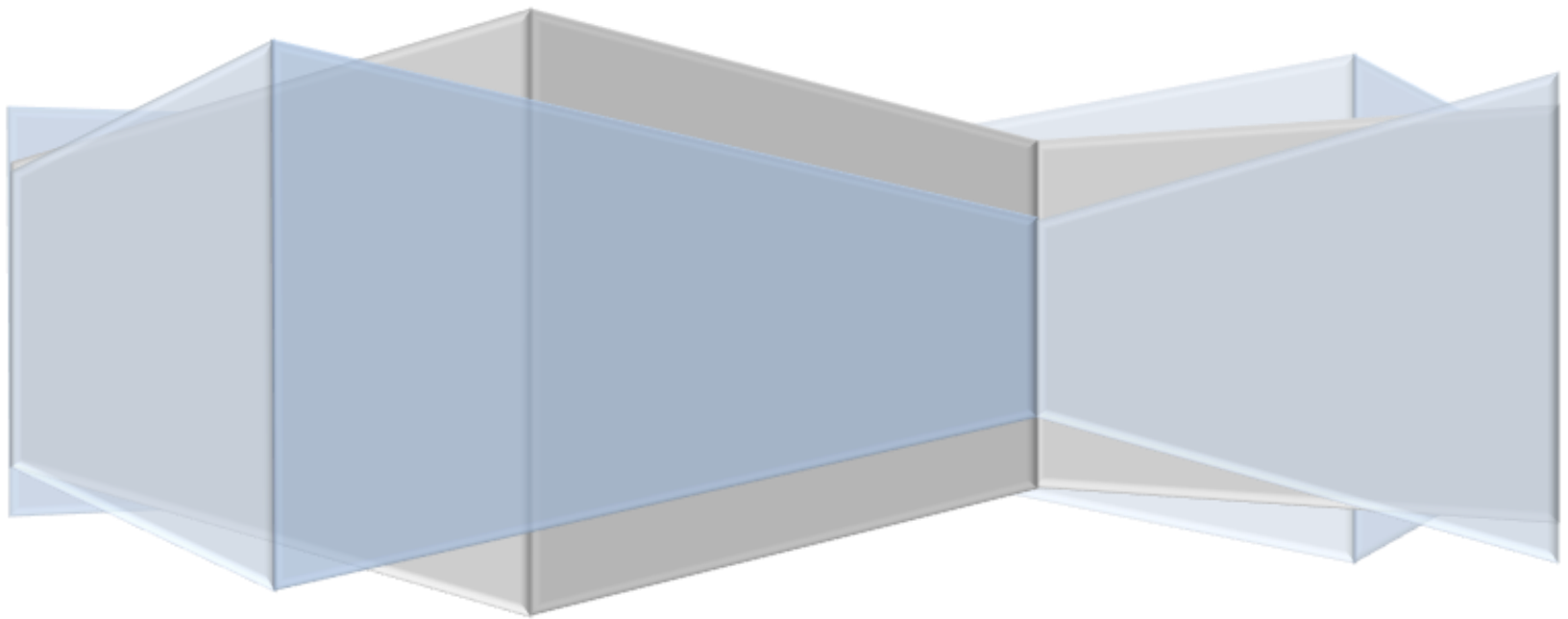


GIT Version Control System

Ganesh Palnitkar



GIT - Distributed Version Control System

Installation:

Windows: Install git by downloading latest version for windows installable from <https://git-scm.com/downloads> URL. On windows GIT is available in three interfaces as 'GIT Bash', 'GIT cmd' and 'GIT GUI'.

Linux: Install GIT on Linux by running 'yum' or 'apt' commands as below,

Debian / Ubuntu: `$ apt-get install git`

Centos 7: `$ yum install git -y`

Centos 8:

`$ sudo dnf install git -y`

`$ git --version`

GIT Commands:

1) Creating / initiating a fresh local repository,

`$ git init`

2) Copy / clone a remote repository to local

`$ git clone <remote_repo_url>`

`$ git clone --mirror` ... creates a fresh tracking folder with all remote ref and tags

`$ git clone --bare` ... create a new clone from remote with all tags but with no future fetch url associated.

3) Indexing / adding a file / set of files in git repo.

`$ git add .` ... to index all modified / new files to local repo

`$ git add <file name>` .. index a particular file

4) Commit a file / object to local repository

`$ git commit -m "suitable comment"` ... changes indexed are committed to the repository

5) Now to get a list of commands in git one can run the command,

`$ git help -a`

6) To find difference in the file changes in two versions.

`$ git diff HEAD` ... or,

`$ git diff rev1.0 rev1.1`

`$ git diff --cached` ... provides differences data between staging and repository area.

