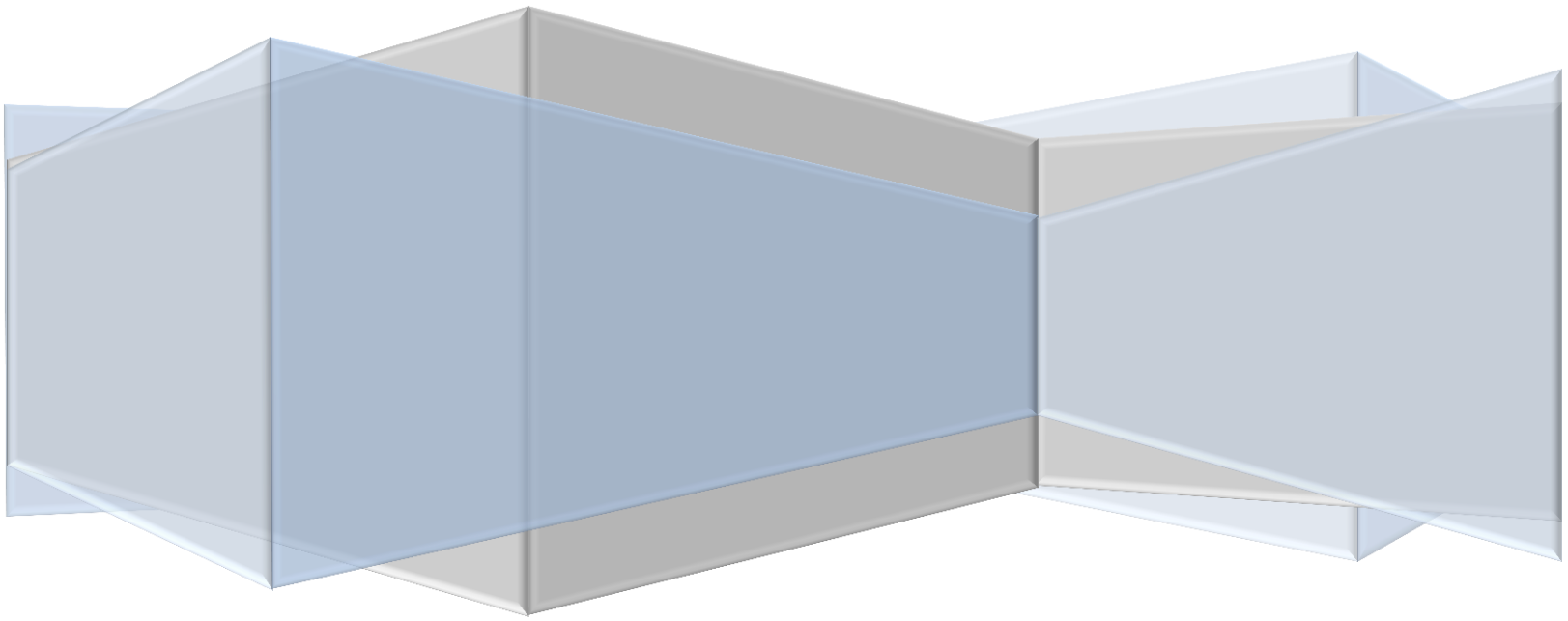
GIT Version Control System

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`

Centos 6:

```
$ yum install
http://opensource.wandisco.com/centos/6/git/x86_64/wandisco-
git-release-6-1.noarch.rpm

$ yum install git

$ 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>`

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`

