets and outputs the changes between them. git diff is a multi-use Git command that when executed runs a diff function on Git data sources. These data sources can be commits, branches, files and more

Configured vim diff as default git difftool by using the following command

$ **git config --global diff.tool vimdif**

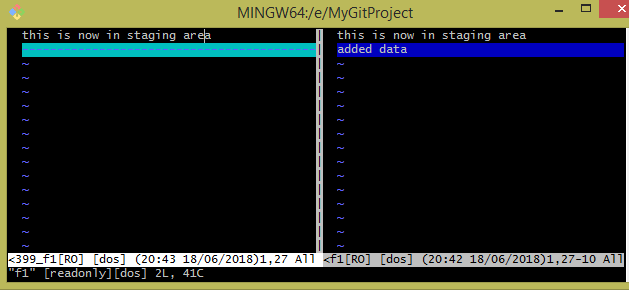
The following command turns off the prompt globally (for all repos):

$ **git config --global difftool.prompt false**

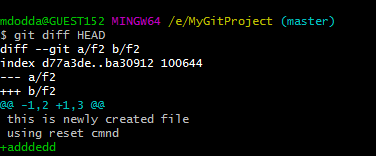
You can use git config --list, or look at your ~/.gitconfig file. Local config will be in your repository's .git/config file. Can view the file by using the following command:

$ **cat ~/.gitconfig**

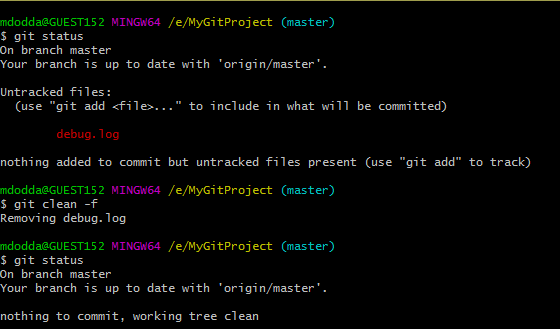
$ **git difftool f2** (before and after staging)



$ **git diff HEAD** (compares between the working tree and the HEAD also known as most recent commit)



$ **git clean -f**

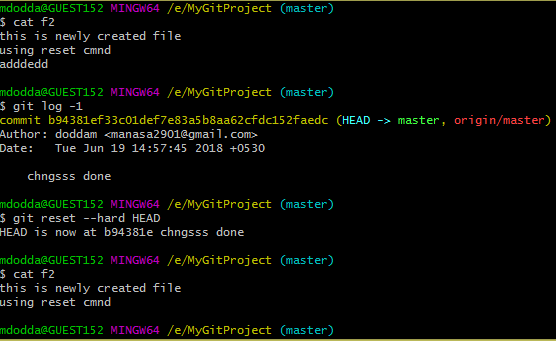
git clean -f will remove untracked files, meaning they're gone for good since they aren't stored in the repository. Make sure you really want to remove all untracked files before doing this.

git reset --hard will not remove untracked files, whereas git-clean will remove any files from the tracked root directory that are not under Git tracking.

git clean -xdf even removes all ignored files

When I want to revert to a previous commit I use the following command:

$ **git reset - -hard** HEAD

c

### **Revert:**

### Reverting undoes (erases the last change done to the document reverting it to an older state)

a commit by creating a *new* commit. This is a safe way to undo changes, as it has no chance of re-writing the commit history. For example, the following command will figure out the changes contained in the second-to-last commit, create a new commit undoing those changes, and tack the new commit onto the existing project.

Example:

**mdodda@GUEST152 MINGW64 /e/MyGitProject (master)**

$ cat >> tt2

this is tt2 file

**mdodda@GUEST152 MINGW64 /e/MyGitProject (master)**

$ cat tt2

this is tt2 file

**mdodda@GUEST152 MINGW64 /e/MyGitProject (master)**

$ git add .

warning: LF will be replaced by CRLF in tt2.

The file will have its original line endings in your working directory.

**mdodda@GUEST152 MINGW64 /e/MyGitProject (master)**

$ git commit -m "added tt2 file"

[master 20d19ad] added tt2 file

1 file changed, 1 insertion(+)

create mode 100644 tt2

**mdodda@GUEST152 MINGW64 /e/MyGitProject (master)**

$ git log -1

commit 20d19ad961bc715792d296102e75e6173bad7a9e (HEAD -> master)

Author: doddam <manasa2901@gmail.com>

Date: Tue Jun 19 22:38:32 2018 +0530

added tt2 file

**mdodda@GUEST152 MINGW64 /e/MyGitProject (master)**

$ git show 20d19

commit 20d19ad961bc715792d296102e75e6173bad7a9e (HEAD -> master)

Author: doddam <manasa2901@gmail.com>

Date: Tue Jun 19 22:38:32 2018 +0530

added tt2 file

diff --git a/tt2 b/tt2

new file mode 100644

index 0000000..53ed10e

--- /dev/null

+++ b/tt2

@@ -0,0 +1 @@

+this is tt2 file

**mdodda@GUEST152 MINGW64 /e/MyGitProject (master)**

$ cat >> tt2

working with git show cmnd

**mdodda@GUEST152 MINGW64 /e/MyGitProject (master)**

$ git add .

warning: LF will be replaced by CRLF in tt2.

The file will have its original line endings in your working directory.

**mdodda@GUEST152 MINGW64 /e/MyGitProject (master)**

$ git commit -m "added tt2 file scnf tym"

[master 1b4d305] added tt2 file scnf tym

1 file changed, 1 insertion(+)

**mdodda@GUEST152 MINGW64 /e/MyGitProject (master)**

$ git push -u origin master

Counting objects: 9, done.

Delta compression using up to 4 threads.

Compressing objects: 100% (6/6), done.

Writing objects: 100% (9/9), 732 bytes | 48.00 KiB/s, done.

Total 9 (delta 3), reused 0 (delta 0)

remote: Resolving deltas: 100% (3/3), completed with 1 local object.

To https://github.com/doddam/Repo1.git

b94381e..1b4d305 master -> master

Branch 'master' set up to track remote branch 'master' from 'origin'.

**mdodda@GUEST152 MINGW64 /e/MyGitProject (master)**

$ git log -1

commit 1b4d3051017e001052c244df529fc8bbd2c85c8e (HEAD -> master, origin/master)

Author: doddam <manasa2901@gmail.com>

Date: Tue Jun 19 22:42:42 2018 +0530

added tt2 file scnf tym

**mdodda@GUEST152 MINGW64 /e/MyGitProject (master)**

$ git revert 1b4d3

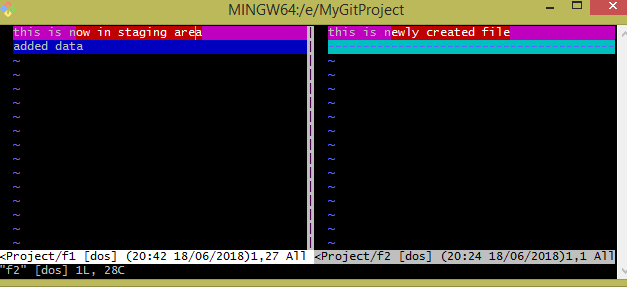
[master f32758c] Revert "added tt2 file scnd tym"

1 file changed, 1 deletion(-)

**mdodda@GUEST152 MINGW64 /e/MyGitProject (master)**

$ cat tt2

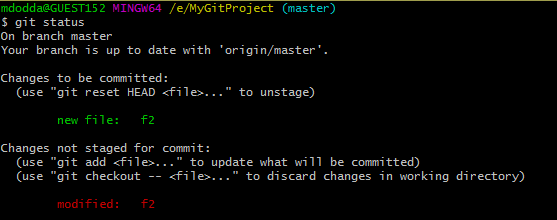
this is tt2 file

**vimdiff file1 file2**

Use the following to exit from git difftool

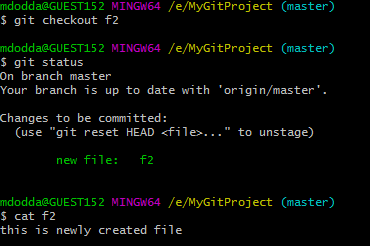
**:qa**

Modified a file f2:



Now file can be bought to the old state usingi.e., to revert the changes made to a specific file.

**“git checkout filename”**



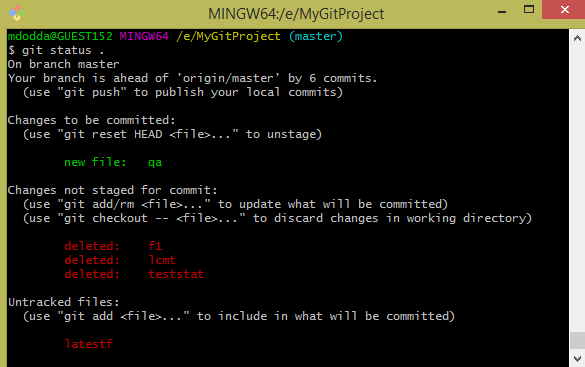
The extra content is deleted which is added before.

**git status:** Summary of {last commit, staging area, working dir}

**How to find what files are changed?**

# show changes between {staging area, last commit} and {staging area, working dir}

**git status .**



The **first section** “# Changes to be committed:” is the diff between {staging area, last commit}.

The **second section** “# Changes not staged for commit:” is the diff between {working dir, staging area}.

The **third section** “# Untracked files:” is also the diff between {working dir, staging area}.

**Creating a branch from another branch:**

If you want create a new branch from any of the existing branches in Git, just follow the options.

First change/checkout into the branch from where you want to create a new branch. For example, if you have the following branches like:

• master

• dev

• branch1

So if you want to create a new branch called "subbranch\_of\_b1" under the branch named "branch1" follow the steps:

1. Checkout or change into "branch1"

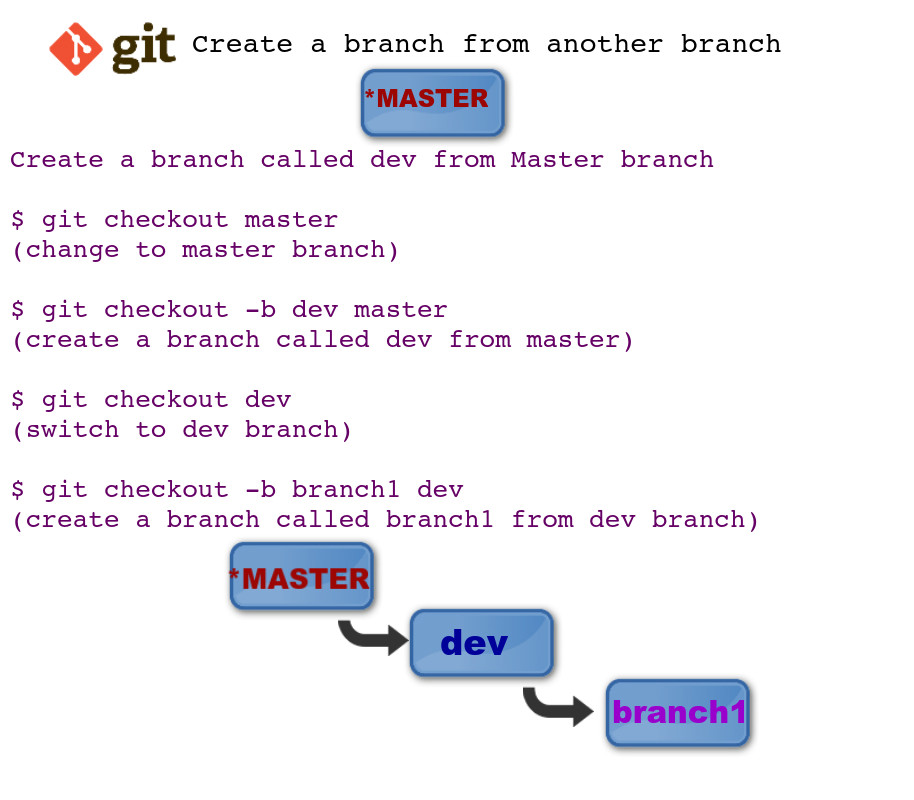
2. git checkout branch1

3. Now create your new branch called "subbranch\_of\_b1" under the "branch1" using the following command.

4. git checkout -b subbranch\_of\_b1 branch1

The above will create a new branch called subbranch\_of\_b1 under the branch branch1 (note that branch1 in the above command isn't mandatory since the HEAD is currently pointing to it, you can precise it if you are on a different branch though).