- 7) To update user name and email address for the user we can use,

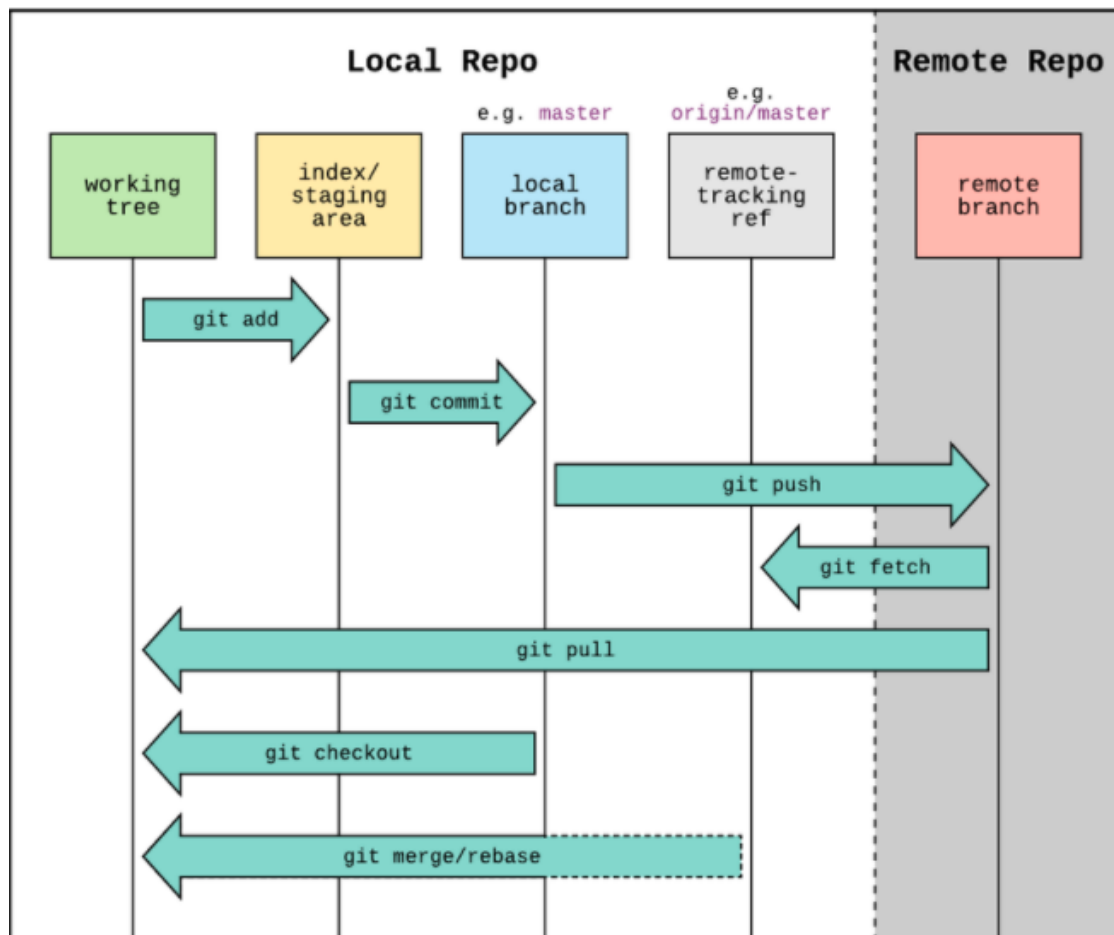
```
$ git config --local user.name "username"
$ git config --local user.email "email@address"
```

.. and if we want to apply this to all repositories on the machine.

```
$ git config --global user.email "email@address"
$ git config --global user.name "username"
```

To verify the effect of config command, check the file `.gitconfig` located at user home directory.. (`/home/<username>/.gitconfig`)

- 8) `$ git log -p` ... detailed pretty output
`$ git log` List of all commits, version inside git repo.
- 9) `$ git status` ... this commands gets information about the status of workspace



- 10) `$ git checkout <branch-name>` ... moves the HEAD pointer from branch to branch.
 or,
`$ git checkout <commit hash>` ... checks out a version to working area (detached HEAD)
- 11) `$ git rm --cached <filename>` ... removed a file from staging back into working area. Or,
`$ git restore --staged <filename>` ... to unstage file from staging area
- 12) Renaming a file in the repository can be done by using `$ git mv` command.
`$ git mv <src-file-name> <target-file-name>`
- 13) To get details of each commit- has details, we use the command, show command.
`$ git show <hash-object>`

```
$ git show HEAD~2
$ git show HEAD^^
```

Both these commands have the same effect. Helps to get details of a hash object code that is, 2 commits earlier from the current HEAD.

GIT Internals

At core GIT is a map. It is a table with key and value control.

Value is any sequence of bytes. It is converted into a hash code (a key) with SHA1 algorithm.

A 'sha1' code can be generated using below syntax on the git bash prompt,

```
$ echo "string" | git hash-object --stdin
```

Object Modelling.

Every object in GIT has its own SHA1 value.

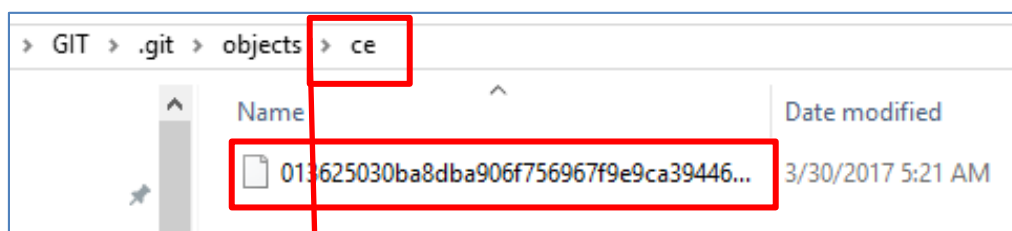
SHA1 values are unique in the Universe. There are very unlikely chances that there are two identical SHA1 code for different string value.

Example:

```
$ echo "hello" | git hash-object --stdin -w
```

..... This will write the sha1 value to repository by creating an object.

If we dig inside the `.git/objects` directory, we get to see an object as shown under,



'Objects' is the object database directory... the file starting with 013 is called the blob data file.