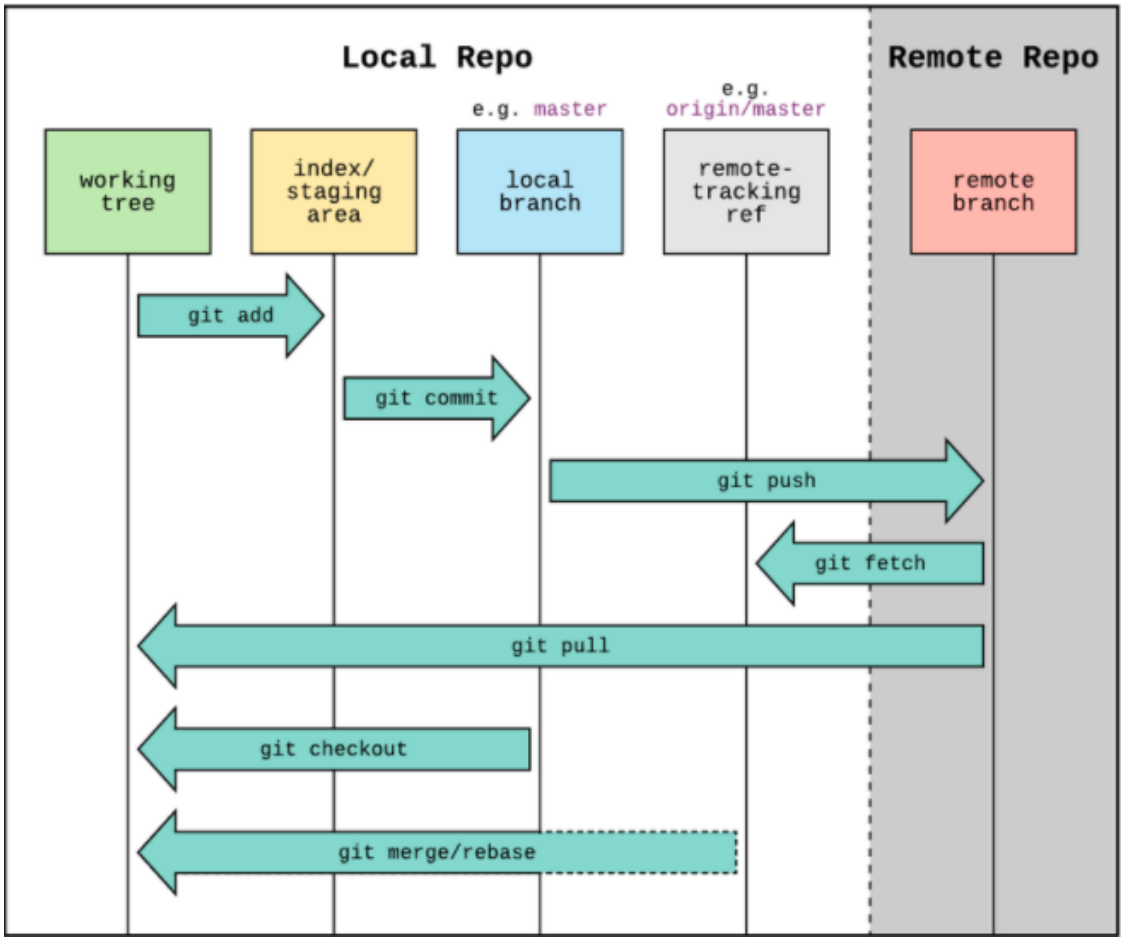
6) TO find difference in the file changes in two versions.

   `$ git diff HEAD` … or,
   `$ git diff rev1.0 rev1.1`

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"
```
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



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.

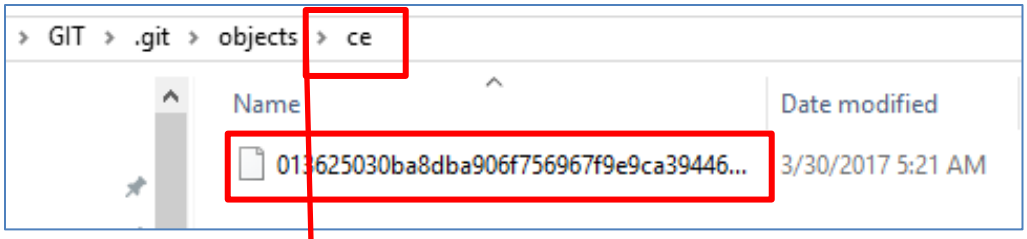
Every object in GIT has its own SHA1 value.