5. Now after working with the subbranch\_of\_b1 you can commit and push or merge it locally or remotely.



**git branch commands’ various options:**

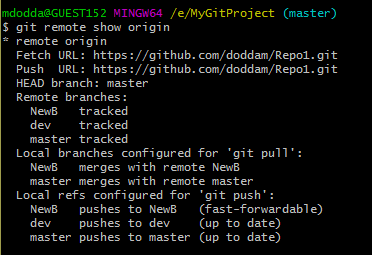
-a shows all local and remote branches

-r shows only remote branches.

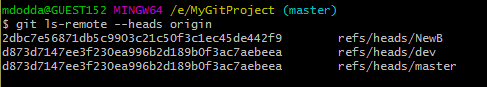


Figured out what branches are on remote by actually using the remote related commands,

**git remote** and **git ls-remote**



The ls-remote command returns the SHA1 hash of the latest commit for that reference, so it is quite easy to parse out and get to the exact commit you need if you’re doing some scripting. The --heads option lists only branch names since the command can list tags too.



**.gitignore:**

## **Ignoring files:**

## From time to time, there are files you don't want Git to check in to GitHub. There are a few ways to tell Git which files to ignore.

**Create a local .gitignore:**

If you create a file in your repository named .gitignore, Git uses it to determine which files and directories to ignore, before you make a commit.

A .gitignore file should be committed into your repository, in order to share the ignore rules with any other users that clone the repository.

GitHub maintains an official list of recommended .gitignore files for many popular operating systems, environments, and languages in the github/gitignore public repository.

1. In Terminal, navigate to the location of your Git repository.
2. Enter **touch .gitignore** to create a .gitignore file.

**Ignoring a previously committed file:**

If you want to ignore a file that you've committed in the past, you'll need to delete the file from your repository and then add a .gitignore rule for it. Using the --cached option with git rm means that the file will be deleted from your repository, but will remain in your working directory as an ignored file.

$ **git rm - -cached FILENAME**

mdodda@GUEST152 MINGW64 /e/MyGitProject (master)

$ git rm --cached file5

rm 'file5'

mdodda@GUEST152 MINGW64 /e/MyGitProject (master)

$ git commit -m "started ignoring file5"

[master 3db131d] started ignoring file5

1 file changed, 2 deletions(-)

delete mode 100644 file5

You can omit the --cached option if you want to delete the file from both the repository and your local file system.

**Committing an ignored file:**

It is possible to force an ignored file to be committed to the repository using the -f (or --force) option with git add:

$ **cat .gitignore**

del.txt

\*.log

#ignoring all html files

\*.html

\*5

$ **git add -f bug.log**

$ **git commit -m "Force adding bug.log"**

You might consider doing this if you have a general pattern (like \*.log) defined, but you want to commit a specific file. However a better solution is to define an exception to the general rule:

$ **cat .gitignore**

del.txt

\*.log

!bug.log

#ignoring all html files

\*.html

\*5

$ **git add bug.log**

$ **git commit -m "Adding bug.log"**

**Git Clean:**

Git clean is to some extent an 'undo' command.

Git clean can be considered complementary to other commands like git reset and git checkout.

Git clean command operates on untracked files. Untracked files are files that have been created within your repo's working directory but have not yet been added to the repository's tracking index using the git add command.

**Common options and usage:**

-n

The -n option will perform a “**dry run**” of git clean. This will show you which files are going to be removed without actually removing them. It is a best practice to always first perform a dry run of git clean. We can demonstrate this option in the demo repo we created earlier.

$ **git clean -n**

Would remove untracked\_file

The output tells us that untracked\_file will be removed when the git clean command is executed. Notice that the untracked\_dir is not reported in the output here. By default git clean will not operate recursively on directories. This is another safety mechanism to prevent accidental permanent deletion.

The force option initiates the actual deletion of untracked files from the current directory. Force is required unless the clean.requireForce configuration option is set to false. This will not remove untracked folders or files specified by .gitignore. Let us now execute a live git clean in our example repo.

$ **git clean -f**

Removing untracked\_file

The command will output the files that are removed. You can see here that untracked\_file has been removed. Executing git status at this point or doing a ls will show that untracked\_file has been deleted and is nowhere to be found. By default git clean -f will operate on all the current directory untracked files. Additionally, a <path> value can be passed with the -f option that will remove a specific file.

$ **git clean –f <path>**

Created a new file “file1” and check below for the executed the commands.

mdodda@GUEST152 MINGW64 /e/MyGitProject (master)

$ touch file1

mdodda@GUEST152 MINGW64 /e/MyGitProject (master)

$ git status

On branch master

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

Untracked files:

(use "git add <file>..." to include in what will be committed)

file1

nothing added to commit but untracked files present (use "git add" to track)

mdodda@GUEST152 MINGW64 /e/MyGitProject (master)

$ ll

total 5

-rw-r--r-- 1 mdodda 1049089 76 Jun 25 20:45 debug.log

-rw-r--r-- 1 mdodda 1049089 75 Jun 26 21:59 del.txt

-rw-r--r-- 1 mdodda 1049089 61 Jun 13 20:13 demo-pull

-rw-r--r-- 1 mdodda 1049089 0 Jun 27 19:35 file1

-rw-r--r-- 1 mdodda 1049089 39 Jun 13 20:13 index.html

-rw-r--r-- 1 mdodda 1049089 199 Jun 13 20:13 Prog.java

mdodda@GUEST152 MINGW64 /e/MyGitProject (master)

$ git clean -n

Would remove file1

mdodda@GUEST152 MINGW64 /e/MyGitProject (master)

$ ll

total 5

-rw-r--r-- 1 mdodda 1049089 76 Jun 25 20:45 debug.log

-rw-r--r-- 1 mdodda 1049089 75 Jun 26 21:59 del.txt

-rw-r--r-- 1 mdodda 1049089 61 Jun 13 20:13 demo-pull

-rw-r--r-- 1 mdodda 1049089 0 Jun 27 19:35 file1

-rw-r--r-- 1 mdodda 1049089 39 Jun 13 20:13 index.html

-rw-r--r-- 1 mdodda 1049089 199 Jun 13 20:13 Prog.java

mdodda@GUEST152 MINGW64 /e/MyGitProject (master)

$ git clean -f

Removing file1

mdodda@GUEST152 MINGW64 /e/MyGitProject (master)

$ ll

total 5

-rw-r--r-- 1 mdodda 1049089 76 Jun 25 20:45 debug.log

-rw-r--r-- 1 mdodda 1049089 75 Jun 26 21:59 del.txt

-rw-r--r-- 1 mdodda 1049089 61 Jun 13 20:13 demo-pull

-rw-r--r-- 1 mdodda 1049089 39 Jun 13 20:13 index.html

-rw-r--r-- 1 mdodda 1049089 199 Jun 13 20:13 Prog.java

mdodda@GUEST152 MINGW64 /e/MyGitProject (master)

$ git status

On branch master

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

nothing to commit, working tree clean

In the same way, applies to the directories:

-d include directories

The -d option tells git clean that you also want to remove any untracked directories, by default it will ignore directories. We can add the -d option to our previous examples:

$ **git clean -dn**

Would remove untracked\_dir/

$ **git clean -df**

Removing untracked\_dir/

Here we have executed a 'dry run' using the -dn combination which outputs untracked\_dir is up for removal. Then we execute a forced clean, and receive output that untracked\_dir is removed.

-x force removal of ignored files

The -x option tells git clean to also include any ignored files. As with previous git clean invocations, it is a best practice to execute a 'dry run' first, before the final deletion. The -x option will act on all ignored files, not just project build specific ones.

$ **git clean –xf**

Like the -d option -x can be passed and composed with other options. This example demonstrates a combination with -f that will remove untracked files from the current directory as well as any files that Git usually ignores.

mdodda@GUEST152 MINGW64 /e/MyGitProject (master)

$ cat .gitignore

#ignoring all html files

\*.html

!index.html

f\*

!f2

mdodda@GUEST152 MINGW64 /e/MyGitProject (master)

$ git status

On branch master

Your branch is ahead of 'origin/master' by 2 commits.

(use "git push" to publish your local commits)

Changes to be committed:

(use "git reset HEAD <file>..." to unstage)

new file: f4

Untracked files:

(use "git add <file>..." to include in what will be committed)

new

mdodda@GUEST152 MINGW64 /e/MyGitProject (master)

$ git clean -xf

Removing f1

Removing f3

Removing new

mdodda@GUEST152 MINGW64 /e/MyGitProject (master)

$ ll

total 4

-rw-r--r-- 1 mdodda 1049089 75 Jun 26 21:59 del.txt

-rw-r--r-- 1 mdodda 1049089 61 Jun 13 20:13 demo-pull

-rw-r--r-- 1 mdodda 1049089 0 Jun 28 15:17 f2

-rw-r--r-- 1 mdodda 1049089 0 Jun 28 15:34 f4

-rw-r--r-- 1 mdodda 1049089 39 Jun 13 20:13 index.html

drwxr-xr-x 1 mdodda 1049089 0 Jun 27 19:45 newdir/

-rw-r--r-- 1 mdodda 1049089 199 Jun 13 20:13 Prog.java

**Interactive mode or git clean interactive:**

git clean has an "interactive" mode that you can initiate by passing the -i option.

$ **git clean -di**

Would remove the following items:

untracked\_dir/ untracked\_file

\*\*\* Commands \*\*\*

1: clean 2: filter by pattern 3: select by numbers 4: ask each 5: quit 6: help

What now>

The -d option so it will also act upon our untracked\_dir. The interactive mode will display a What now> prompt that requests a command to apply to the untracked files. The commands themselves are fairly self explanatory. We'll take a brief look at each in a random order starting with command 6: help. Selecting command 6 will further explain the other commands:

**6: help**

Show brief usage of interactive git-clean.

mdodda@GUEST152 MINGW64 /e/MyGitProject (master)

$ git clean -di

Would remove the following items:

hh1 newdirec/

\*\*\* Commands \*\*\*

1: clean 2: filter by pattern 3: select by numbers

4: ask each 5: quit 6: help

What now> 6

clean - start cleaning

filter by pattern - exclude items from deletion

select by numbers - select items to be deleted by numbers

ask each - confirm each deletion (like "rm -i")

quit - stop cleaning

help - this screen

? - help for prompt selection

**5: quit**

This lets you quit without do cleaning, will exit the interactive session.

$ git clean -di

Would remove the following items:

hh1 newdirec/

\*\*\* Commands \*\*\*

1: clean 2: filter by pattern 3: select by numbers

4: ask each 5: quit 6: help

What now> 5

Bye.

**1: clean**

Will delete the indicated items. If we were to execute 1: clean at this point untracked\_dir/ untracked\_file would be removed.

mdodda@GUEST152 MINGW64 /e/MyGitProject (master)

$ ll

total 5

-rw-r--r-- 1 mdodda 1049089 75 Jun 26 21:59 del.txt

-rw-r--r-- 1 mdodda 1049089 61 Jun 13 20:13 demo-pull

-rw-r--r-- 1 mdodda 1049089 17 Jun 28 18:38 hh1

-rw-r--r-- 1 mdodda 1049089 39 Jun 13 20:13 index.html

drwxr-xr-x 1 mdodda 1049089 0 Jun 28 18:34 newdirec/

-rw-r--r-- 1 mdodda 1049089 199 Jun 13 20:13 Prog.java

mdodda@GUEST152 MINGW64 /e/MyGitProject (master)

$ git status

On branch master

Your branch is ahead of 'origin/master' by 4 commits.

(use "git push" to publish your local commits)

Untracked files:

(use "git add <file>..." to include in what will be committed)

hh1

newdirec/

nothing added to commit but untracked files present (use "git add" to track)

We can find above there is a file and a directory which are untracked.

