**Creating a branch:**

Command: **git branch branchname**

**Moving to other branch:**

Command: **git checkout branchname**

**Merging new branch into master branch:**

1)first change the branch to master branch

2)then for example to merge the changes from developer1 branch.

run the command: git merge developer1

3)Then push the changes to master branch:

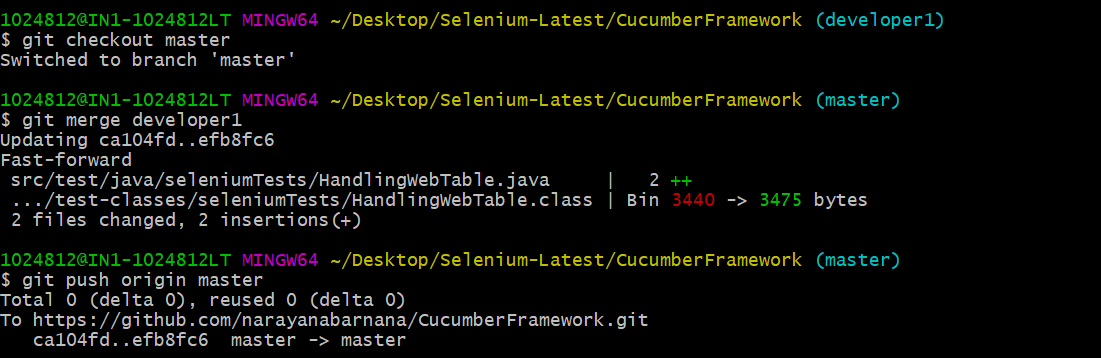
Command: Git push origin master

**Delete Branch from local:**

Command: git branch -d “branch name”

**Delete branch from remote:**

Command: git push origin -delete “branch name”



GIT Tags:

Tagging in GIT or any other VCS refers to creating specific points in history for your repository /data

This is usually done to mark release points.

Example: V1.0

Why should I create Tags:

1. To mark release points for your core data.
2. To create historic restore points.

When to create Tags:

1. When you want to create a release pont for a stable version of your code.
2. When you want to create a historic point for your code/data that you can refer at any future time(to restore your data).

How to create a tags:

Step 1: Checkout the branch where you want to create the tag git checkout "branch name"

example : git checkout master

Step 2: Create tag with some name git tag "tag name" example : git tag v1.0 git tag -a v1.0 -m "ver 1 of .." (to create annotated tags)

**Step 3: Display or Show tags**

git tag

git show v1.0

git tag -l “v1.\*”

**Step 4: Push tags to remote**

git push origin v1.0

git push origin --tags

git push --tags (to push all tags at once)

Step 5: Delete tags (if required only)

**to delete tags from local :** git tag -d v1.0

git tag --delete v1.0

**to delete tags from remote :**

git push origin -d v1.0

git push origin --delete v1.0

git push origin :v1.0

**to delete multiple tags at once:**

git tag -d v1.0 v1.1 (local)

git push origin -d v1.0 v1.1 (remote)

Checking out TAGS We cannot checkout tags in git We can create a branch from a tag and checkout the branch git checkout -b "branch name" "tag name" example : git checkout -b ReleaseVer1 v1.0

**Creating TAGS from past commits**

git tag "tag name" "reference of commit" example : git tag v1.2 5fcdb03

**What is a commit message?**

The command that is used to write a commit message is “git commit -a”.  
Now explain about -a flag by saying -a on the command line instructs git to commit the new content of all tracked files that have been modified. Also, mention you can use “git add <file>” before git commit -a if new files need to be committed for the first time

**How can you fix a broken commit?**

In order to fix any broken commit, use the command “git commit --amend”. When you run this command, you can fix the broken commit message in the editor.

**What is a repository in Git?**

Repository in Git is a place where Git stores all the files. Git can store the files either on the local repository or on the remote repository.

**How can you create a repository in Git?**

To create a repository, create a directory for the project if it does not exist, then run the command “git init”. By running this command .git directory will be created in the project directory.

**What is ‘bare repository’ in Git?**

A “bare” repository in Git contains information about the version control and no working files (no tree) and it doesn’t contain the special .git sub-directory. Instead, it contains all the contents of the .git sub-directory directly in the main directory itself, whereas the working directory consists of :

* A .git subdirectory with all the Git related revision history of your repository.
* A working tree, or checked out copies of your project files.

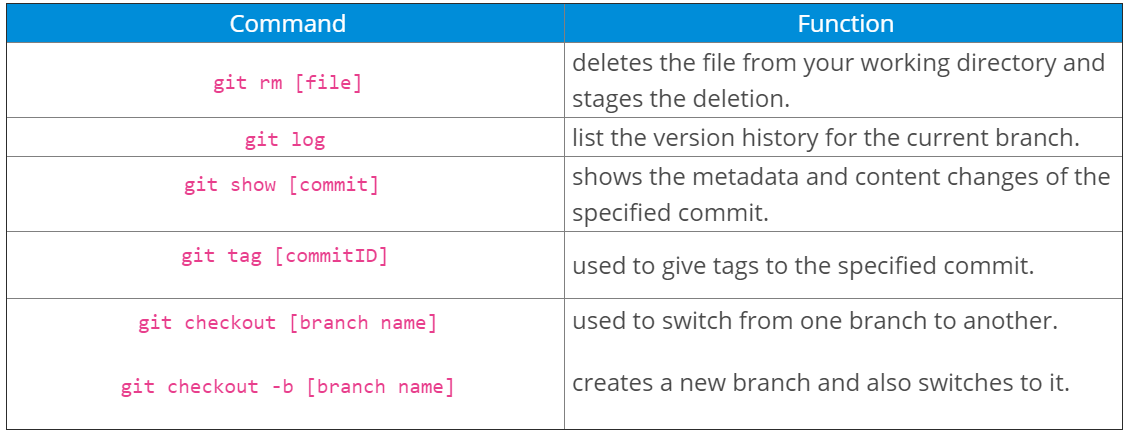
**What is a ‘conflict’ in git?**

Git can handle on its own most merges by using its automatic merging features. **There arises a conflict when two separate branches have made edits to the same line in a file, or when a file has been deleted in one branch but edited in the other**. Conflicts are most likely to happen when working in a team environment.