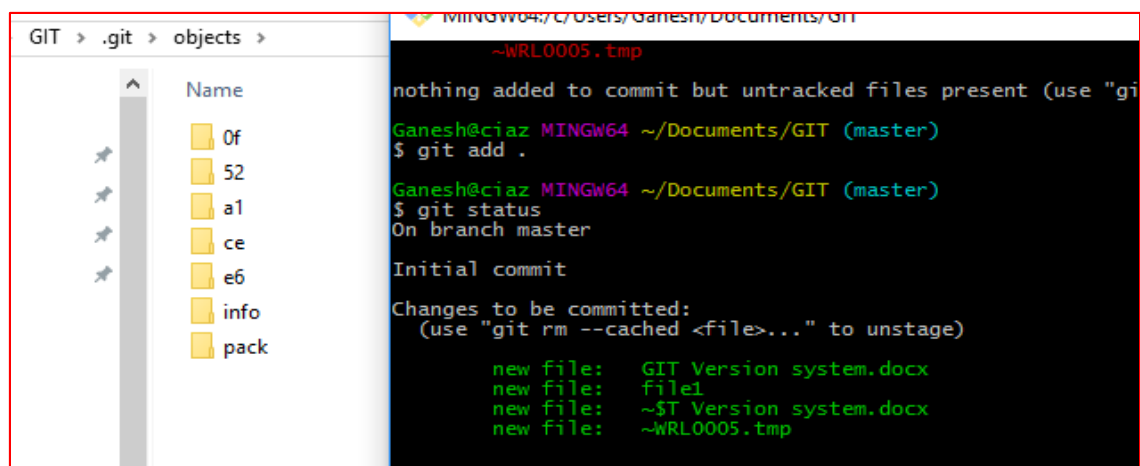
`$ git cat-file <ce013...> -t` Displays the file data type

```
Ganesh@ciaz MINGW64 ~/Documents/GIT/.git/objects/ce (GIT_DIR!)
$ git cat-file ce013625030ba8dba906f756967f9e9ca394464a -t
blob
```

`$ git cat-file <013...> -p ...` displays the file contents.

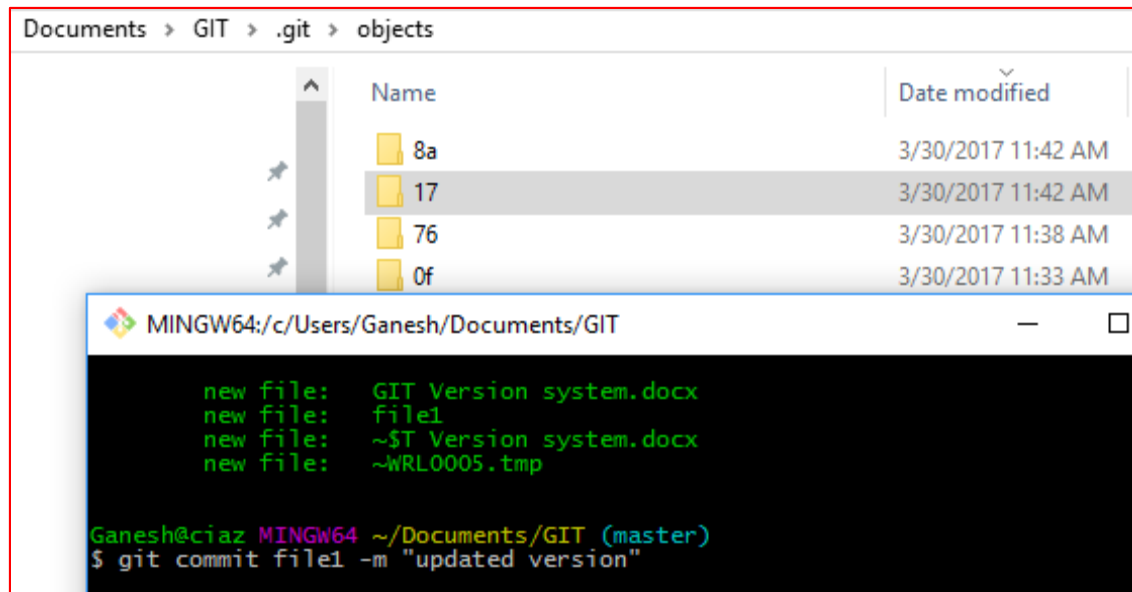
```
Ganesh@ciaz MINGW64 ~/Documents/GIT/.git/objects/ce (GIT_DIR!)
$ git cat-file ce013625030ba8dba906f756967f9e9ca394464a -p
hello
```

When a file is added to staging area, that's when the object related to each file is created in the .git directory. This is as shown in the below image.