$ git clean -di

Would remove the following items:

hh1 newdirec/

\*\*\* Commands \*\*\*

1: clean 2: filter by pattern 3: select by numbers

4: ask each 5: quit 6: help

What now> 1

Removing hh1

Removing newdirec/

**4: ask each**

This will start to clean, and you must confirm one by one in order to delete items.

will iterate over each untracked file and display a Y/N prompt for a deletion. It looks like the following:

\*\*\* Commands \*\*\*

1: clean 2: filter by pattern 3: select by numbers 4: ask each 5: quit 6: help

What now> 4

Remove untracked\_dir/ [y/N]? N

Remove untracked\_file [y/N]? N

**Changing a commit message:**

If a commit message contains unclear, incorrect, or sensitive information, you can amend it locally and push a new commit with a new message to GitHub. You can also amend a commit to add a co-author.

**Rewriting the most recent commit message:**

You can change the most recent commit message using the git commit --amend command.

mdodda@GUEST152 MINGW64 /e/MyGitProject (master)

$ git commit -m "added file mn"

[master 20ce92e] added file mn

1 file changed, 1 insertion(+)

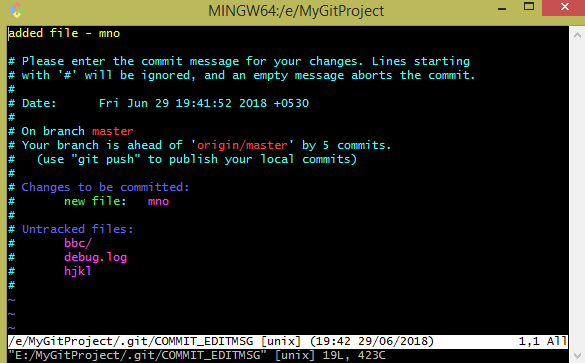
create mode 100644 mno

To change the commit message, use the below command,

mdodda@GUEST152 MINGW64 /e/MyGitProject (master)

$ git commit --amend

Once above command is given, then the editor is opened and there we can change the commit message after entering into the insert mode – “i”



[master f0a6f36] added file - mno

Date: Fri Jun 29 19:41:52 2018 +0530

1 file changed, 1 insertion(+)

create mode 100644 mno

Now check the commit message has been changed.

The above procedure is for the commits those are not pushed online i.e., not pushed into github.

# Amending the message of the most recently pushed commit:

1. Follow the steps above to amend the commit message (check above steps)
2. Use the push –force or -f command to force push over the old commit.

**git push -f origin master**

# Amending older commit messages:

# If you have already pushed the commit to GitHub, you will have to force push a commit with an amended message.

# Procedure to change/amend git commit message after a file is pushed into GitHub:

Open GitHub account > click on commits on left top > click on “older” at the bottom means older commits

Then check for or pick up the commit id by clicking on the yellow mark as shown in the blow screenshot for which you need to change the commit message.



If you need to amend the message for multiple commits or an older commit, you can use interactive rebase, then force push to change the commit history.

1. On the command line, navigate to the repository that contains the commit you want to amend
2. Use the **git rebase -i HEAD~n** command to display a list of the last n commits in your default text editor.

**git rebase -i HEAD~3**  # Displays a list of the last 3 commits on the current branch

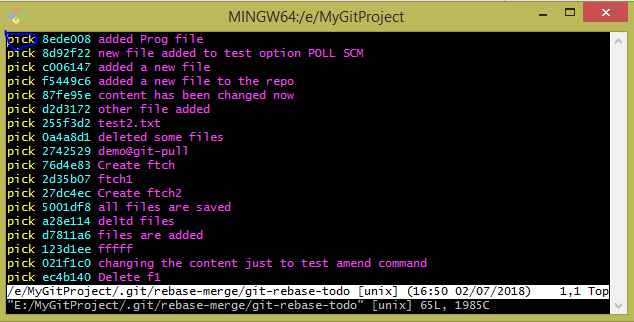
(or) pick the commit from older commits as mentioned in the above procedure from github account.

Use **git rebase** to interactively change the commit history.

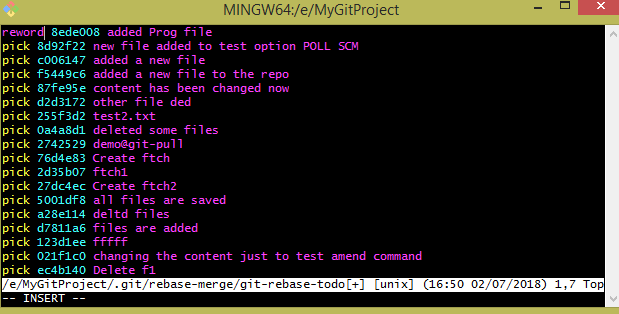
mdodda@GUEST152 MINGW64 /e/MyGitProject (master)

$ git rebase -i ad6eaa69e2e16e1aa2ed8d8a6b5bea2a993f9fa2

The following list will be displayed,



1. Replace pick with reword before each commit message you want to change.



1. Save and close the commit list by typing “:wq” , new editor is opened and can edit the commit message as we need.
2. So in each resulting commit file, type the new commit message, save the file, and close it as shown below. See the differences between Fig (1) and Fig (2)

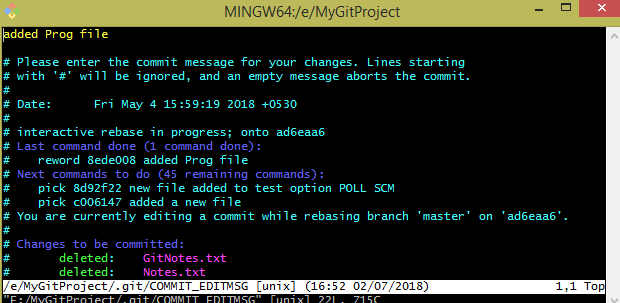
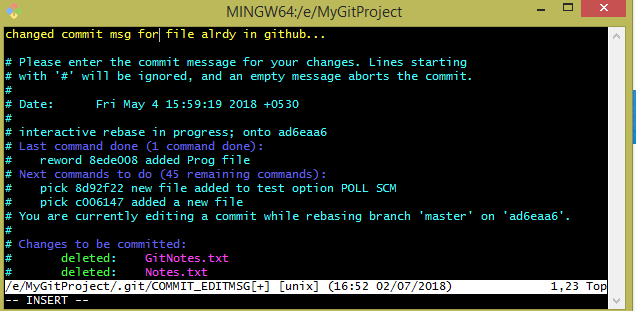


Fig (1)

  
Fig (2)

Once closed the above commit file, we will get a message as “Successfully rebased and updated refs/heads/master.”

1. Force-push the amended commits.

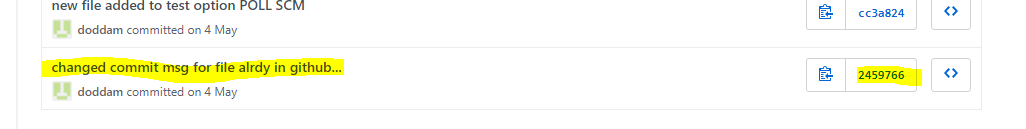
$ git push -f origin master

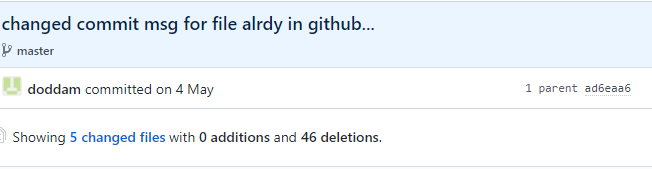
**NOTE:**

As before, amending the commit message will result in a new commit with a new ID. However, in this case, every commit that follows the amended commit will also get a new ID because each commit also contains the id of its parent.

If you have included sensitive information in a commit message, force pushing a commit with an amended commit may not remove the original commit from GitHub. The old commit will not be a part of a subsequent clone; however, it may still be cached on GitHub and accessible via the commit ID.

Check the following screenshots where the commit message has been changed.



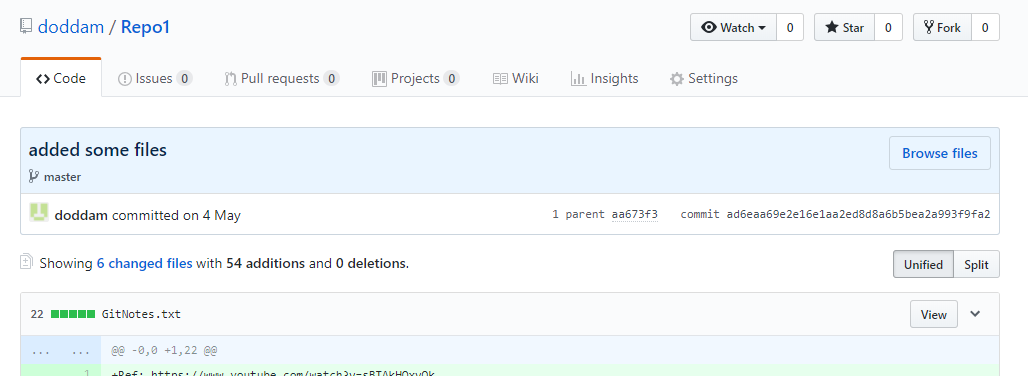


If we still know old commit id we can access it by passing in the url as shown below.

https://github.com/doddam/Repo1/commit/**ad6eaa69e2e16e1aa2ed8d8a6b5bea2a993f9fa2**

New commit id - 245976698169c05cc4ce9fb5e127388206e6540a

Old commit id - ad6eaa69e2e16e1aa2ed8d8a6b5bea2a993f9fa2 still exists as shown below.



**Git rebase?**

Rebasing is the process of moving or combining a sequence of commits to a new base commit.

**Usage**

The primary reason for rebasing is to maintain a linear project history.

**Normal Rebase vs Interactive Rebase:**

The interactive rebase will open an editor with a list of the commits which are about to be changed. This list accepts commands, allowing the user to edit the list before initiating the rebase action.

**Normal Rebase:** Current branch is simply rebase. No feedback taken from the user

**Interactive Rebase:** User has an option to play around with the commits in the current branch. User can re-order commits and other options shown below:

**Interactive rebase commands:**

p, pick = use commit

r, reword = use commit, but edit the commit message

e, edit = use commit, but stop for amending

s, squash = use commit, but meld into previous commit

f, fixup = like "squash", but discard this commit's log message

x, exec = run command (the rest of the line) using shell

d, drop = remove commit

$ **git rebase –i HEAD~4**

You should see a file displayed by your default editor. It should contain a list of commits made:

pick 07e03c2 add 2.txt

pick 5a71aa8 add 3.txt

pick d693749 add 4.txt

# Rebase 785d03b..d693749 onto 785d03b (3 commands)

#

# Commands:

# p, pick = use commit

# r, reword = use commit, but edit the commit message

# e, edit = use commit, but stop for amending

# s, squash = use commit, but meld into previous commit

# f, fixup = like "squash", but discard this commit's log message

# x, exec = run command (the rest of the line) using shell

# d, drop = remove commit

#

# These lines can be re-ordered; they are executed from top to bottom.

#

# If you remove a line here THAT COMMIT WILL BE LOST.

#

# However, if you remove everything, the rebase will be aborted.

#

# Note that empty commits are commented out

NOTE: Commit ids such as “07e03c2” are a stand-in for whatever commit id Git creates for each specific commit.

**pick:**

pick (p for short) is the default action. In this case it would reapply the commit as is, no changes in its contents or message. Saving (and executing) this file would make no changes to the repository.

**reword:**

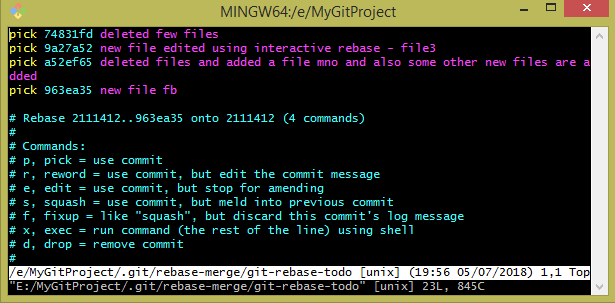
If I say reword (r for short)

If I want to change a commit message.