**What is git is-tree?**

‘git is-tree’ represents a tree object including the mode and the name of each item and the SHA-1 value of the blob or the tree.

**Few git commands and their usage**



**How to resolve a conflict in Git?**

Conflicts generally arise when two people have changed the same lines in a file, or if one developer deleted a file while another developer was modifying it. In these cases, Git cannot automatically determine what is correct. Conflicts only affect the developer conducting the merge, the rest of the team is unaware of the conflict. Git will mark the file as being conflicted and halt the merging process. It is then the developers' responsibility to resolve the conflict.

**Types of Merge conflicts**

A merge can enter a conflicted state at two separate points. When starting and during a merge process. The following is a discussion of how to address each of these conflict scenarios.

**1)Git fails to start the merge:**

A merge will fail to start when Git sees there are changes in either the working directory or staging area of the current project. Git fails to start the merge because these pending changes could be written over by the commits that are being merged in. When this happens, it is not because of conflicts with other developer's, but conflicts with pending local changes. The local state will need to be stabilized using git stash, git checkout, git commit or git reset. A merge failure on start will output the following error message:

error: Entry '<fileName>' not uptodate. Cannot merge. (Changes in working directory)

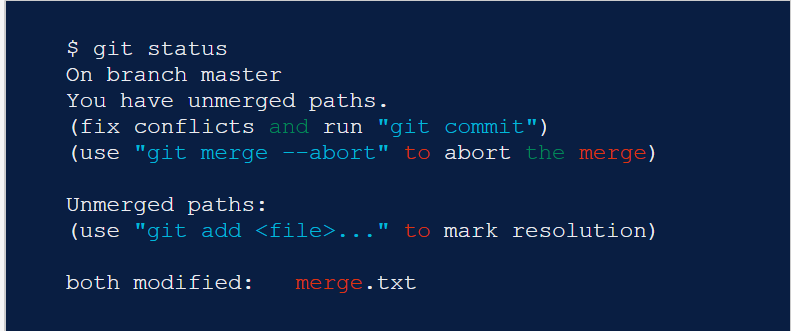
**2) Git fails during the merge**

A failure DURING a merge indicates a conflict between the current local branch and the branch being merged. This indicates a conflict with another developers code. Git will do its best to merge the files but will leave things for you to resolve manually in the conflicted files. A mid-merge failure will output the following error message:

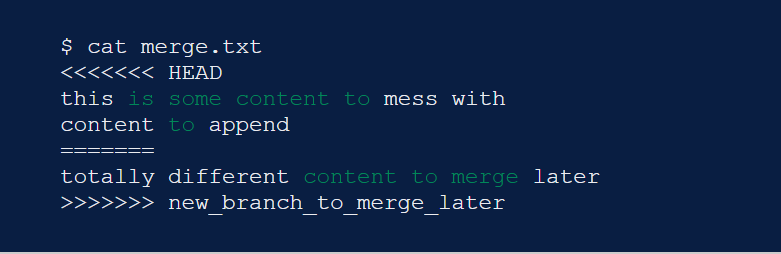
error: Entry '<fileName>' would be overwritten by merge. Cannot merge. (Changes in staging area)

**How to identity merge conflicts:**

As we have experienced from the proceeding example, Git will produce some descriptive output letting us know that a CONFLICT has occcured. We can gain further insight by running the [git status](https://www.atlassian.com/git/tutorials/inspecting-a-repository) command



The output from git status indicates that there are unmerged paths due to a conflict. The merge.text file now appears in a modified state. Let's examine the file and see whats modified.



Here we have used the cat command to put out the contents of the merge.txt file. We can see some strange new additions

<<<<<<< HEAD

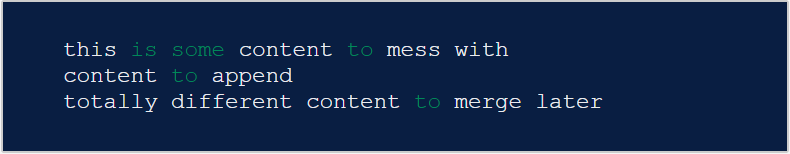
=======

>>>>>>> new\_branch\_to\_merge\_later

Think of these new lines as "conflict dividers". The ======= line is the "center" of the conflict. All the content between the center and the <<<<<<< HEAD line is content that exists in the current branch master which the HEAD ref is pointing to. Alternatively all content between the center and >>>>>>> new\_branch\_to\_merge\_later is content that is present in our merging branch.

**How to resolve merge conflicts using the command line**

The most direct way to resolve a merge conflict is to edit the conflicted file. Open the merge.txt file in your favorite editor. For our example lets simply remove all the conflict dividers. The modified merge.txt content should then look like:



Once the file has been edited use git add merge.txt to stage the new merged content. To finalize the merge create a new commit by executing:

git commit -m "merged and resolved the conflict in merge.txt"

Git will see that the conflict has been resolved and creates a new merge commit to finalize the merge.

**Git commands that can help resolve merge conflicts**

**General tools**

**git status**

The status command is in frequent use when a working with Git and during a merge it will help identify conflicted files.

**git log --merge**

Passing the --merge argument to the git log command will produce a log with a list of commits that conflict between the merging branches.

**git diff**

diff helps find differences between states of a repository/files. This is useful in predicting and preventing merge conflicts.

**Tools for when git fails to start a merge**

git checkout

checkout can be used for undoing changes to files, or for changing branches

git reset --mixed

reset can be used to undo changes to the working directory and staging area.