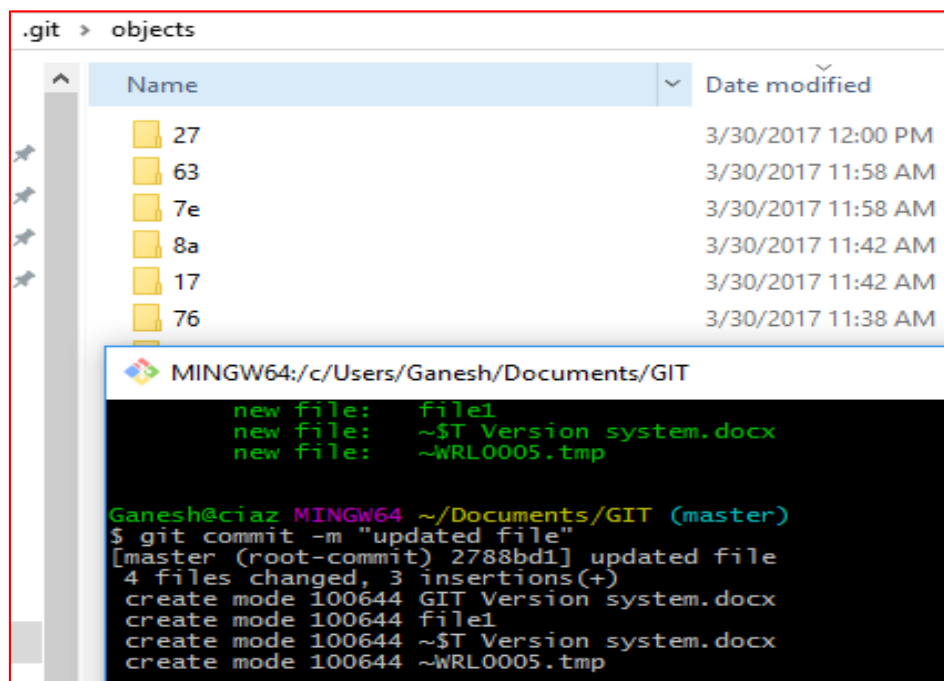


If a file is changed and committed to the repository, a new entry is added to the object folder for the new version of file. This means that that GIT creates a snapshot or a blob object with a 'SHA1 hash' for each version of file and preserves it.

To reinstate a file to it earlier version the SHA1 has can be referred.



For every action of update to the git repository, GIT creates a SHA1 (snapshot) for the file version.



If we dig inside the 27* directory and query the file sha1 code, we get below output.

```
$ git cat-file -p 2788bd15df880b0033d8fe35b354a741320359f6
tree 63065c607f25355d618c3b5bd52d6982d739d286
author ganesh palnitkar <ganeshhp@gmail.com> 1490855413 +0530
committer ganesh palnitkar <ganeshhp@gmail.com> 1490855413 +0530
updated file
```

The commit 'sha1' code includes information about the committer, author and also the tree information as to which blob this commit is related to.

If we try to get information about the tree sha1 code, we get details about all the commits that included in it. For example,

```
$ git cat-file -p 63065c607f25355d618c3b5bd52d6982d739d286
100644 blob 7e49db4b9291546ef0a91395c5e9593a37e805fa    GIT Version system.docx
100644 blob 17c2a175074b5d8694588f4cf2d276e644b2f948    file1
100644 blob a1b61c4c67393a8ba6a59c9987a2eb8b7a2902db    ~$T Version system.docx
100644 blob 524e38bb7cbfecf0d2ecd024a8f5af70bed173e2    ~WRL0005.tmp
```

./ commit (Tree)-> (blob) file1 → blob content. → (Tree) →

If we create files with same content the blob for these objects would be same. As shown in below image file1 and file2 has same contents thus the SHA1 is also the same. So GIT uses the information from the git database.

```
Ganesh@ciaz MINGW64 ~/Documents/GIT/.git/objects/bf (GIT_DIR!)
$ git cat-file -p efc4dde4a9b7799fda1709ee066f05c8da5affaa
100644 blob ed5d517812a7165128bfcfd2025be50874532174    GIT Version system.docx
100644 blob 17c2a175074b5d8694588f4cf2d276e644b2f948    file1
100644 blob 17c2a175074b5d8694588f4cf2d276e644b2f948    file2
100644 blob a1b61c4c67393a8ba6a59c9987a2eb8b7a2902db    ~$T Version system.docx
100644 blob 524e38bb7cbfecf0d2ecd024a8f5af70bed173e2    ~WRL0005.tmp
```

So a blob is not about the file but the contents of the file. The author and permission information about a file is stored in the **tree sha1** code of the file.

GIT stores all data in the form of SHA1 hash (snapshot) and not any more information about the file. This makes the GIT database very light weight. This is the total size of objects in the database.

GIT count-objects:

The count-objects command helps to get information on number of objects and files size.

```
$ git count-objects
14 objects, 236 kilobytes
```

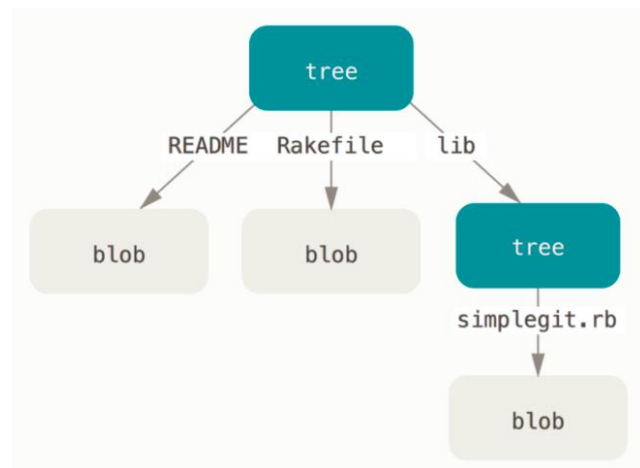
Tree Objects:

Git stores content in a manner similar to a UNIX file system, but a bit simplified. All the content is stored as tree and blob objects, with trees corresponding to UNIX directory entries and blobs corresponding more or less to inodes or file contents. A single tree object contains one or more entries, each of which is the SHA-1 hash of a blob or sub tree with its associated mode, type, and filename.