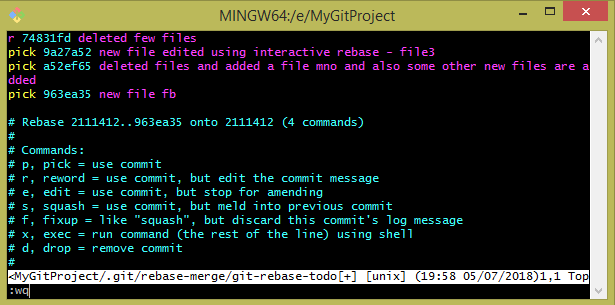
Use the command as discussed above

$ **git rebase –i HEAD~4**

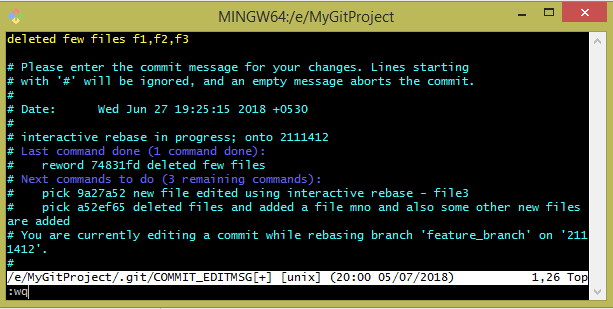
Which displays the last four commit messages.



Now, place letter “r” wherever you wanted to change the commit message instead of command pick > next save & exit the editor



Another window is opened, there commit message can be edited > save and quit the editor.



Check the output in the below screenshot.



**edit:**

Git gets to the edit 9690473 operation, stops, and prints the following message to the terminal:

You can amend the commit now, with

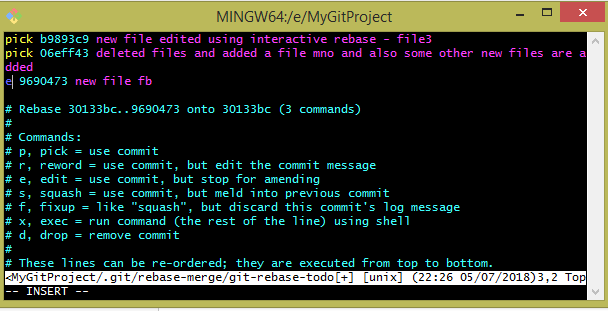
$ **git commit --amend**

Once you are satisfied with your changes, run

$ **git rebase --continue**

At this point, you can edit any of the files in your project to make any additional changes. For each change you make, you'll need to perform a new commit, and you can do that by entering the git commit --amend command. When you're finished making all your changes, you can run git rebase --continue.

$ git rebase -i HEAD~3



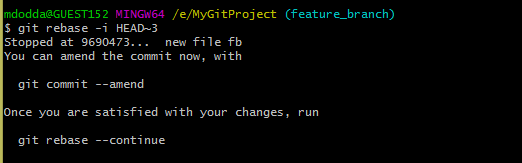
Steps:

After issuing the below command, file editor is opened then edit the commit message - save & exit.

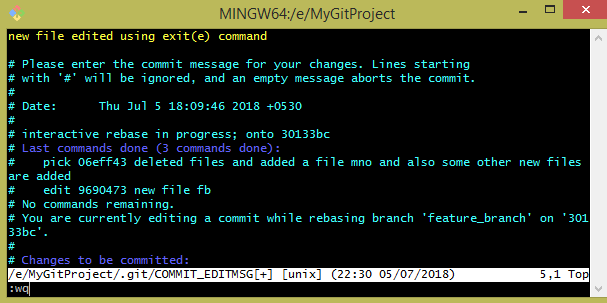
$ git commit - - amend

Then run the below command,

$ git rebase - -continue



$ git commit --amend



output:

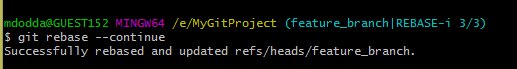
[detached HEAD 24a3862] new file edited using exit(e) command

Date: Thu Jul 5 18:09:46 2018 +0530

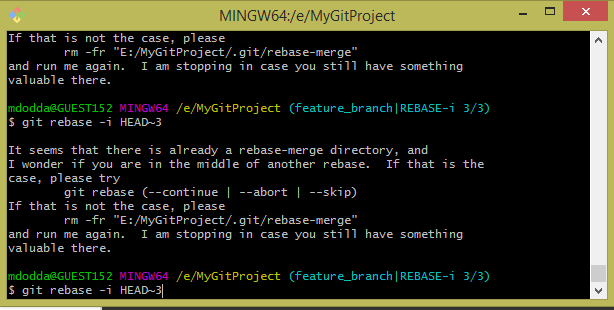
1 file changed, 1 insertion(+)

create mode 100644 fb

$ git rebase –continue



I was working with edit interactive rebase command but forgot to continue it, so below message is displayed.

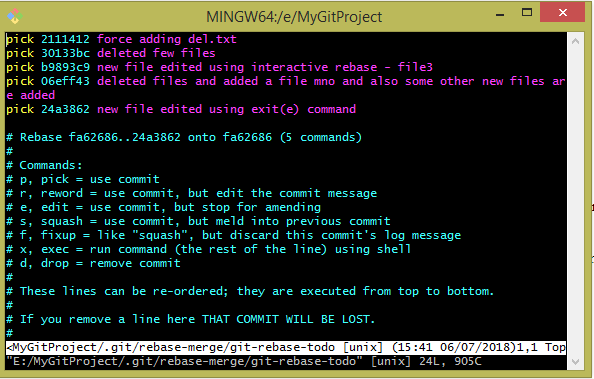


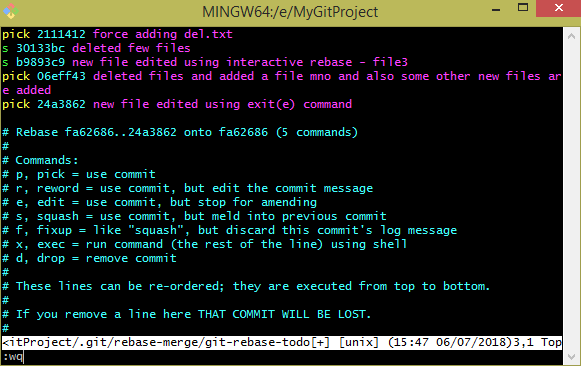
**squash:**

Squash “s” allows you to specify which commits you want to merge into the previous commits.

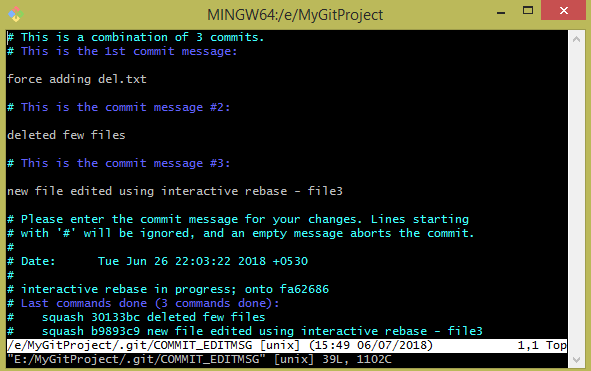
In the case of squash commits, Git will open your configured text editor and prompt to combine the specified commit messages.

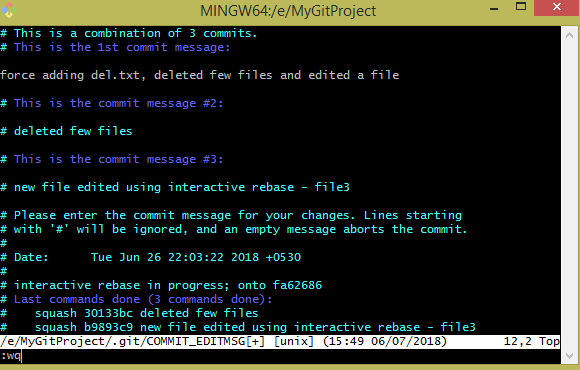
$ **git rebase -i HEAD~5**



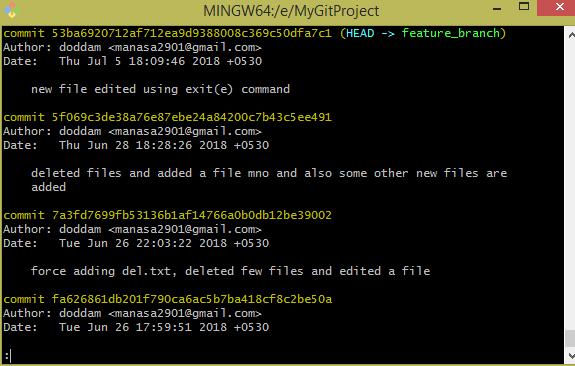


Once you save and exit from the window the other editor is opened as shown below,





You can now check with the command **$ git log**

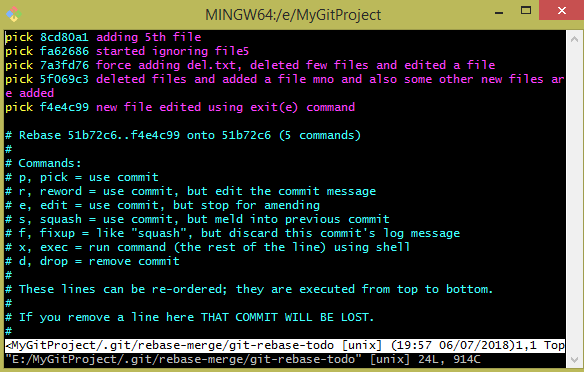


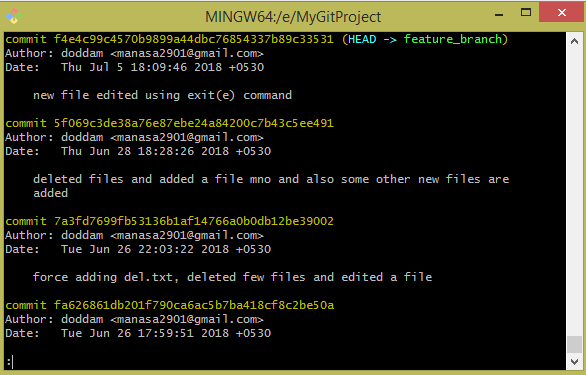
**NOTE:**

Note that the commits modified with a rebase command have a different ID than either of the original commits. Commits marked with pick will have a new ID if the previous commits have been rewritten.

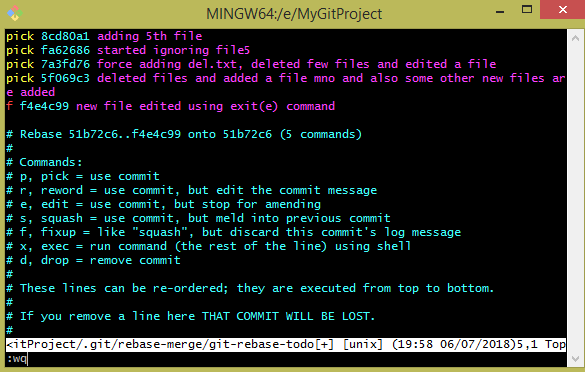
**fixup:**

Fixup or 'f' has the same combining effect as squash. Unlike squash, fixup commits will not interrupt rebase playback to open an editor to combine commit messages. The commits marked 'f' will have their messages discarded in-favor of the previous commit's message.

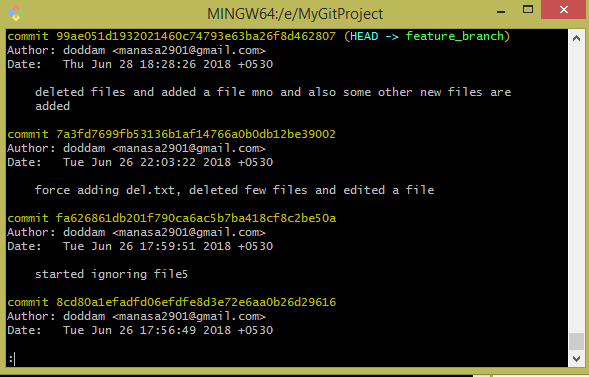




Change the command where you need to discard the commit’s log message.