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.

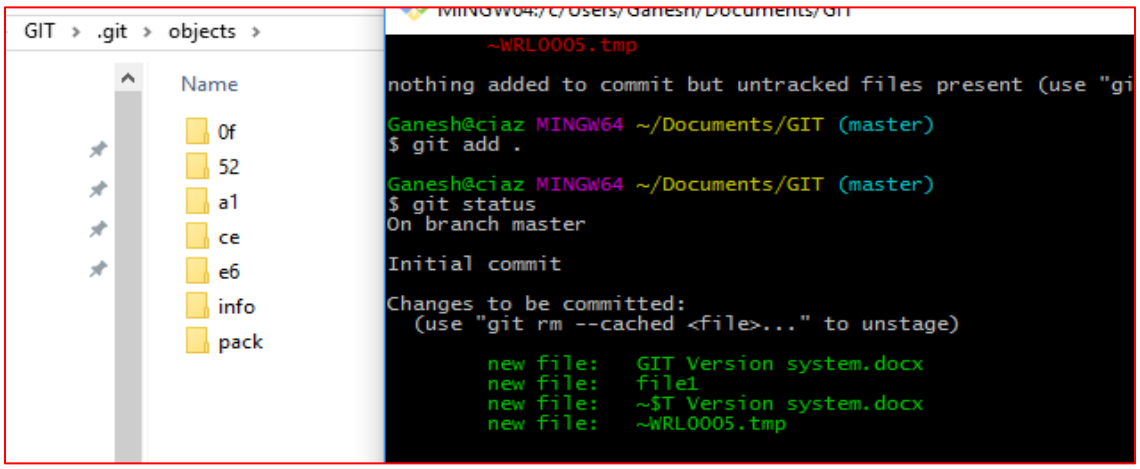
`$ git cat-file <ce013….> -t` …. Displays the file data type

```
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.

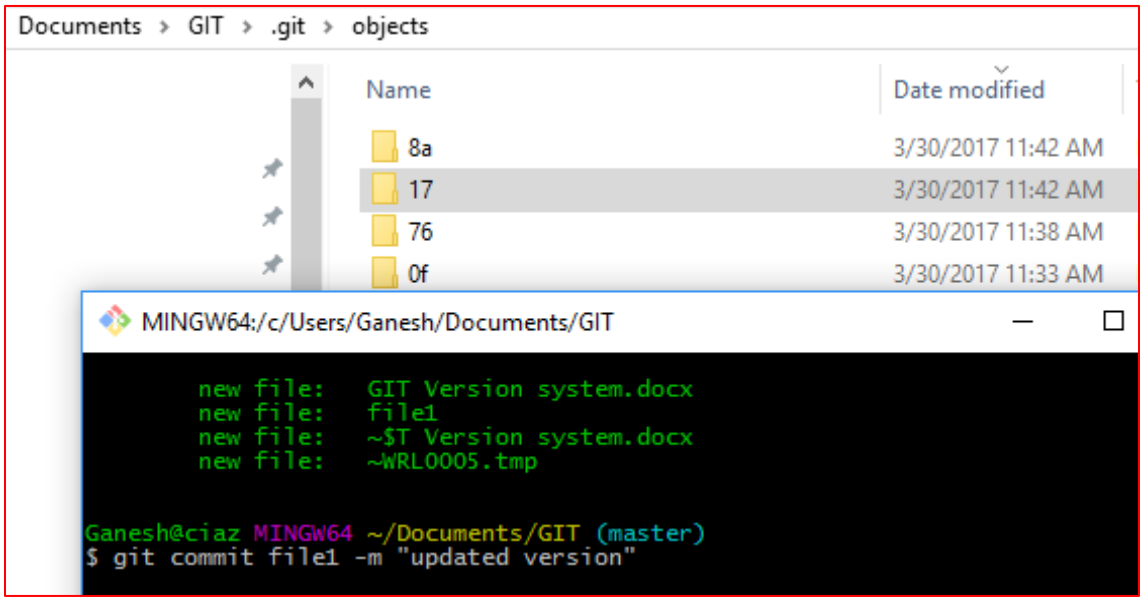
```
Ganesh@ciaz MINGW64 ~/Documents/GIT/.git/objects/ce (GIT_DIR!)
$ git cat-file ce013625030ba8dba906f756967f9e9ca394464a -p
hello
```

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.

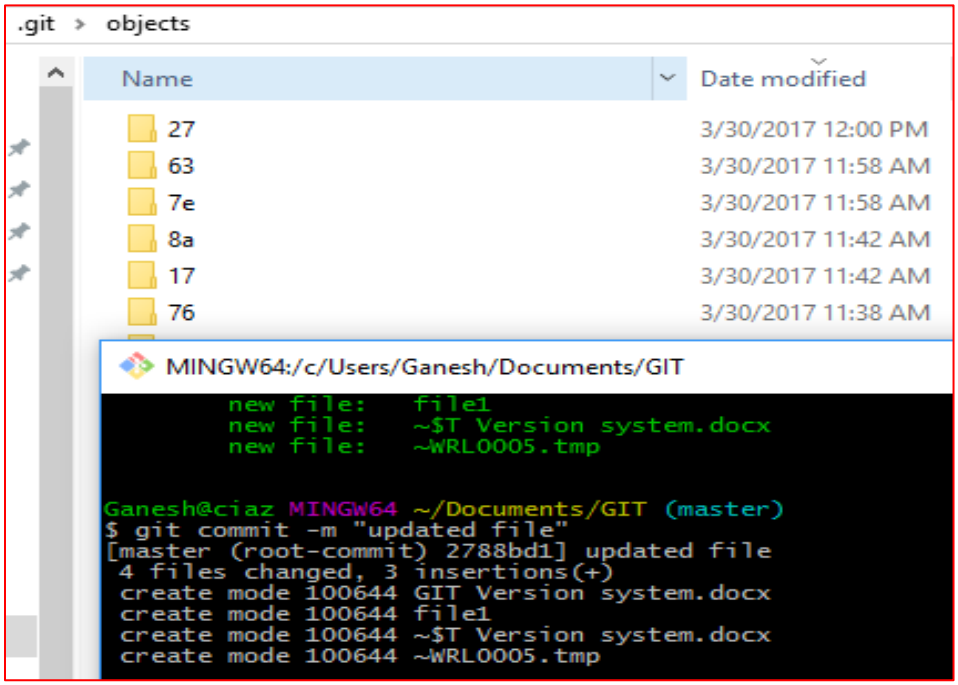


If a file is changed and committed to the repository, a new entry is added to the object folder for the new version of file. This means that that GIT creates a snapshot or a blob object with a ‘SHA1 hash’ for each version of file and preserves it.

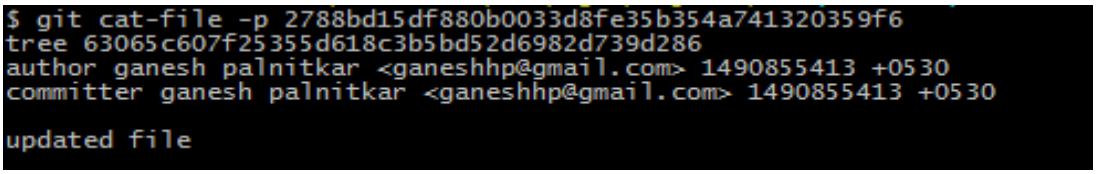
To reinstate a file to it earlier version the SHA1 has can be referred.