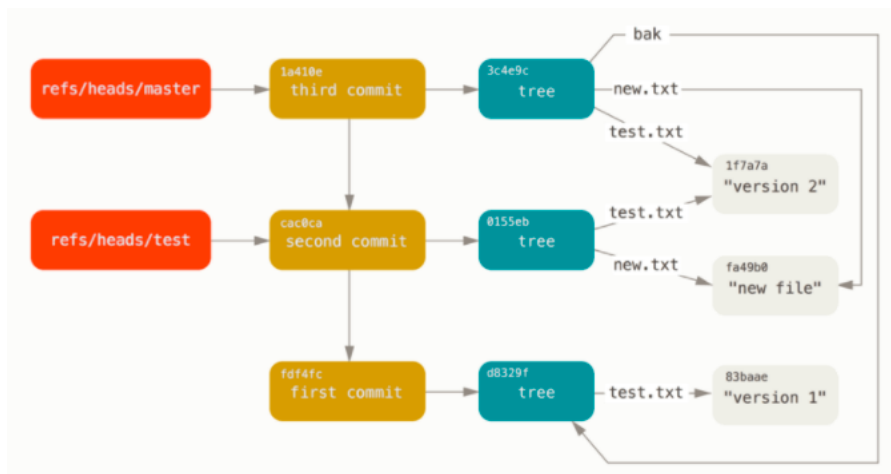
```
$ git cat-file -p master^{tree}
```

```
100644 blob 9ae7bca0fcb3390675e369adc223fabd87bf8b08 .classpath
100644 blob 3217d4e4f09fa055870ad8caba9c7964cd5fdb5f .gitignore
100644 blob 7cf6fe3490d539dd25415162fa02bb2eebfae78b .gitlab-ci.yml
040000 tree 14617dd1f0ad007e824fe5e8d9f0727d2915d5d6 .gradle
100644 blob 7c88c1529f78b5cb63f7a042f6b0addb8a7158dd .project
040000 tree ae6ded6253f738f48854d95f08e953bbbd1431fb .scannerwork
040000 tree ff36a41064904ec22efad4f5fc6bc400ccad978a .settings
100644 blob e4a2a1a32fbe7171631bf745098a927cafa037f1 Dockerfile
100644 blob 854c284b50855c4c3f87ea7fb1b70fc1e7eb13ef Helloworldwebapp-dev.war
100644 blob 05cf3273053a9e25ce24176297a40ce8fa0fa057 Jenkinsfile
100644 blob a1af257fb3e8f0ef5df00851280d63180c41a943 Jenkinsfile.declarative
100644 blob f8d14b406f1eb24171799b35e00ca730859fd495 README.md
100644 blob c83063ffbb2d2b34befdc89d6982393f07b8f2f0 pom.xml
040000 tree 280d3d1517cef9c31c1acf5e851e3df7f6f76641 src
040000 tree 2da8cc1ecb9a2e370edec68351e657a61396c6e3 target
```

The `master^{tree}` syntax specifies the tree object that is pointed to by the last commit on your `master` branch. Notice that the `src`, `target` subdirectory isn't a blob but a pointer to another tree:



If you follow all the internal pointers, you get an object graph something like this:



GIT directory objects with branch head references included.

GIT Branches:

Branch is a reference or a pointer to a commit hash.

GIT keep all branch related data in the ref folder inside the .git folder. Inside the **refs** folder there are **heads** and **tags** folder.

```
Lenovo@DESKTOP-RO77FPU MINGW64 ~/helloworldweb/.git (GIT_DIR!)
$ ll
total 23
-rw-r--r-- 1 Lenovo 197121 13 Apr 22 2021 COMMIT_EDITMSG
-rw-r--r-- 1 Lenovo 197121 103 Apr 22 2021 FETCH_HEAD
-rw-r--r-- 1 Lenovo 197121 23 Apr 21 2021 HEAD
-rw-r--r-- 1 Lenovo 197121 41 Apr 22 2021 ORIG_HEAD
-rw-r--r-- 1 Lenovo 197121 425 Apr 22 2021 config
-rw-r--r-- 1 Lenovo 197121 73 Apr 21 2021 description
drwxr-xr-x 1 Lenovo 197121 0 Apr 21 2021 hooks/
-rw-r--r-- 1 Lenovo 197121 2698 Apr 22 2021 index
drwxr-xr-x 1 Lenovo 197121 0 Apr 21 2021 info/
drwxr-xr-x 1 Lenovo 197121 0 Apr 21 2021 logs/
drwxr-xr-x 1 Lenovo 197121 0 Apr 22 2021 objects/
-rw-r--r-- 1 Lenovo 197121 390 Apr 21 2021 packed-refs
drwxr-xr-x 1 Lenovo 197121 0 Apr 21 2021 refs/
```

Current active branch reference.

Refs folder contains branch **head information** and branch **remote repository** reference.

```
refs/
|-- heads
|   |-- feature10
|   |-- master
|-- remotes
|   |-- github
|   |   |-- dev
|   |   |-- development
|   |   |-- feature1
|   |   |-- master
|   |   |-- staging
|   |-- origin
|   |   |-- HEAD
|   |   |-- master
|-- tags
```

Let's move into `Refs` folder.

```
Ganesh@ciaz MINGW64 ~/Documents/GIT/.git (GIT_DIR!)
$ cd refs/

Ganesh@ciaz MINGW64 ~/Documents/GIT/.git/refs (GIT_DIR!)
$ ls
heads/  tags/

Ganesh@ciaz MINGW64 ~/Documents/GIT/.git/refs (GIT_DIR!)
$ cd heads/

Ganesh@ciaz MINGW64 ~/Documents/GIT/.git/refs/heads (GIT_DIR!)
$ ls
master

Ganesh@ciaz MINGW64 ~/Documents/GIT/.git/refs/heads (GIT_DIR!)
$ cat master
bfaed43e3dfca4d1ab0dea30212ac61c9147b7a8
```