**Tools for when git conflicts arise during a merge**

git merge --abort

Executing git merge with the --abort option will exit from the merge process and return the branch to the state before the merge began.

git reset

Git reset can be used during a merge conflict to reset conflicted files to a know good state

The following steps will resolve conflict in Git-

* Identify the files that have caused the conflict.
* Make the necessary changes in the files so that conflict does not arise again.
* Add these files by the command git add.
* Finally to commit the changed file using the command git commit.

<https://www.atlassian.com/git/tutorials/using-branches/merge-conflicts>

**In Git how do you revert a commit that has already been pushed and made public?**

There can be two approaches to tackle this question and make sure that you include both because any of the below options can be used depending on the situation:

* Remove or fix the bad file in a new commit and then push it to the remote repository. This is the most obvious way to fix an error. Once you have made necessary changes to the file, then commit it to the remote repository using the command: git commit -m “commit message”
* Also, you can create a new commit that undoes all changes that were made in the bad commit. To do this use the command

git revert <name of bad commit>

**Git Merge Vs Rebase:**

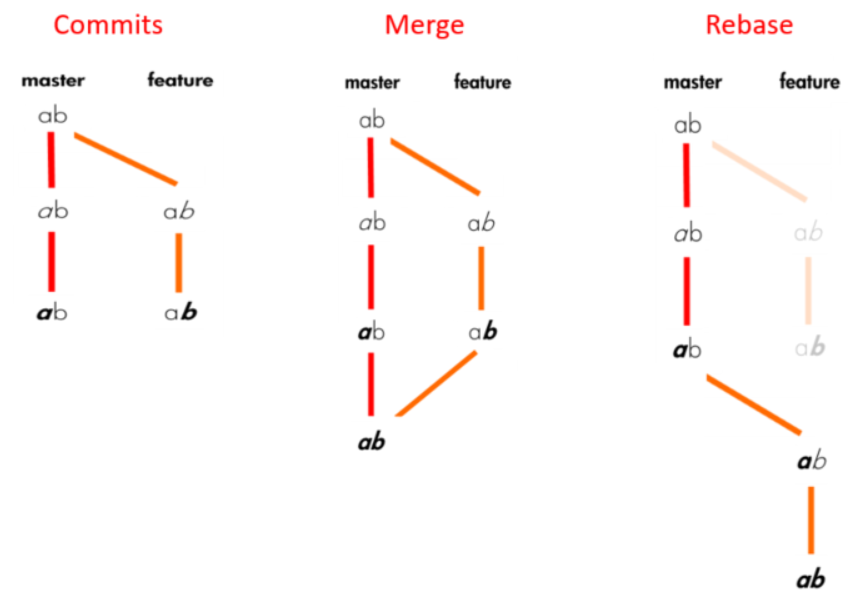
In Git there are 2 ways to integrate changes from one branch to another

* git merge
* git rebase

Rebasing and merging are both designed to integrate changes from one branch into another branch but in different ways.

For ex. let’s say we have commits like below, the merge will result as a combination of commits, whereas rebase will add all the changes in feature branch starting from the last commit of the master branch:

<https://medium.com/datadriveninvestor/git-rebase-vs-merge-cc5199edd77c>



* When you do rebase a feature branch onto master, you move the base of the feature branch to master branch’s ending point.
* Merging takes the contents of the feature branch and integrates it with the master branch. As a result, only the master branch is changed. The feature branch history remains same.
* Merging adds a new commit to your history.

**When to rebase? When to Merge?**

If the feature branch you are getting changes from is shared with other developers, rebasing is not recommended, because the rebasing process will create inconsistent repositories. For individuals, rebasing makes a lot of sense.

If you want to see the history completely same as it happened, you should use merge. **Merge preserves history whereas rebase rewrites it.**

Rebasing is better to streamline a complex history, you are able to change the commit history by [interactive rebase](https://medium.com/@filissen/git-interactive-rebase-e265bb55952a). You can remove undesired commits, squash two or more commits into one or edit the commit message.

Rebase will present conflicts one commit at a time whereas merge will present them all at once. It is better and much easier to handle the conflicts but you shouldn’t forget that reverting a rebase is much more difficult than reverting a merge if there are many conflicts.

<https://www.atlassian.com/git/tutorials/merging-vs-rebasing>

**Difference between GIT Pull and GIT Fetch**

GIT Pull: Git Pull command pulls new changes and commits from a particular branch from your central repository and updates your target branch in your local repository.

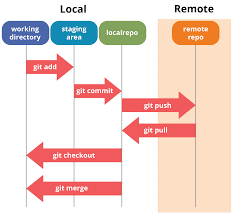
GIT Fetch: GIT Fetch is also used for the same purpose but it works in a slightly different way. When you perform a GIT Fetch, it pulls new commits from the desired branch and stores it in a new branch in your local repository. If you want to reflect these changes in your target branch, GIT Fetch must be follwed by GIT Megre. Your targeted branch will be updated after merging the target branch and fetched branch.

GIT Pull = GIT Fetch + GIT Merge.

**What is Staging area or Index in GIT?**

Before completing the commits, the code can be formatted and reviewed in an intermediate area known as Staging area or Index.

From the below diagram it is evident that every change is first verified in the staging area and then the change is committed to the repository.



**GIT Stash:**

GIT Stash temporarily shelves/stashes changes you have made to your working copy so you can work on something else, and then come back and reapply them later on.

Stashing is handy if you need to quickly switch context and work on something else,but you are mid-way through a code change and aren’t quite ready to commit.

**GIT Stash commands:**

1)git stash: 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:

Command: git stash

2)git stash pop: You can reapply previously stashed changes with git stash pop

Command: git stash pop

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:

Command: git stash apply

**Difference between git stash pop and git stash apply:**

git stash pop throws away the (topmost, by default) stash after applying it, whereas git stash apply leaves it in the stash list for possible later reuse (or you can then git stash drop it).