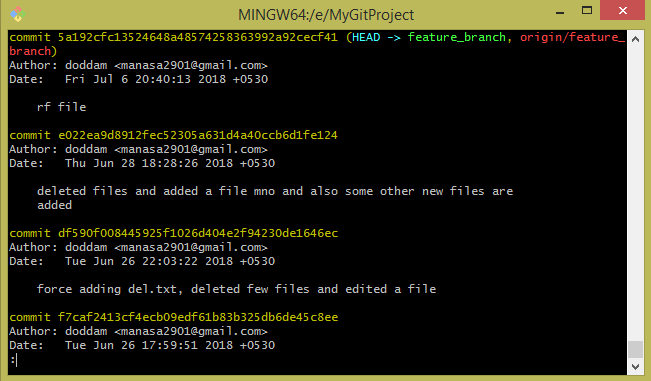


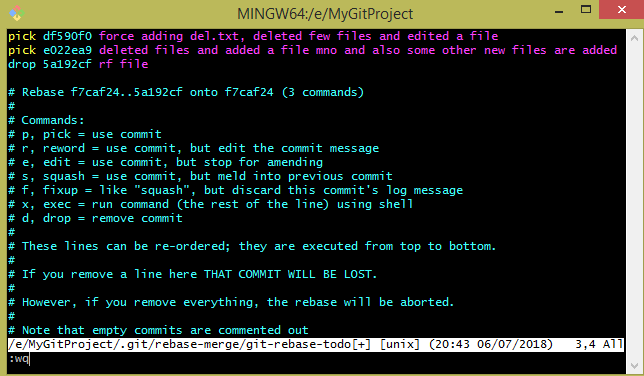
After interactive rebase command is applied just check out the logs using **git log** command in order to get a clear picture.

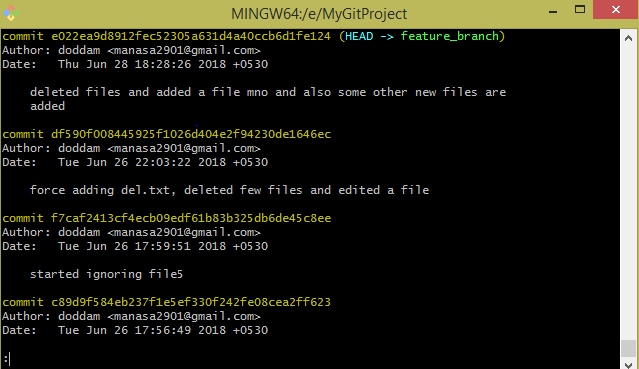


**drop:**

Created a new file “rf” to verify drop command, this command is used to remove the commit.







**Git rebase interactive drop vs deleting the commit line:**

Marking a commit as drop is the same as deleting the line.

If you remove a line in the file editor THAT COMMIT WILL BE LOST.

NOTE:

mdodda@GUEST152 MINGW64 /e/MyGitProject (feature\_branch)

$ git rebase -i HEAD~2

Stopped at 5a192cf... rf file

You can amend the commit now, with

git commit --amend

Once you are satisfied with your changes, run

git rebase --continue

mdodda@GUEST152 MINGW64 /e/MyGitProject (feature\_branch|REBASE-i 2/2)

$ git rebase -i

It seems that there is already a rebase-merge directory, and

I wonder if you are in the middle of another rebase. If that is the

case, please try

git rebase (--continue | --abort | --skip)

If that is not the case, please

rm -fr "E:/MyGitProject/.git/rebase-merge"

and run me again. I am stopping in case you still have something

valuable there.

**git reset:**

To properly understand git reset usage, we must first understand Git's internal state management systems. Sometimes these mechanisms are called Git's "**three trees**".

**git reset** is a powerful command that is used to **undo local changes** to the state of a Git repo. Git reset operates on "The Three Trees of Git". These trees are the Commit History ( HEAD ), the Staging Index, and the Working Directory.

**The working directory**

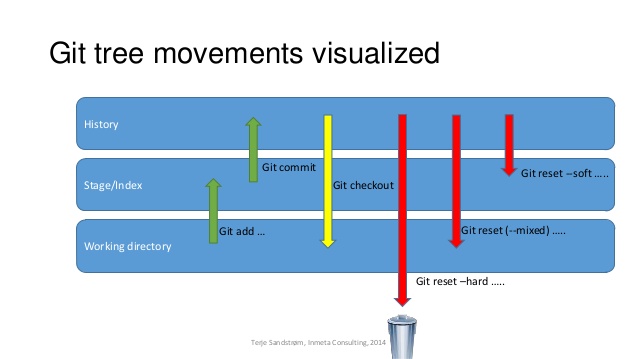
The Working Directory, this tree is in sync with the local filesystem i.e., the current set of files you’re working on in your file system.

**Staging index**

This tree is tracking Working Directory changes, that have been promoted with git add, to be stored in the next commit. This tree is a complex internal caching mechanism. Git generally tries to hide the implementation details of the Staging Index from the user.

**Commit history**

The final tree is the Commit History. The git commit command adds changes to a permanent snapshot that lives in the Commit History. This snapshot also includes the state of the Staging Index at the time of commit.



**HEAD**

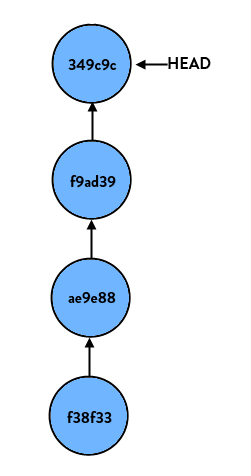
HEAD is a reference that points to such commit objects. It is merely a pointer that updates with every new commit or change of branch. You can move it around and target any point in time with the commit object. This is an alias for the tip of the current branch, which is the most recent commit you have made to that branch.

When you do a **git add**, it stages the file in the index.

When you then perform a **git commit**, Git creates a new commit that HEAD now points to.

**Reset:**

Reset command does is reset HEAD (the tip of the current branch) to another commit. For instance, say we have a branch (let’s call this one “feature”) and it looks like so:



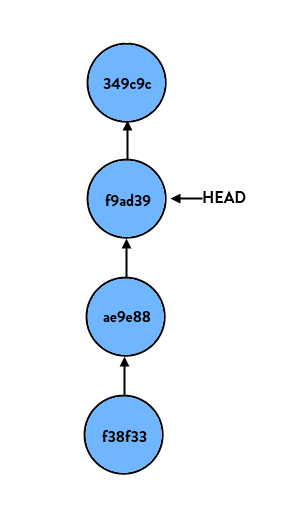
If we perform:

> **git reset HEAD**

Nothing happens. This is because we tell git to reset this branch to HEAD, which is where it already is. But if we do:

> **git reset HEAD~1**

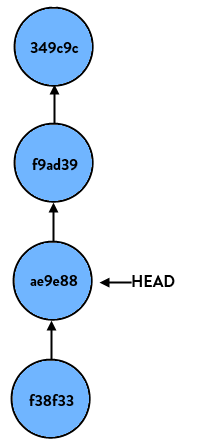
(HEAD~1 is shorthand case for “the commit right before HEAD”, or put differently “HEAD’s parent”) our branch now looks like so:



If we start at the latest commit again and do:

> **git reset HEAD~2**

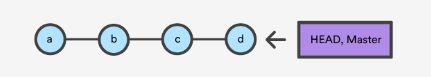
our branch would look like so:



Again, all it does on a basic level is move HEAD to another commit.

**How Reset works?**

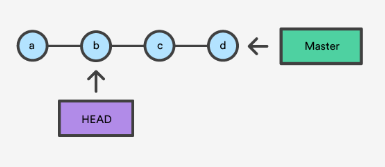
git reset is similar in behavior to git checkout. Where git checkout solely operates on the HEAD ref pointer,**git reset will move the HEAD ref pointer and the current branch ref pointer.** To better demonstrate this behavior consider the following example:



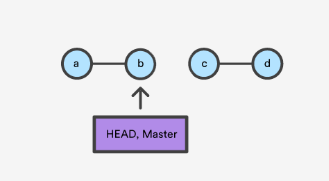
This example demonstrates a sequence of commits on the master branch. The HEAD ref and master branch ref currently point to commit d. Now let us execute and compare, both git checkout b and git reset b.

**git checkout b**

With git checkout, the master ref is still pointing to d. The HEAD ref has been moved, and now points at commit b. The repo is now in a 'detached HEAD' state.



**git reset b**



Comparatively, git reset, moves both the HEAD and branch refs to the specified commit.

In addition to updating the commit ref pointers, git reset will modify the state of the three trees. The ref pointer modification always happens and is an update to the third tree, the Commit tree. The command line arguments --soft, --mixed, and --hard direct how to modify the Staging Index, and Working Directory trees.

**Main options:**

The default invocation of git reset has implicit arguments of --mixed and HEAD. This means executing git reset is equivalent to executing git reset --mixed HEAD.

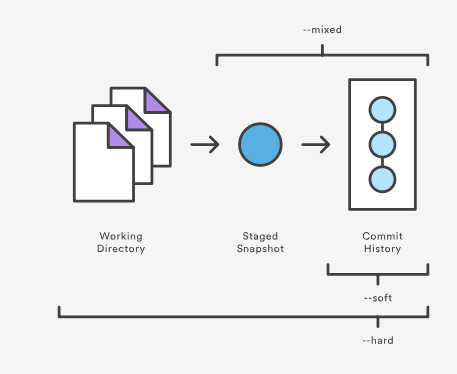


fig: The scope of git reset modes

**git reset [<mode>] [<commit>]**

This form resets the current branch head to <commit> and possibly updates the index (resetting it to the tree of <commit>) and the working tree depending on <mode>. If <mode> is omitted, defaults to "--mixed". The <mode> must be one of the following:

**- -soft**

Does not touch the index file or the working tree at all (but resets the head to <commit>, just like all modes do). This leaves all your changed files "Changes to be committed", as git status would put it. (Moves all the changes into staging area)

Eg: Undo a commit

$ **git reset - -soft HEAD^**

**HEAD vs ORIG\_HEAD**

HEAD is (direct or indirect, i.e. symbolic) reference to the current commit.

ORIG\_HEAD is previous state of HEAD, set by commands that have possibly dangerous behavior, to be easy to revert them. It is less useful now that Git has reflog: HEAD@{1} is roughly equivalent to ORIG\_HEAD (HEAD@{1} is always last value of HEAD, ORIG\_HEAD is last value of HEAD before dangerous operation).

Eg: **Undo a commit and redo**

$ git commit ...

$ git reset --soft HEAD^ (1)

$ edit (2)

$ git commit -a -c ORIG\_HEAD (3)

(1) This is most often done when you remembered what you just committed is incomplete, or you misspelled your commit message, or both. Leaves working tree as it was before "reset".

(2) Make corrections to working tree files.

(3) "reset" copies the old head to .git/ORIG\_HEAD; redo the commit by starting with its log message. If you do not need to edit the message further, you can give -C option instead.

i.e., **$ git commit -C ORIG\_HEAD**

**- -mixed (default option)**

Resets the index but not the working tree (i.e., the changed files are preserved but not marked for commit) and reports what has not been updated. This is the default action.

(git reset - -mixed - moves all the changes into working directory)

Eg: Undo add

We know that, file is added from working tree to index/staging area using $ git add command.

$ **git reset**

**Reset a single file in the index:**

Suppose you have added a file to your index, but later decide you do not want to add it to your commit. You can remove the file from the index while keeping your changes with git reset.

$ git reset - - filename (1)

$ git commit -m "Commit files in index" (2)

$ git add filename (3)

(1)This removes the file from the index while keeping it in the working directory.

(2)This commits all other changes in the index.

(3)Adds the file to the index again.