The SHA1 code is actually for the commit. If we try to get information about the SHA1, we see that the master branch points to a tree which in turn points to multiple blobs referring to files.

```
Ganesh@ciaz MINGW64 ~/Documents/GIT/.git/refs/heads (GIT_DIR!)
$ git cat-file -p bfaed43e3dfca4d1ab0dea30212ac61c9147b7a8
tree efc4dde4a9b7799fda1709ee066f05c8da5affaa
parent 2788bd15df880b0033d8fe35b354a741320359f6
author ganesh palnitkar <ganeshhp@gmail.com> 1490875501 +0530
committer ganesh palnitkar <ganeshhp@gmail.com> 1490875501 +0530

new file

Ganesh@ciaz MINGW64 ~/Documents/GIT/.git/refs/heads (GIT_DIR!)
$ git cat-file -p efc4dde4a9b7799fda1709ee066f05c8da5affaa
100644 blob ed5d517812a7165128bcfdc2025be50874532174    GIT Version system.docx
100644 blob 17c2a175074b5d8694588f4cf2d276e644b2f948    file1
100644 blob 17c2a175074b5d8694588f4cf2d276e644b2f948    file2
100644 blob a1b61c4c67393a8ba6a59c9987a2eb8b7a2902db    ~$T Version system.docx
100644 blob 524e38bb7cbfecf0d2ecd024a8f5af70bed173e2    ~WRL0005.tmp
```

If we create a new branch 'dev' and then run the git branch command, it shows that we are currently on the 'master' branch. GIT has an entry for the current branch in the HEAD file.

```
Ganesh@ciaz MINGW64 ~/Documents/GIT (master)
$ git branch
dev
* master

Ganesh@ciaz MINGW64 ~/Documents/GIT (master)
$ cat .git/HEAD
ref: refs/heads/master
```

If we switch the branch the ref in the HEAD file changes as shown below.

```
Ganesh@ciaz MINGW64 ~/Documents/GIT (master)
$ git checkout dev
M      GIT Version system.docx
Switched to branch 'dev'

Ganesh@ciaz MINGW64 ~/Documents/GIT (dev)
$ cat .git/HEAD
ref: refs/heads/dev
```

Switching between branches can be done using command,

```
$ git switch <branch_name>
```

A checkout mean to GIT is to move head to new branch and update the workarea.

To restore the HEAD to an earlier status, we first need to know the exact SHA1 code of the commit to which we want to reset the HEAD to. This can be done by running the `git reset` command.

A branch can also be renamed. While you are in a branch `dev`, we want to rename the `dev` branch to `development`, then we use below command.

```
$ git branch -m development
```

Branch Merge Operations:

When we want to merge the feature / dev branch to the master branch, we do this by first switching the branch to master, using command,

```
$ git switch master ... this will switch the current branch to 'master'. Once in master branch we then can use the command,
```

```
$ git merge dev ... considering the branch 'dev' is to be merged with master. We might see merge conflict depending on what files are merged to master from dev. GIT will try to auto resolve the conflict and if auto conflict is not possible, GIT will open the file in default editor and prompt the user for making changes to the conflicting file. Once the file is edited and saved, the file has to be → staged and → committed to the repository. Thus making the merge action complete.
```

The commit hash created after `merge` operation will have two `parent` hash ref.

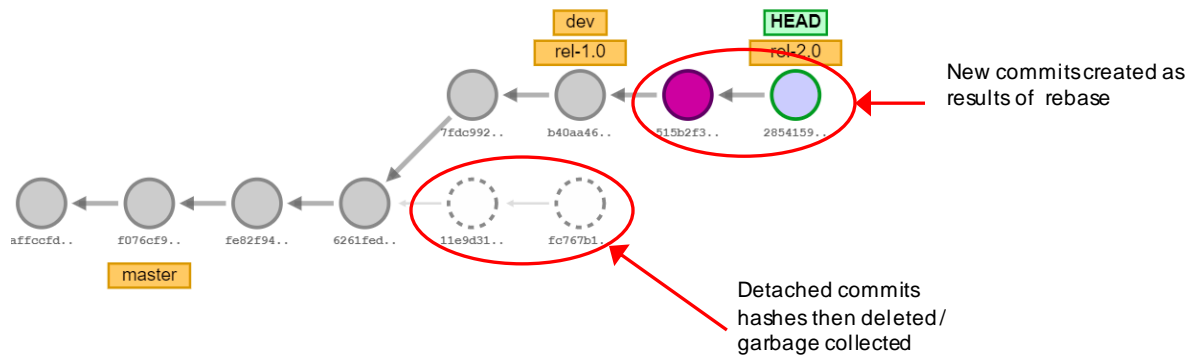
Check the '`git-Merge-Vs-Rebase.docx`' for details.

Related to working with branches, there are three rules followed by `git`.,

- 1) The current branch tracks new commits.

- 2) When we move to another branch using switch or checkout, Git updates the working directory contents.
- 3) Unreachable objects and garbage collected.

Branch Rebase operation: Reapply commits on top of another base tip. Avoid rebasing a commit (branch) if it is shared to a repository (remote).