This happens unless there are conflicts after git stash pop, in which case it will not remove the stash, leaving it to behave exactly like git stash apply.

Another way to look at it: git stash pop is git stash apply && git stash drop

**Difference between git diff and status**

Git diff shows the difference between various commits and also between the working directory and index.

Git status shows the difference between working directory and index.

**Difference between git remote and git clone:**

**Git remote add** cretes an entry in your git config that specifies the name of a particular URL.

**Git clone** creates a new git repository by copying an existing one located at the URL.

**What is the use of GIT Stash drop?**

Git stash drop is used to remove the stashed item. It will remove the last added stash item by default, and it can also remove a specific item if you include it as an argument.

Without argument: git stash drop

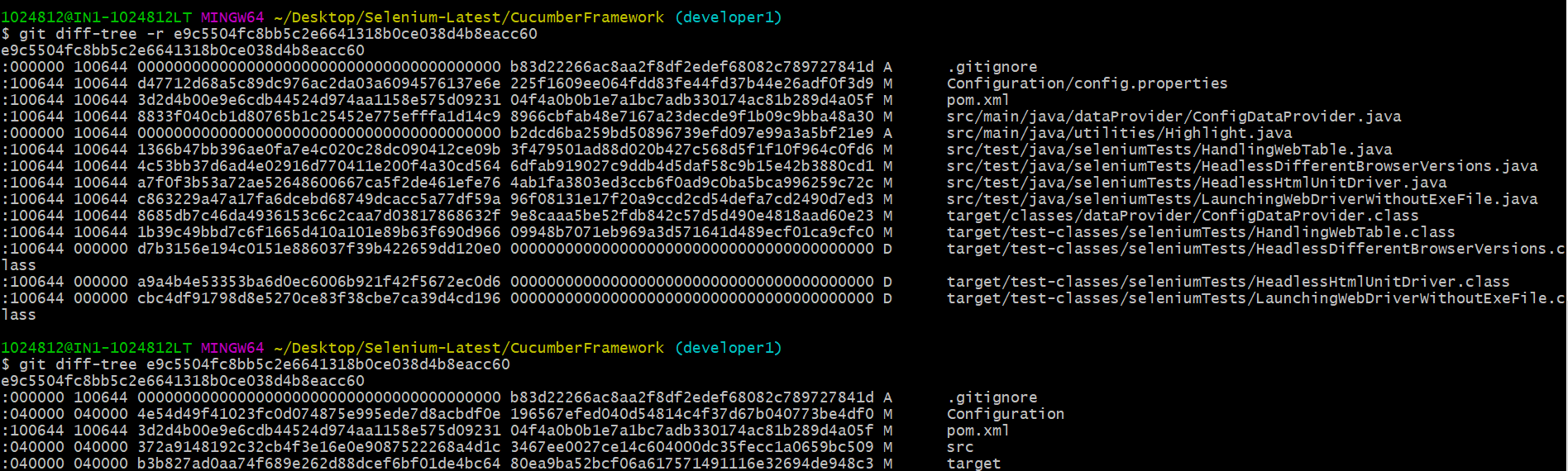
With argument: git stash drop stash@{0}

**How do you find a list of files that you have changed in a particular commit?**

**1**) git diff-tree -r {hash}

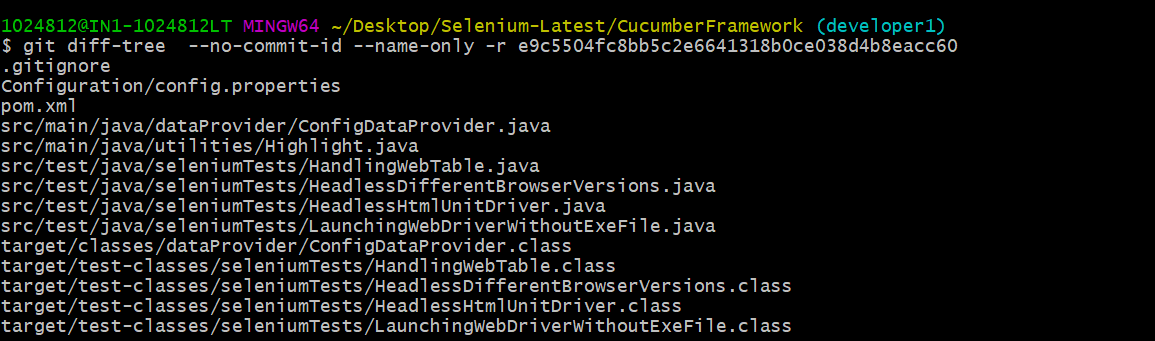
Eg: git diff-tree -r e9c5504fc8bb5c2e6641318b0ce038d4b8eacc60

Hash means commit id. By giving the commit id, this will list all the files that were changed or added in that commit. The -r flag makes the list of individual files, rather than collapsing them into root directory names only. i.e. if we don’t use -r flag it will display only the folder names.

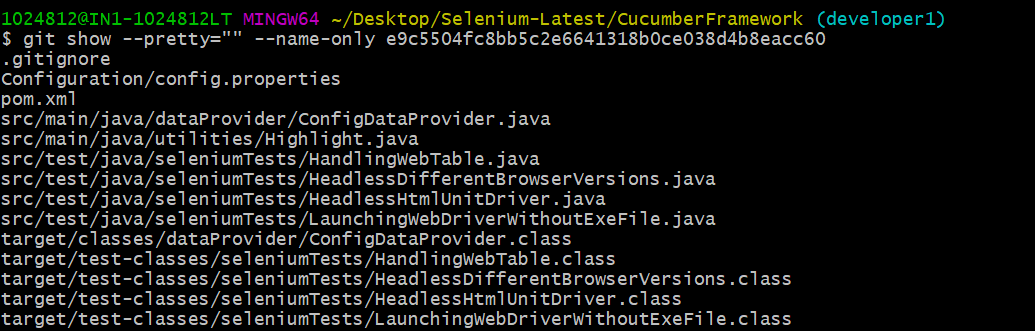


**2) git diff-tree --no-commit-id --name-only -r e9c5504fc8bb5c2e6641318b0ce038d4b8eacc60**

Here no-commit-id will suppress the commit hashes from appearing in the output and –name-only will only print the file names, instead of their paths.