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

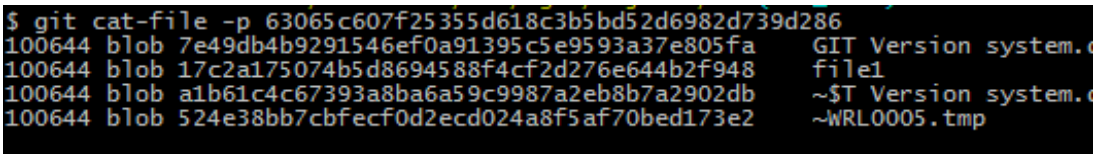


If we dig inside the 27* directory and query the file sha1 code, we get below output.



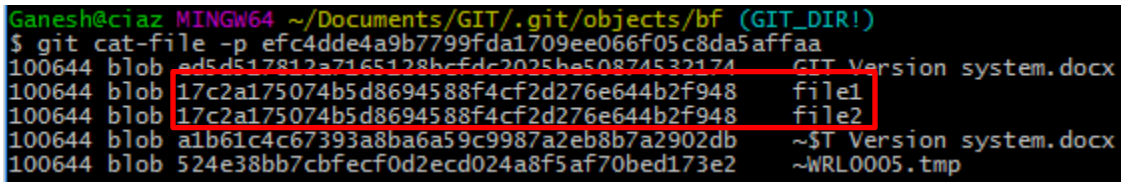
The commit 'sha1' code includes information about the committer, author and also the tree information as to which blob this commit is related to.

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



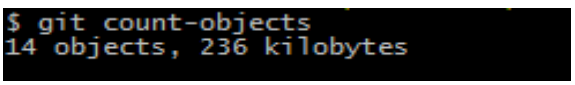
./ commit (Tree)-→ (blob) file1 → blob content. → (Tree) → …….

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.



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



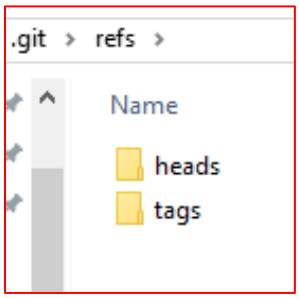
GIT TAGS

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 Branches:

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