GIT TAGS

Annotated tag. This is the one that comes with a comment. Here a tag that shows the comment and also metadata information about the tag and points to an object, in this case a commit object. So, a tag is a simple label in GIT.

```
$ git tag -a mytag -m "this file is important"
Ganesh@ciaz MINGW64 ~/Documents/GIT/.git/objects/bf (GIT_DIR!)
$ git tag
mytag
Ganesh@ciaz MINGW64 ~/Documents/GIT/.git/objects/bf (GIT_DIR!)
$ git cat-file -p mytag
object bfaed43e3dfca4d1ab0dea30212ac61c9147b7a8
type commit
tag mytag
tagger ganesh palnitkar <ganeshhp@gmail.com> 1490876903 +0530
this file is important
```

\$ git log This command lists out all history with commit statements, date and time of commit and owner of commit., etc. this is as shown below.

```
commit 77deca60c56a2f791003670c82cb08941654eecb
Author: ganesh palnitkar <ganeshhp@gmail.com>
Date: Thu Mar 30 18:39:15 2017 +0530

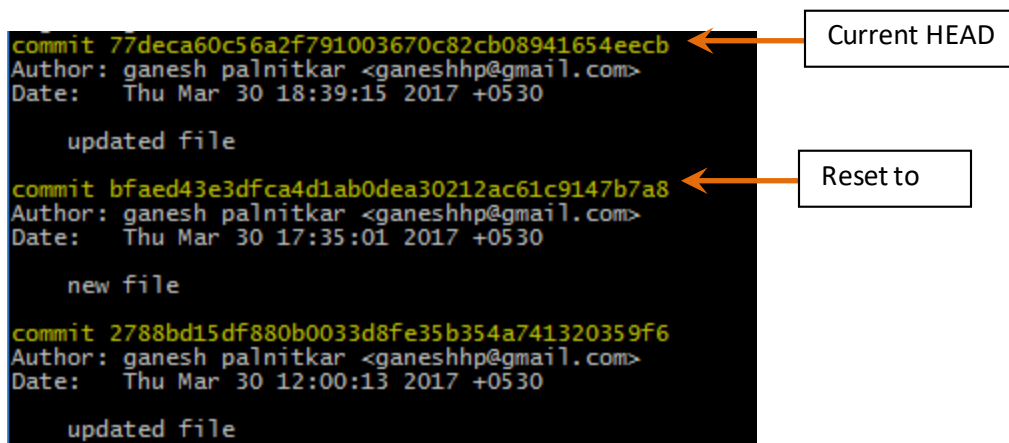
    updated file

commit bfaed43e3dfca4d1ab0dea30212ac61c9147b7a8
Author: ganesh palnitkar <ganeshhp@gmail.com>
Date: Thu Mar 30 17:35:01 2017 +0530

    new file

commit 2788bd15df880b0033d8fe35b354a741320359f6
Author: ganesh palnitkar <ganeshhp@gmail.com>
Date: Thu Mar 30 12:00:13 2017 +0530

    updated file
```



The diagram shows a terminal window with a Git commit history. The first commit (77deca60c56a2f791003670c82cb08941654eecb) is highlighted with a yellow background. An orange arrow points from a box labeled 'Current HEAD' to this commit. The second commit (bfaed43e3dfca4d1ab0dea30212ac61c9147b7a8) is also highlighted with a yellow background. An orange arrow points from a box labeled 'Reset to' to this commit. The third commit (2788bd15df880b0033d8fe35b354a741320359f6) is not highlighted. The commit messages are 'updated file', 'new file', and 'updated file' respectively.

Reset to a version in repository:

The log always shows the current HEAD first. Now if we want to reset the HEAD to the state as pointed by the arrow, we can use the command,

`$ git reset bfaed43` This will reset the HEAD to the commit status as shown. Care has to be taken while resetting the HEAD to earlier state as all changes after this commit would be lost.

Reset command has three options, Hard, Mixed and Soft.

`$ git reset --hard <hash-code>` ... file and hash object is removed from work directory and repo database.

`$ git reset --mixed <hash-code>` ... files are moved into work directory, but hash object is removed from repo database (garbage collected)

`$ git reset --soft <hash-code>` ... files are moved into staging and hash object for subsequent commits are removed (garbage collected).

Stash operations:

`$ git stash ...` this command gets applied to the files in work area. Once applied the files are moved to temporary storage area which git manages.

`$ git stash apply ...` the apply option with stash gets the files back into work area.

`$ git stash drop ...` the drop option deletes all files that area stashed.

`$ list` and `show` option are used to get details about the stashed operations performed.

`$ git stash list` ... this will list all stashes as `stash@{0}`, `stash@{1}`

`$ git stash show -p stash@{1}` ... this command shows details about a particular stash.

The latest stash created is stored in `refs/stash`. Older stashed are found in reflog of this reference and can be named using the usual reflog syntax (e.g. `stash @{0}` is the most recently created stash, `stash @{1}` is the one before it, `stash @{2.hours.ago}` is also possible).

Few more with Stash

```
$ git stash save "describe it"    # give the stash a name
$ git stash clear                 # delete a stashed commit
$ git stash save --keep-index     # stash only unstaged files
```