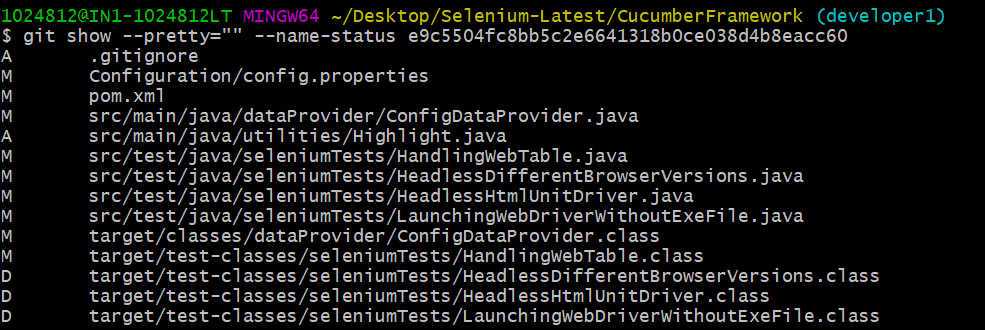


**3)git show –pretty=”” –name-only commitid**



**4)git show –pretty=”” –name-status commitid**

**This will display the status of the file like added or modified or deleted.**



**What is the function of git config?**

Git uses your username to associate commits with an identity. The git config command can be used to change your Git configuration, including your username.

Example:Suppose you want to give a username and email id to associate a commit with an identity so that you can know who has made a particular commit. For that I will use:

git config –global user.name “Your Name”: This command will add a username.  
git config –global user.email “Your E-mail Address”: This command will add an email id.

**What does a commit object contain?**

Commit object contains the following components, you should mention all the three points presented below:

* A set of files, representing the state of a project at a given point of time
* Reference to parent commit objects
* An SHA-1 name, a 40 character string that uniquely identifies the commit object

**Describe the Branching strategies you have used?**

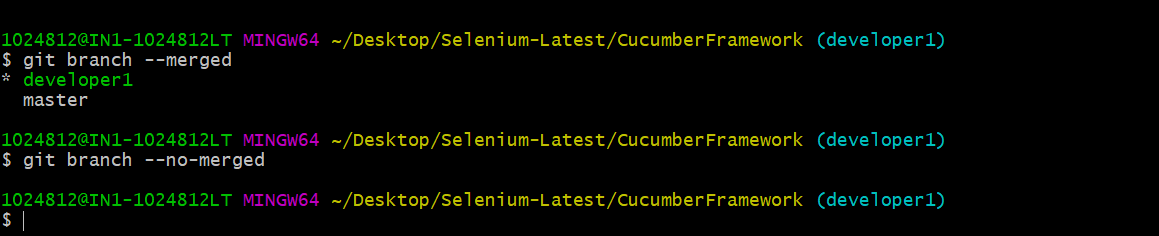
* **Feature branching** – A feature branch model keeps all of the changes for a particular feature inside of a branch. When the feature is fully tested and validated by automated tests, the branch is then merged into master.
* **Task branching** – In this model, each task is implemented on its own branch with the task key included in the branch name. It is easy to see which code implements which task, just look for the task key in the branch name.
* **Release branching** – Once the develop branch has acquired enough features for a release, you can clone that branch to form a Release branch. Creating this branch starts the next release cycle, so no new features can be added after this point, only bug fixes, documentation generation, and other release-oriented tasks should go in this branch. Once it is ready to ship, the release gets merged into master and tagged with a version number. In addition, it should be merged back into the develop branch, which may have progressed since the release was initiated.
* In the end tell them that branching strategies vary from one organization to another so I know basic branching operations like delete, merge, checking out a branch, etc.

**How will you know in GIT if a branch has already been merged into master?**

Use the below commands:

git branch –merged – It lists the branches that have been merged into the current branch.

Git branch –no-merged – it lists the branches that have not been merged.



**How to view all commits across all branches ?**

When you execute git log command it will show all the commits related to the current branch only.

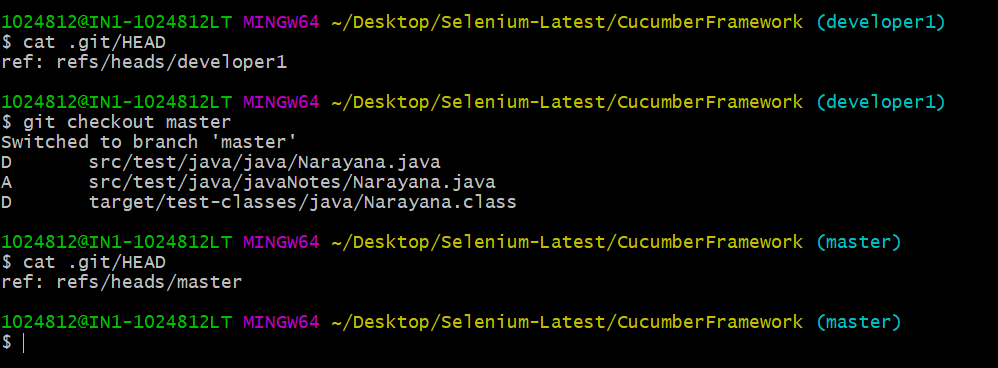
Using the command: git log –branches=\*



**What is HEAD in GIT:**

Head is a reference to the last commit in the currently checkout branch.