```
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
```

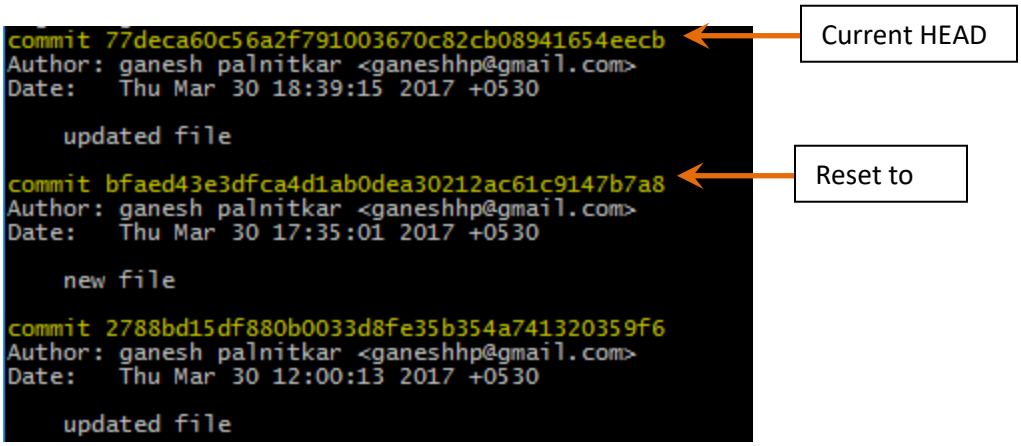
If we switch the branch the ref in the HEAD file changes as shown below.