**- -hard**

Resets the index and working tree. Any changes to tracked files in the working tree since <commit> are discarded. (Move the changes into trash)

Eg: Undo commits permanently

$ **git reset --hard HEAD~1**

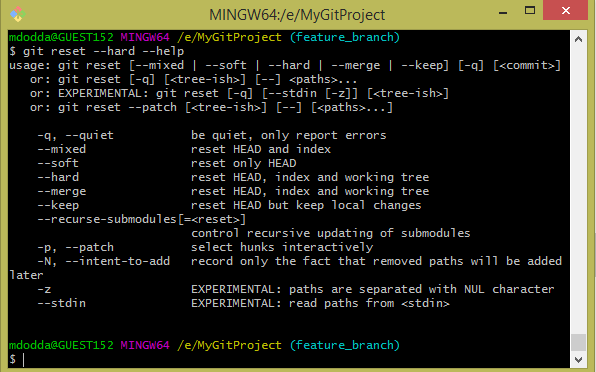
**- -merge**

Resets the index and updates the files in the working tree that are different between <commit> and HEAD, but keeps those which are different between the index and working tree (i.e. which have changes which have not been added). If a file that is different between <commit> and the index has unstaged changes, reset is aborted.

Eg: Undo a merge or pull inside a dirty working tree

Pulled files from github remote using $ **git pull**

but now if needed to undo a merge or pull use $ **git reset --merge ORIG\_HEAD**

Options for “git reset”

Eg: git commit -o filename > commit only specified files

file editor is opened > add the commit message.

**Snapshot:**

A snapshot is the state of something (e.g. a folder) at a specific point in time. In this case, snapshot means the current content of the test branch.

**Scenario:** Interrupted workflow

Suppose you are interrupted by an urgent fix request while you are in the middle of a large change. The files in your working tree are not in any shape to be committed yet, but you need to get to the other branch for a quick bug fix.

$ git checkout feature ;# you were working in "feature" branch and

$ work work work ;# got interrupted

$ git commit -a -m "snapshot WIP" (1)

$ git checkout master

$ fix fix fix

$ git commit ;# commit with real log

$ git checkout feature

$ git reset --soft HEAD^ ;# go back to WIP state (2)

$ git reset (3)

(1) This commit will get blown away so a throw-away log message is OK.

(2) This removes the WIP commit from the commit history, and sets your working tree to the state just before you made that snapshot.

(3) At this point the index file still has all the WIP changes you committed as snapshot WIP. This updates the index to show your WIP files as uncommitted.

**Undoing/Reverting/Resetting code changes:**

***Undoing:***

Undo uncommitted changes/to discard changes in the working directory using the command $ **git checkout - - filename**

A single line in the code can be removed but when you have 100 lines of code it is difficult to undo changes manually so use git checkout

* I have a file namely “hungry” which is already committed.
* I have modified it and I did not committed the file after modification.
* Now I want to undo the changes.
* This can done by the command as the file is not committed,

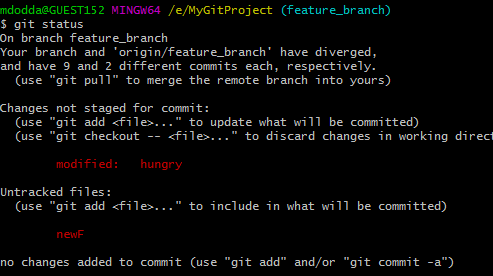
$ **git checkout -- filename**

$ git checkout -- hungry

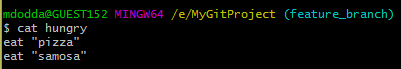
**Content in the file “hungry”**



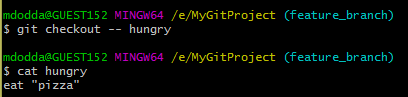
**Added an extra line in the file “hungry”**



**Now check the content**

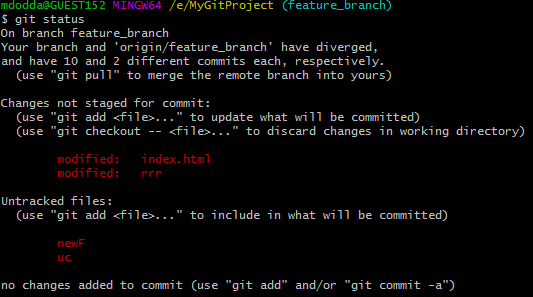


**I want to undo the changes. As my work is uncommitted, I used the following command.**

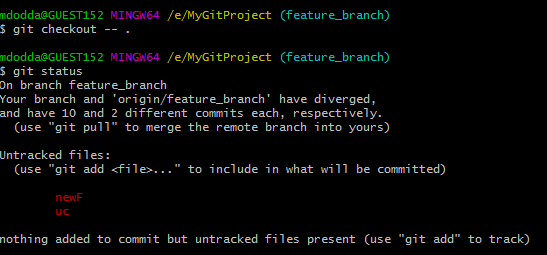


***Undoing multiple files at a time:***

I have two files that are modified, now they needed to be bought to the old state,



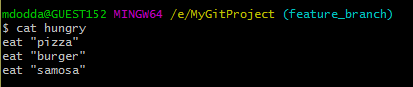
Command used for undoing multiple uncommitted files at a time $ **git checkout - - .**



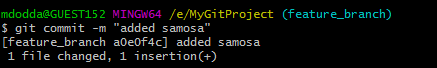
***Reverting:***

Undo committed changes using **revert.**

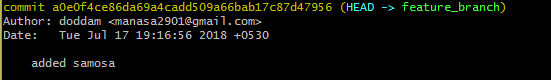
Had a file “hungry” and committed it





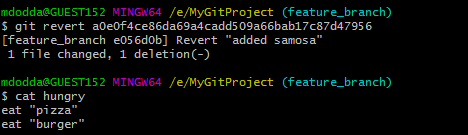


Check the log using **$ git log**

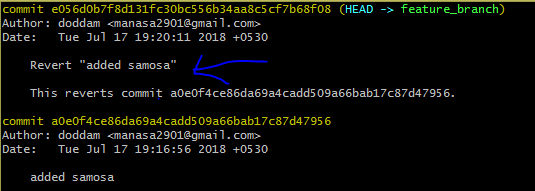


Now, I want to undo the change using $ **git revert commit\_id**

Need to mention commit\_id as the file is already committed.

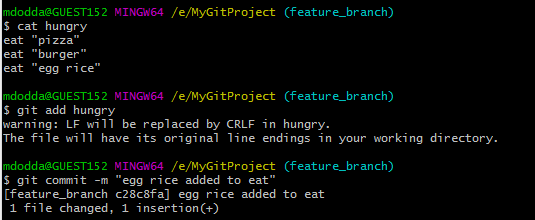


Once git revert is used the changes are committed directly, check below logs.

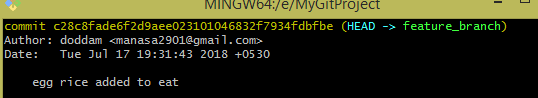


But now, I want to commit the changes explicitly so I go with the option -n in the command $ **git revert -n commit\_id**

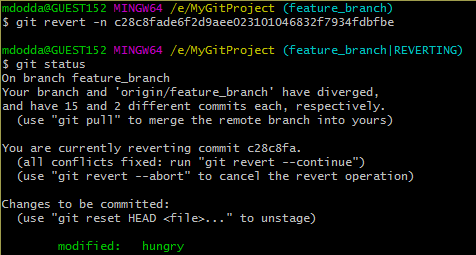
Added a line in the file



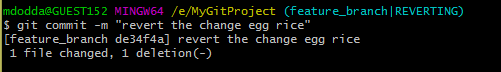
Check the log

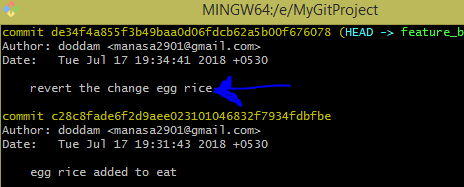


Undo the line added recently using git revert and check the status that the file needs to be committed.



Committed the file explicitly by using option -n



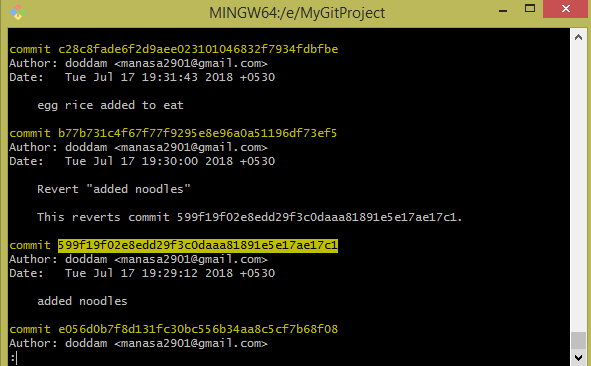
Below is the log to check

***Resetting:***

git reset is a powerful command, need to be careful with this command as data may be destructed, if any small mistake is done.

For suppose if you want to go back to the previous commit, just note down the commit id of the particular commit in the log and get back to it(i.e., particular code)

In the code evaluation, I wanted to go to the status “noodles” as I realized that code after that is not needed so I used the following the command “$ git reset --hard commit\_id” to undo the changes.



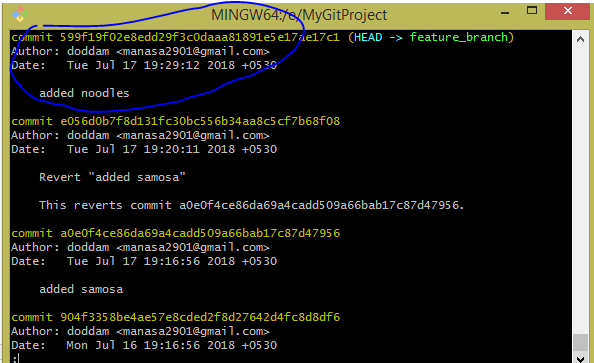
Copy the commit\_id of the state where you need to go,

$ git reset --hard 599f19f02e8edd29f3c0daaa81891e5e17ae17c1

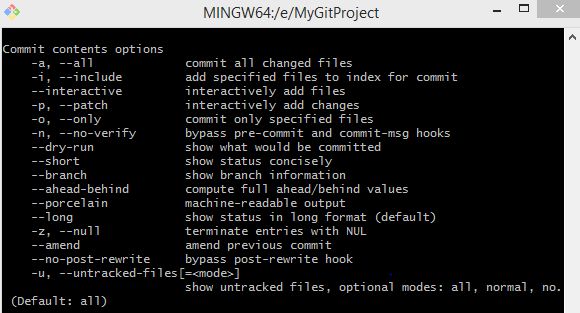
Option - - hard can take to any previous timestamp in the history, but this is destructive.



Check the log,



**Note:**All the options that are available for “commit”



**Revert vs Reset:**

**Git revert** command is used to revert a particular commit.

This internally creates a separate commit for reverting a previous commit.

**Git reset** on the other hand is primarily used to reset the current HEAD to the specified state.

***git reset is also very helpful in the following scenarios,***  
  
1. When you have made some changes and you no longer wanted them to be committed you just wanted to go back to the older clean state  
$ **git reset --hard HEAD**2. When you wanted to go to two commits back  
$ **git reset --hard HEAD~2**3. When you are doing git pull. It might result in merging errors. In this case you might just want to get rid of the mess, then you can simply use   
$ **git reset --hard HEAD**4. You might have added a file to commit using git add abc.c but later you realized you do not want to add then you can use the following  
$ **git reset -- abc.c**

### **What is Git cherry-pick?**

Cherry-pick simply means picking a commit from a branch and applying that commit onto another branch. It is more useful for sampling out a small subset of changes from a topic branch you've decided to delete, but you still got some useful commits on it. This will introduce a new, distinct commit.

**Use cases of cherry-pick:**

(1) Cherry-picking becomes very useful and essential when it comes to **bug fixing**. This is because bugs are fixed and tested on the development branch with their respective snapshots (committed) - which might have been on a different level from your production branch.

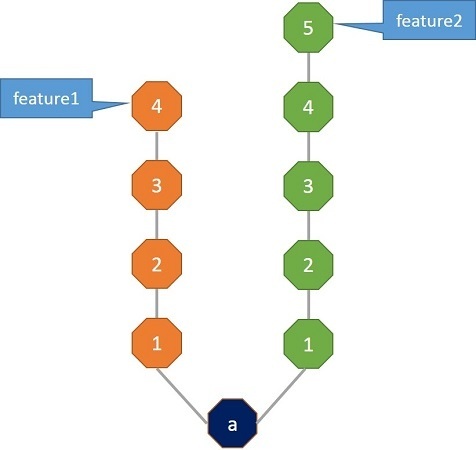
As with other Git operations that introduce changes via the process of applying a diff, you may need to resolve conflicts to fully apply the changes from the given commit. This could be averted by doing a git cherry-pick.