You can think of the HEAD as the “current branch”. When you switch branch with checkout, the HEAD revision changes to the point to the tip of the new branch.



It is possible for HEAD to refer to a specific version that is not associated with a branch name. This situation is called “Detached HEAD”

**Undoing Changes in GIT:**

When 'undoing' in Git, you are usually moving back in time, or to another timeline where mistakes didn't happen.

Method 1: By using git checkout

When you have found a commit reference to the point in history you want to visit,

Step 1: get the commit you want to do the undo. Command: git log –oneline

**Tell me the difference between HEAD, working tree and index, in Git.**

* The working tree/working directory/workspace is the directory tree of (source) files that you are able to see and edit.
* The index/staging area is a single, large, binary file in <baseOfRepo>/.git/index, which lists all files in the current branch, their SHA-1 checksums, timestamps, and the file name – it is not another directory which contains a copy of files in it.
* HEAD is used to refer to the last commit in the currently checked-out branch.

**What is Git fork? What is the difference between fork, branch, and clone?**

* A fork is a copy of a repository. Normally you fork a repository so that you are able to freely experiment with changes without affecting the original project. Most commonly, forks are used to either propose changes to someone else’s project or to use someone else’s project as a starting point for your own idea.
* git cloning means pointing to an existing repository and make a copy of that repository in a new directory, at some other location. The original repository can be located on the local file system or on remote machine accessible supported protocols. The git clone command is used to create a copy of an existing Git repository.
* In very simple words, git branches are individual projects within a git repository. Different branches within a repository can have completely different files and folders, or it could have everything the same except for some lines of code in a file.

**What are the different ways you can refer to a commit?**

* In Git each commit has a unique hash. These hashes are used to identify the corresponding commits in various scenarios, for example, while trying to checkout a particular state of the code using the git checkout {hash} command.
* Along with this, Git maintains a number of aliases to certain commits, known as refs. Also, every tag that is created in the repository effectively becomes a ref and that is exactly why you can use tags instead of committing hashes in various git commands. Git also maintains a number of special aliases that are changed based on the state of the repository, such as HEAD, FETCH\_HEAD, MERGE\_HEAD, etc.
* In Git, commits are allowed to be referred to as relative to one another. In the case of merge commits, where the commit has two parents, ^ can be used to select one of the two parents, for example, HEAD^2 can be used to follow the second parent.
* And finally, refspecs are used to map local and remote branches together. However, these can also be used to refer to commits that reside on remote branches allowing one to control and manipulate them from a local git environment.

**Explain the difference between reverting and resetting.**

* Git reset is a powerful command that is used to undo local changes to the state of a Git repository. Git reset operates on “The Three Trees of Git” which are, Commit History ( HEAD ), the Staging Index, and the Working Directory.
* Revert command in Git creates a new commit that undoes the changes from the previous commit. This command adds a new history to the project. It does not modify the existing history.

**What is git reflog?**

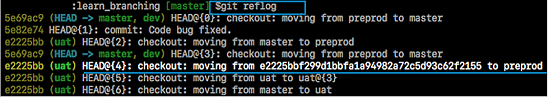
* The ‘reflog’ command keeps a **track of** **every single change made in the references**(branches or tags) of a repository and keeps a log history of the branches and tags that were either created locally or checked out. Reference logs such as the commit snapshot of when the branch was created or cloned, checked-out, renamed, or any commits made on the branch are maintained by [Git](https://www.edureka.co/blog/what-is-git/) and listed by the ‘reflog’ command.
* *Note:  The branch will be recoverable from your working directory only if the branch ever existed in your local repository i.e. the branch was either created locally or checked-out from a remote repository in your local repository for Git to store its reference history logs.*
* This command must be executed in the repository that had the lost branch. If you consider the remote repository situation, then you have to execute the reflog command on the developer’s machine who had the branch.

**command: git reflog**

**How to recover a deleted branch using git reflog?**

**Step 1**: **History logs of all the references**

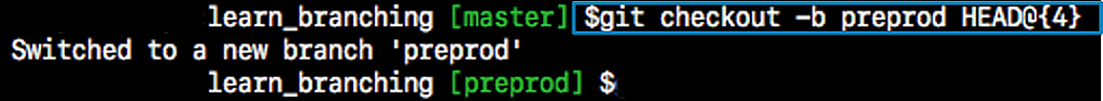
Get a list of all the local recorded history logs for all the references (‘master’, ‘uat’ and ‘prepod’) in this repository.



**Step 2: Identify the history stamp**

As you can see from the above snapshot, the highlighted commit id: e2225bb along with the HEAD pointer index:4 is the one when ‘preprod’ branch was created from the current HEAD pointer pointing to your latest work.

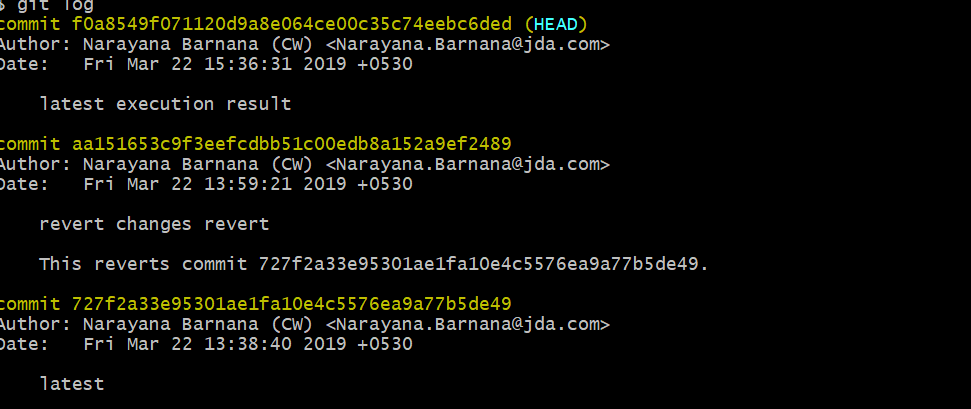
**Step 3**: **Recover**

If you want to recover back the ‘preprod‘ branch then use the command  ‘git checkout’ passing the HEAD pointer reference with the index id – 4. This is the pointer reference when ‘preprod’ branch was created long commit id highlighted in the output screenshot.