```
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
```

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 command,

`$ 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.



Reset 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.

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 checkout 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 committed to the repository.** Thus making the merge action complete.

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} ….`

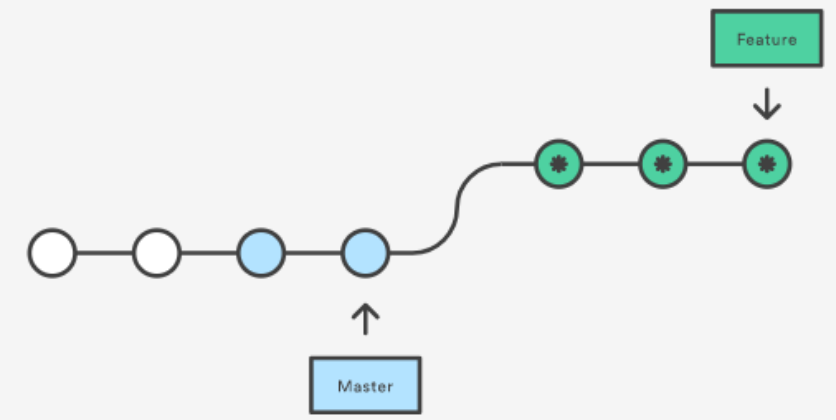
`$ git stash show -p stash@{1}` … this command shows details about a particular stash.

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
```

Rebase operation:

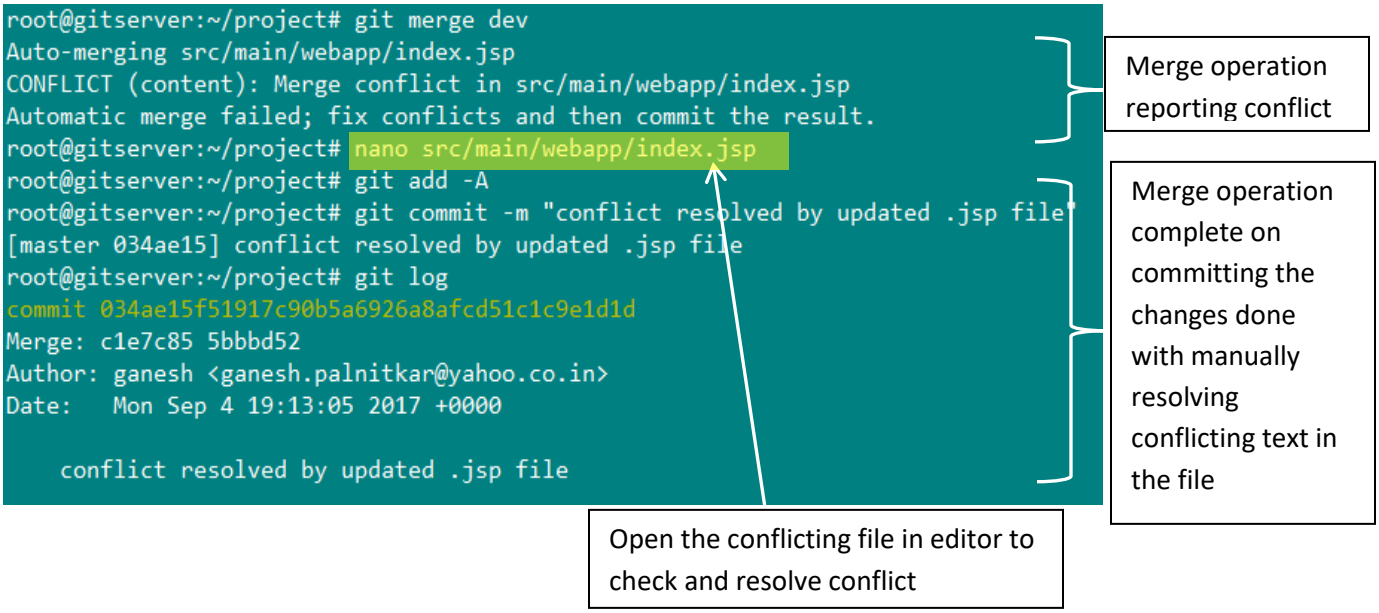


With rebase operation the Feature branch is entirely moved to the tip of the master branch.

```
$ git checkout dev
```

```
$ git rebase master
```

Resolving Conflict:



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

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`:

```
$ 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

Seating 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
```