(2) This could also be useful whenever a full branch merge is not possible due to incompatible versions in the various branches.

(3) Also, you can use git cherry-pick to pull the changes introduced to a sub-branch, without changing the branch, by your colleague working on the same project.

Let’s just say you have two branches *feature1*and *feature2*as in the following picture.

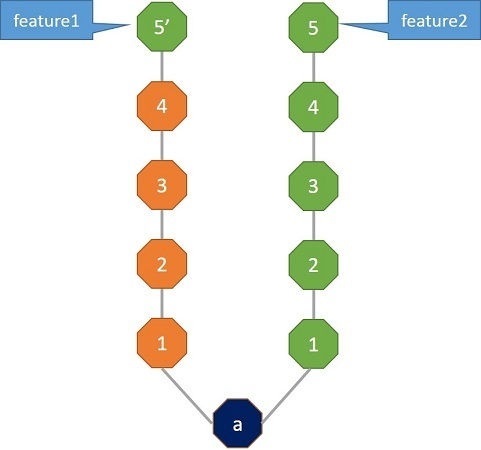
Now, the green commit 5 on branch 2, has some interesting code that you want on feature1. How would you get that? One way would be to merge/rebase the branch. But with that you will get all the other green commits from 1–4 as well, which you don’t want. Cherry-pick for the rescue.



Assuming you are on feature1, all you have to say is

$ git cherry-pick green5 (Assuming 'green5' is the commit id)

And that’s it. You will have the green5 commit on your orange4 commit like in this picture as you wanted.



Notice, that the green commit is no longer “5” but has been changed to “5′”. This is to show that, though the changes in the commit remain the same, Git will generate a new commit hash for this because hashes take parent node also into account. So, that is cherry-pick in short.

**Advantage of cherry-pick over merge:**

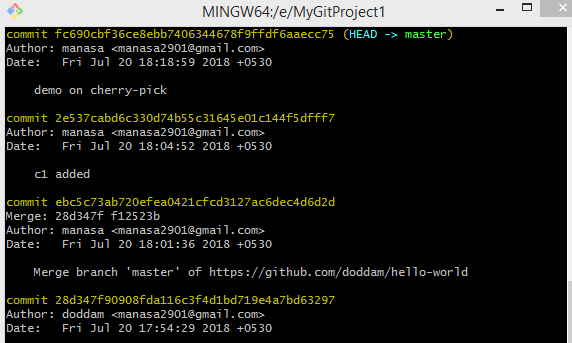
No need to have the whole branch with green commits on branch1, so can have only the commit required.

**Why cherry-pick**: Can have required feature on the required branch

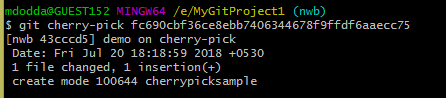
**Why can't merge:** This will merge whole code from branch2 to branch1 where unnecessary code is moved where more space is consumed.

Example:

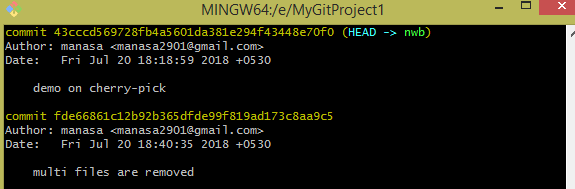
Consider a master branch and following is the log history of the branch,



Now I wanted tochoose a commit from one branch and apply it onto another (i.e., from “master” to “nwb”)

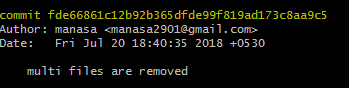


You can compare logs of two branches:



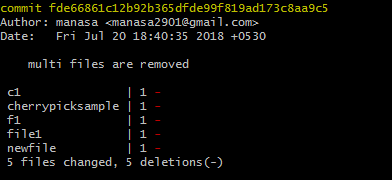
**Viewing commit history:**

$ git log



If you want to see some abbreviated stats for each commit, you can use the “--stat” option:

$ git log --stat



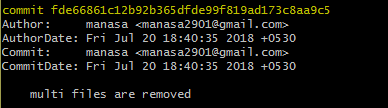
As you can see, the **--stat** option prints below each commit entry a list of modified files, how many files were changed, and how many lines in those files were added and removed. It also puts a summary of the information at the end.

Another useful option is --pretty. This option changes the log output to formats other than the default. A few prebuilt options are available for you to use. **The oneline option prints each commit on a single line**, which is useful if you’re looking at a lot of commits. In addition, the short, full, and fuller options show the output in roughly the same format but with less or more information, respectively:

$ git log --pretty=oneline



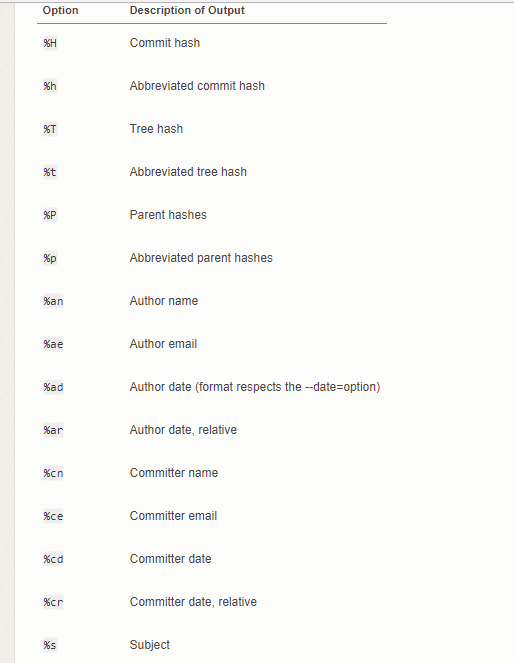
$ git log --pretty=fuller



$ git log --pretty=format:"%h - %an, %ar : %s"



Where the useful options for git log --pretty=format



**More options for git log:**

git log can be a more effective way of searching for text across the branches, especially if there are many matches, and you want to see more recent (relevant) changes first.

$ git log -p --all -S 'search string'

$ git log -p --all -G 'match regular expression

These log commands list commits that add or remove the given search string/regex, (generally) more recent first. The -p option causes the relevant diff to be shown where the pattern was added or removed, so you can see it in context.

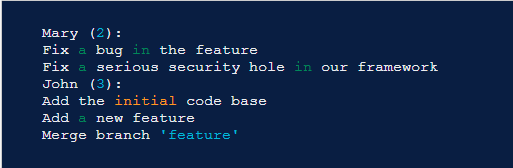
Having found a relevant commit that adds the text you were looking for (eg. 8beeff00d), find the branches that contain the commit:

$ git branch -a --contains <commit-id>

$ git branch -a --contains 8beeff00d

The git shortlog command is a special version of git log intended for creating release announcements. It groups each commit by author and displays the first line of each commit message. This is an easy way to see who’s been working on what.

For example, if two developers have contributed 5 commits to a project, the git shortlog output might look like the following:



**Other example of three commits:**

manasa (3):

adding test file

added index.html | modified test1.txt

merge new file

By default, git shortlog sorts the output by author name, but you can also pass the -n option to sort by the number of commits per author.

**Filtering the Commit History:**

***By Amount:***

The most basic filtering option for git log is to limit the number of commits that are displayed. When you’re only interested in the last few commits, this saves you the trouble of viewing all the commits in a page.

$ git log -3

***By Date:***

If you’re looking for a commit from a specific time frame, you can use the --after or --before flags for filtering commits by date. These both accept a variety of date formats as a parameter. For example, the following command only shows commits that were created *after* July 23rd, 2018 (inclusive):

$ git log --after=”2018-7-23”

***By relative references:***

You can also pass in relative references like "1 week ago" and "yesterday":

$ git log --after=”yesterday”

To search for a commits that were created between two dates, you can provide both a --before and --after date. For instance, to display all the commits added between July 11th, 2018 and July 25th, 2018, you would use the following:

$ git log --after="2018-7-11" --before="2018-7-25"

Note that the --since and --until flags are synonymous with --after and --before, respectively.

***By Message:***

To filter commits by their commit message, use the --grep flag. This works just like the --author flag discussed above, but it matches against the commit message instead of the author.

For example, if your team includes relevant issue numbers in each commit message, you can use something like the following to pull out all of the commits related to that issue:

***By File:***

Many times, you’re only interested in changes that happened to a particular file. To show the history related to a file, all you have to do is pass in the file path. For example, the following returns all commits that affected either the file foo.py or the bar.py file:

$ git log -- foo.py bar.py

The -- parameter is used to tell git log that subsequent arguments are file paths and not branch names. If there’s no chance of mixing it up with a branch, you can omit the --.

***By Range:***

You can pass a range of commits to git log to show only the commits contained in that range. The range is specified in the following format, where <since> and <until> are commit references: $ git log --since="june" --until="july"

Viewing all commits of a branch from any other branch:

$ git log <branch-name>

Created an orphan branch:

git checkout - -orphan <branch-name>

**Patch:**

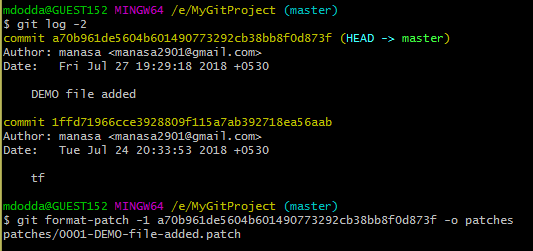
A patch is a small file that indicates what was changed in a repository. It's generally used when someone from outside your team has read-only access but had a good code change available. He then creates a patch and sends it to you. You apply it and push it to the git repository.

Patch is a text file, whose contents are similar to Git diff, but along with code, it also has metadata about commits; e.g., commit ID, date, commit message, etc. We can create a patch from commits and other people can apply them to their repository.

**When can we use patch:**

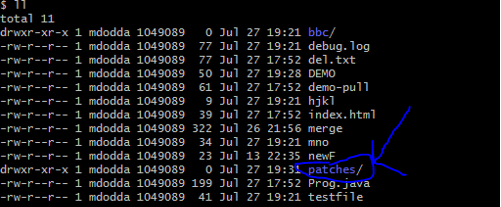
Patch is used in the same situation where cherry-pick is used (same scenario but different solution) when we need to apply commit id to one branch after picking up to other branch.

Select the commit id from the branch using git log which needed to be applied on the other branch



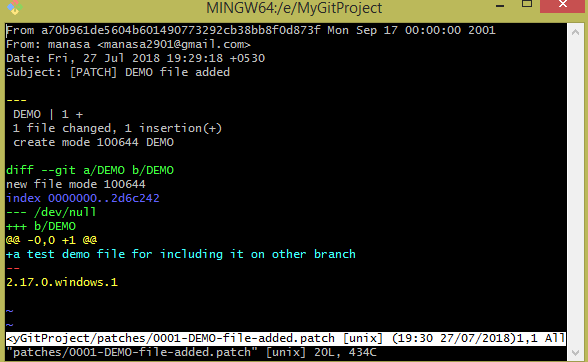
The above git command exports the commit as a patch file which can applied to another branch or common repository

A Patch directory is created in ‘master branch’

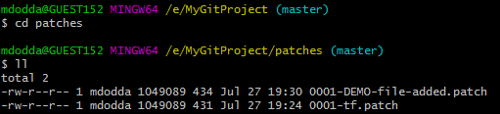


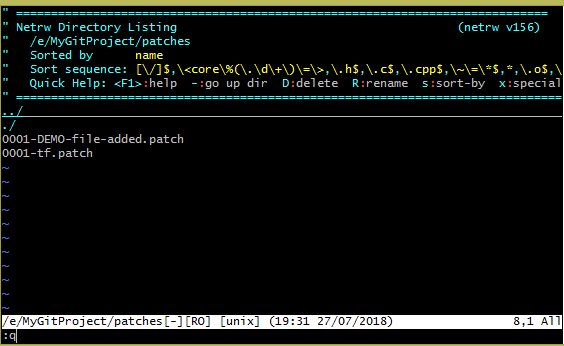
Now view the file which is in patches directory,





Or navigate to the patch directory and view required file in the patch,

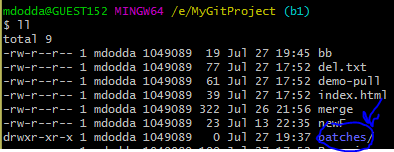


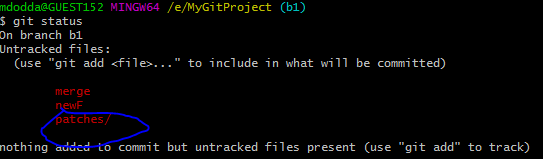
$ vim patches

Now checkout to the branch, where you need to apply the patch:

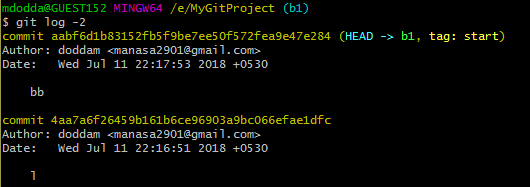
$ git checkout b1

When you check out the branch you can find patches directory, it is found in untracked state.