Resolving Conflict:

```
root@gitserver:~/project# git merge dev
Auto-merging src/main/webapp/index.jsp
CONFLICT (content): Merge conflict in src/main/webapp/index.jsp
Automatic merge failed; fix conflicts and then commit the result.
root@gitserver:~/project# nano src/main/webapp/index.jsp
root@gitserver:~/project# git add -A
root@gitserver:~/project# git commit -m "conflict resolved by updated .jsp file"
[master 034ae15] conflict resolved by updated .jsp file
root@gitserver:~/project# git log
commit 034ae15f51917c90b5a6926a8afcd51c1c9e1d1d
Merge: c1e7c85 5bbbd52
Author: ganesh <ganesh.palnitkar@yahoo.co.in>
Date: Mon Sep 4 19:13:05 2017 +0000

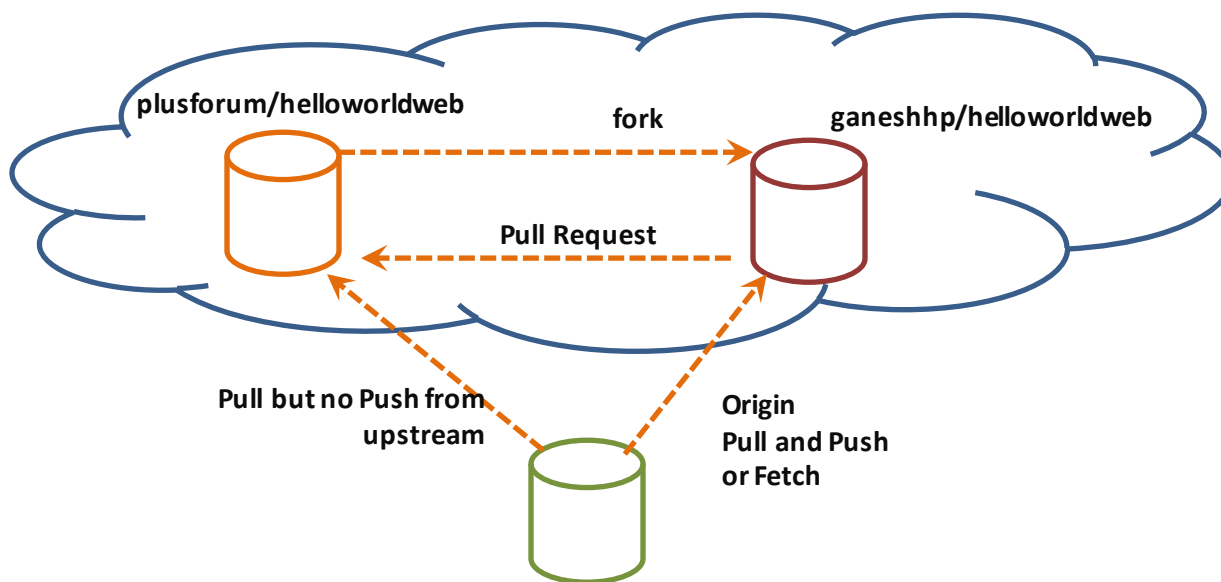
conflict resolved by updated .jsp file
```

Merge operation reporting conflict

Merge operation complete on committing the changes done with manually resolving conflicting text in the file

Open the conflicting file in editor to check and resolve conflict

Fork and Up-stream:



Some of the 'not-so-usual' GIT commands:

Split a file (Hunk) while adding to staging:

```
$ git add --patch <file-name>
```

--patch options allows the staging of selected file with selected part of the file only, we can also split the hunk into smaller parts so that we can decide to stage only part of the file (hunk).

```
Stage this hunk [y,n,q,a,d,e,?]? ?
y - stage this hunk
n - do not stage this hunk
q - quit; do not stage this hunk or any of the remaining ones
a - stage this hunk and all later hunks in the file
d - do not stage this hunk or any of the later hunks in the file
e - manually edit the current hunk
? - print help
```

Split a subfolder out into a new repository:

Sometimes you may want to turn a specific folder within your Git repo into a brand new repo. This can be done with `git filter-branch`:

Working with History:

`$ git commit --amend` ... this will help to make changes to the last commit by allowing to add additional object that are staged to it. In this operation what git does, is it copies the last commit objects and adds new objects to it and creates a new commit. And the last commit that we want to amend gets garbage collected.

```
$ git filter-branch --prune-empty --subdirectory-filter <folderName> master
```

```
# Filter the master branch to your directory and remove empty commits
Rewrite 48dc599c80e20527ed902928085e7861e6b3cbe6 (89/89)
Ref 'refs/heads/master' was rewritten
```

The repository now contains all the files that were in the specified subfolder. Although all of your previous files have been removed, they still exist within the Git history. You can now push your new local repo to the remote.

GIT cherry-pick

Cherry picking in git means to choose a commit from one branch and apply it onto another. If someone wants to commit specific commits in one branch to a target branch, then cherry-pick is used.

Checkout (switch to) target branch.


```
$ git cherry-pick <commit id>
```

Here commit id is activity id of another branch e.g.

```
$ git cherry-pick 4370c48c18aa560a3f8f716f16d11da94b8e31e6
```

```
$ git cherry-pick master
```

This applies the change introduced by commit at the tip of master and creates a new commit in target branch.

```
$ git cherry-pick master~4 master~2
```

Apply the changes introduced by the fifth and third last commits pointed to by master and create 2 new commits with these changes.

GIT Clean

```
$ git clean -fd ... forcefully remove untracked files and folders
```

```
$ git clean -nfd .. list files and folders that will be removed
```

Setting up remote repository as default for PUSH and PULL operations.

```
$ git branch --set-upstream-to myfork/master
```

GIT rev-list

```
$ git rev-list foo bar ^baz
```

means "list all the commits which are reachable from foo or bar, but not from baz".

A special notation "<commit1>..<commit2>" can be used as a short-hand for "^<commit1>'<commit2>". For example, either of the following may be used interchangeably:

```
$ git rev-list origin..HEAD
```

```
$ git rev-list HEAD ^origin
```

Another special notation is "<commit1>...<commit2>" which is useful for merges. The resulting set of commits is the symmetric difference between the two operands. The following two commands are equivalent:

```
$ git rev-list A B --not $(git merge-base --all A B)
```

```
$ git rev-list A...B
```