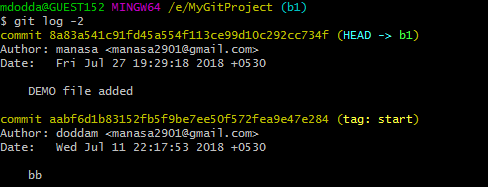
Check the log,



Applying the patch, **am - applies patch to the current branch**



Check the log once again to check whether patch is applied or not:



**In what scenario do we use patch?**

Instead of merging two branches where whole code of one branch moved onto other branch as more space is consumed instead single commit-id is taken and applied on other branch.

**gitk:**

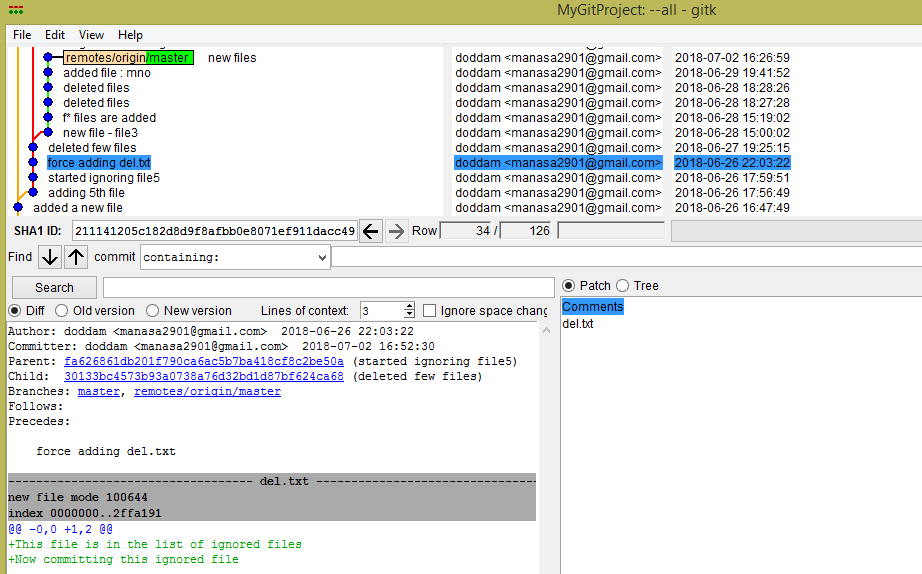
The Git repository browser

**SYNOPSIS:**

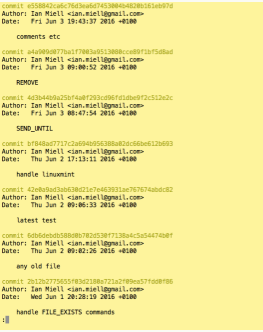
*gitk* [<options>] [<revision range>] [--] [<path>…​]

**DESCRIPTION**

Displays changes in a repository or a selected set of commits. This includes visualizing the commit graph, showing information related to each commit, and the files in the trees of each revision.



You already know **git log,** the command to show the history from your current point is and the output is most-recent commit first, down to the oldest.



**oneline:** Most of the time I don’t care about the author or the date, so in order that I can see more per screen, I use - -oneline to only show the commit id and comment per-commit.

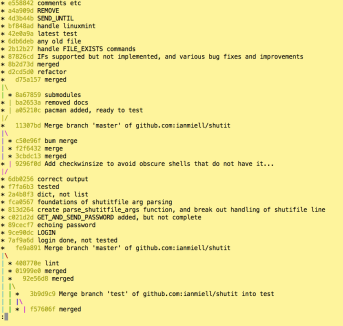




Let’s check out other options:

The problem with the above is that you only see a linear series of commits, and get no sense of what was merged in where. To see this aspect of the history use - -graph.



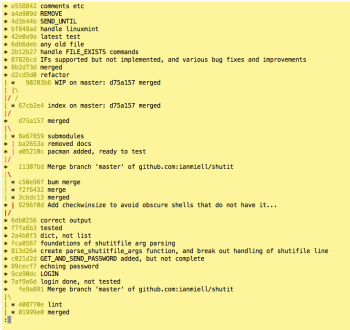


You can see where merges take place, and what commits were merged.

**- -all**

By default you only get the history leading up to the HEAD i.e., where you are currently in the git history. Often I want to see **all the branches in the history**, so I add the –all flag.

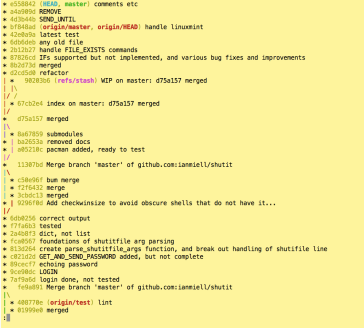




**– -decorate**

But I can’t see what branch is where! This is where you use – -decorate.





Each remote or type of branch/tag is shown in a different color (even stashes!). On my terminal, remotes are in red, HEAD is blue, local branches are in green, stashes in pink.

**Refs**

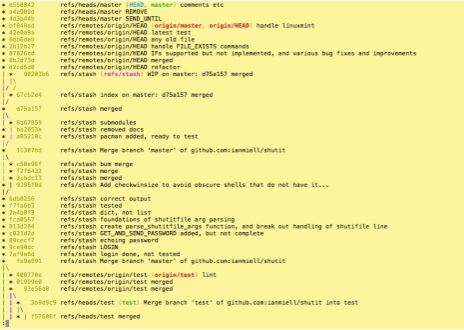
A ref is an indirect way of referring to a commit. You can think of it as a user-friendly alias for a commit hash. This is Git’s internal mechanism of representing branches and tags.

Refs are stored as normal text files in the .git/refs directory, where .git is usually called .git. To explore the refs in one of your repositories, navigate to .git/refs. You should see the following structure, but it will contain different files depending on what branches, tags, and remotes you have in your repo:



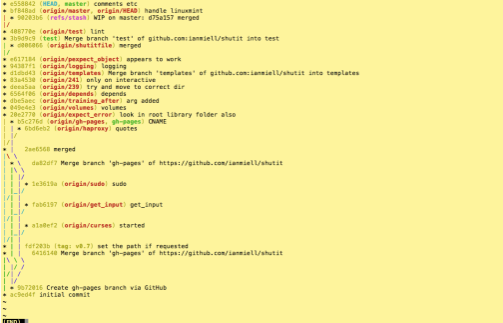
If you want, you can show the ref name on each line by adding - -source, but I usually find this to be overkill:





**- -simplify-by-decoration**

If you’re looking at the whole history of your project, you may want to see only the significant points of change (i.e. the lines affected by - -decorate above) to eliminate all the intermediary commits. This is perfect for getting an overview of the project as a whole.

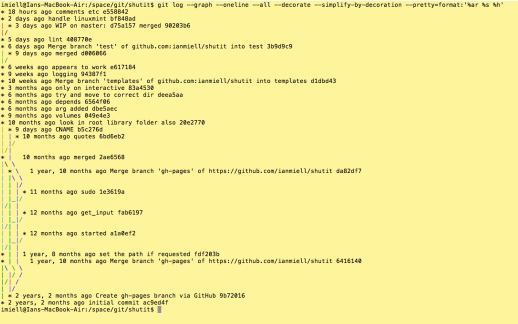
$ git log --graph --oneline --all --decorate **--simplify-by-decoration**

**- -pretty**

When viewing the whole history of the project in this way, you might want to re-introduce the Date info with --pretty=

$ git log --graph --oneline --all --decorate --simplify-by-decoration **--pretty='%ar %s %h'**

This gives a formatted output, showing (in this case) the relative timestamp (%ar), the commit subject (%s), and the short hash (%h).



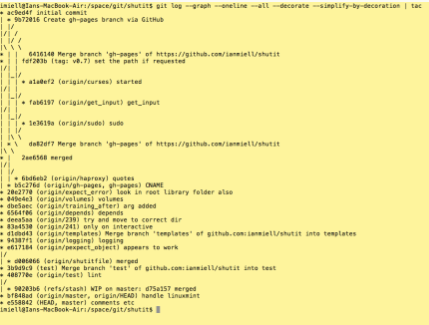
You can even see the **abstract shape** of the git repo!

$ git log --graph --oneline --all --decorate --simplify-by-decoration **--pretty=**



To show commits from oldest to newest:

$ git log --graph --oneline --all --decorate --simplify-by-decoration **| tac**



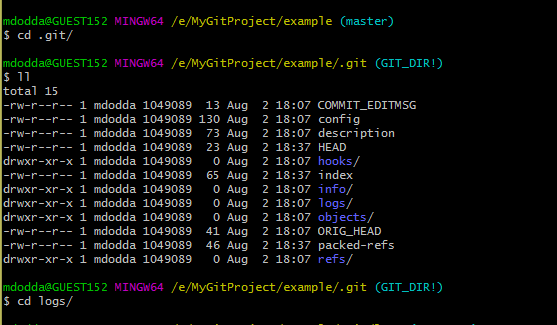
**git reflog:**

Git keeps track of updates to the tip of branches using a mechanism called **reference logs**, or "**reflogs**."

Many Git commands accept a parameter for specifying a reference or "ref", which is a pointer to a commit.

Reflogs track when Git refs were updated in the local repository.

In addition to branch tip reflogs, a special reflog is maintained for the Git stash.



Reflogs are stored in directories under the local repository's .git directory. git reflog directories can be found at .git/logs/refs/heads/., .git/logs/HEAD, and also .git/logs/refs/stash if the git stash has been used on the repo.

**Basic usage:**

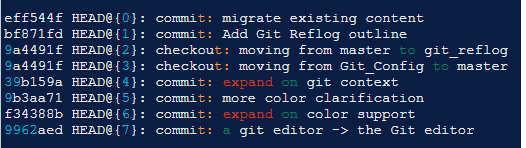
The most basic Reflog use case is invoking:

$ **git reflog**

This is essentially a short cut that's equivalent to:

$ **git reflog show HEAD**

This will output the HEAD reflog. You should see output similar to:



**Reflog references:**

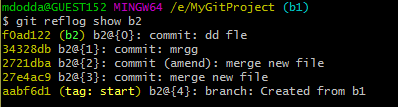
By default, git reflog will output the reflog of the HEAD ref. HEAD is a symbolic reference to the currently active branch. Reflogs are available for other refs as well. The syntax to access a git ref is name@{qualifier}. In addition to HEAD refs, other branches, tags, remotes, and the Git stash can be referenced as well.

You can get a complete reflog of all refs by executing:

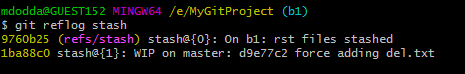
$ **git reflog show --all**

To see the reflog for a specific branch pass that branch name to git reflog show:

$ **git reflog show branch-name**



The following example assumes you have previously stashed some changes using the git stash command.



This will output a reflog for the Git stash. The returned ref pointers can be passed to other Git commands:

$ **git diff stash@{0} b1@{0}**

When executed, this example code will display Git diff output comparing the stash@{0} changes against the branch-name@{0} ref (i.e. b1@{0})