Log Commands:

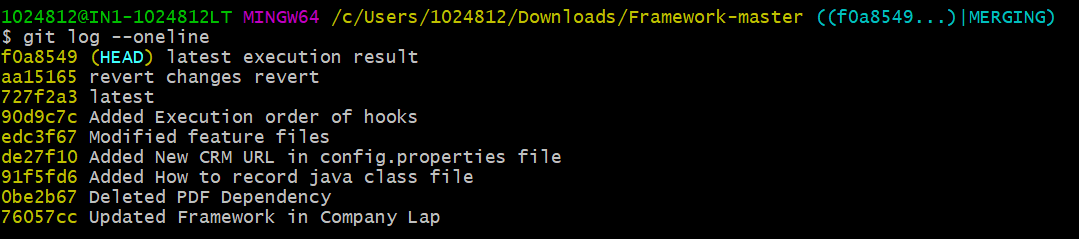
1)git log: shows commit logs

Command: git log



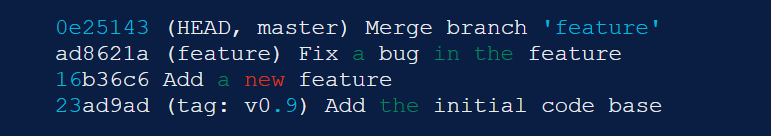
2)oneline: The --oneline flag condenses each commit to a single line. By default, it displays only the commit ID and the first line of the commit message. Your typical git log --oneline output will look something like this.

Command: git log --oneline



3)decorate:

Many times it’s useful to know which branch or tag each commit is associated with. The --decorate flag makes git log display all of the references (e.g., branches, tags, etc) that point to each commit.This can be combined with other configuration options. For example, running git log --oneline --decorate will format the commit history like so:



This lets you know that the top commit is also checked out (denoted by HEAD) and that it is also the tip of the master branch. The second commit has another branch pointing to it called feature, and finally the 4th commit is tagged as v0.9.

Branches, tags, HEAD, and the commit history are almost all of the information contained in your Git repository, so this gives you a more complete view of the logical structure of your repository

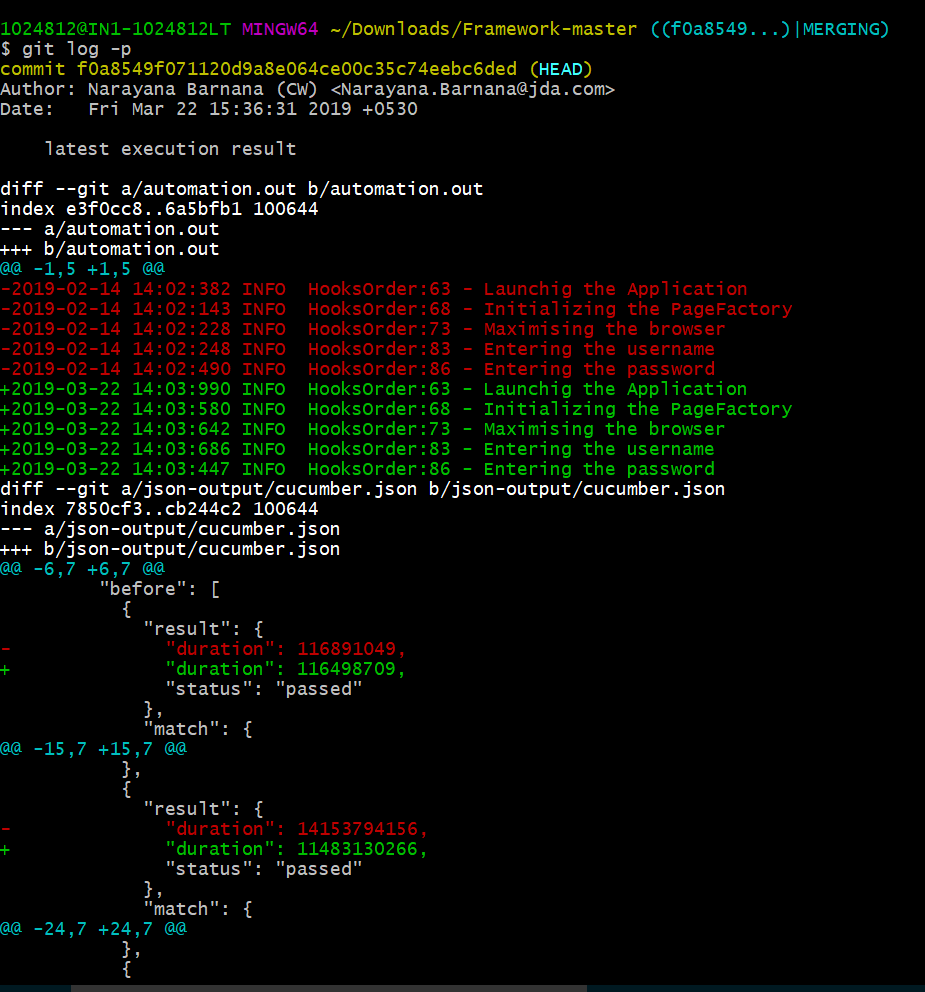
4)diff:

Command:

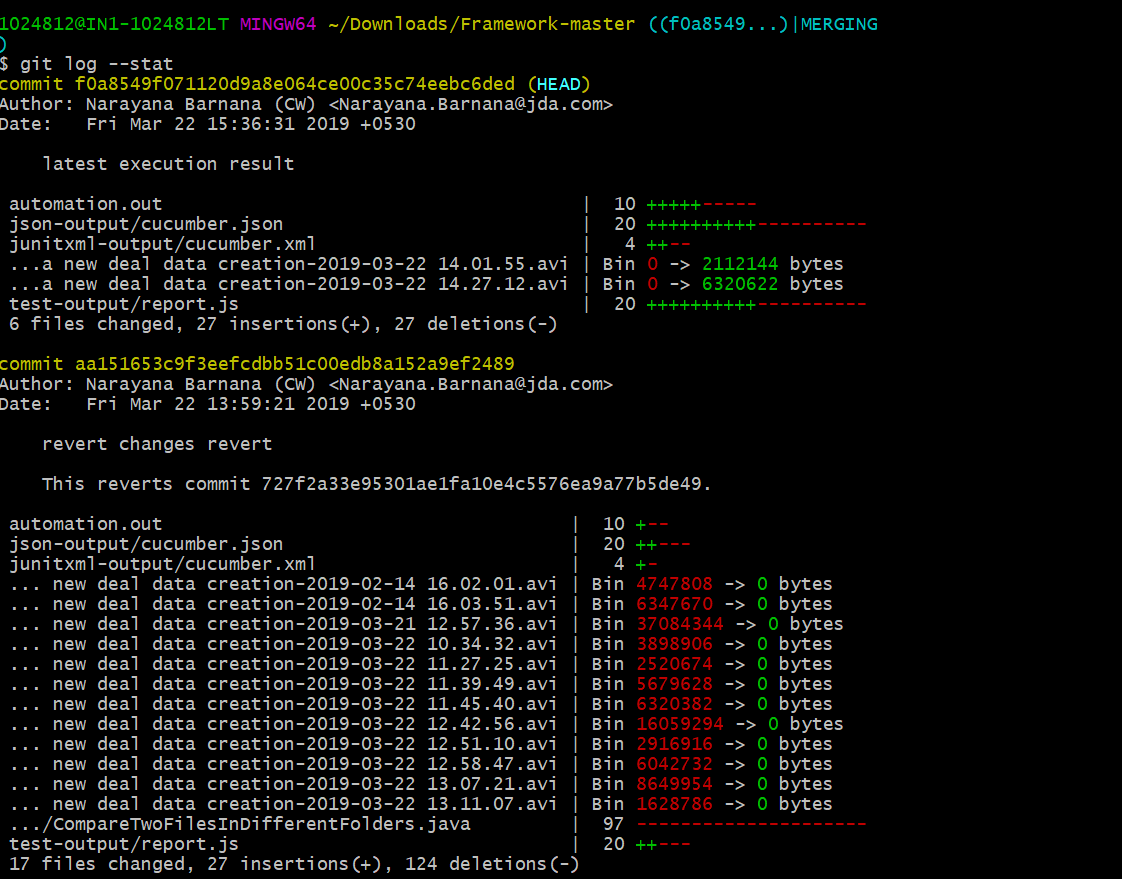
The git log command includes many options for displaying diffs with each commit. Two of the most common options are --stat and -p.

The --stat option displays the number of insertions and deletions to each file altered by each commit (note that modifying a line is represented as 1 insertion and 1 deletion). This is useful when you want a brief summary of the changes introduced by each commit. For example, the following commit added 67 lines to the hello.py file and removed 38 line

Git log -p



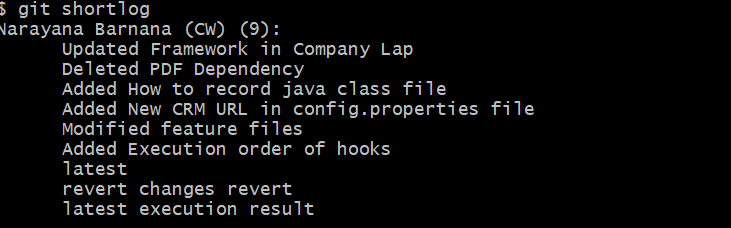
Git log –stat



**Shortlog:**

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.

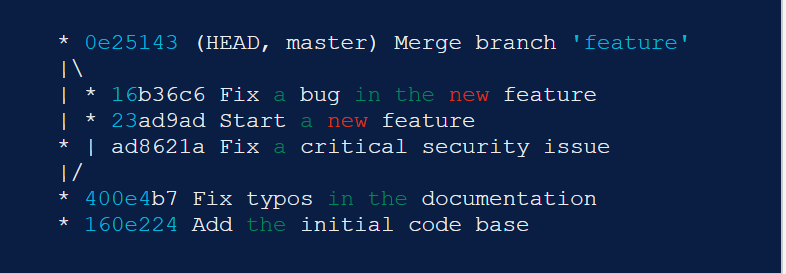


Graph:

The --graph option draws an ASCII graph representing the branch structure of the commit history. This is commonly used in conjunction with the --oneline and --decorate commands to make it easier to see which commit belongs to which branch:

Command: git log - -graph - -oneline – - decorate

For a simple repository with just 2 branches, this will produce the following:



The asterisk shows which branch the commit was on, so the above graph tells us that the 23ad9ad and 16b36c6 commits are on a topic branch and the rest are on the master branch.

While this is a nice option for simple repositories, you’re probably better off with a more full-featured visualization tool like gitk or Sourcetree for projects that are heavily branched.

**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.

You can limit git log’s output by including the -<n> option. For example, the following command will display only the 3 most recent commits.

Command: 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 1st, 2014 (inclusive):

Command: git log –after=”2014-7-1”

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 1st, 2014 and July 4th, 2014, you would use the following:

git log --after="2014-7-1" --before="2014-7-4"

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

**By Author**

When you’re only looking for commits created by a particular user, use the --author flag. This accepts a regular expression, and returns all commits whose author matches that pattern. If you know exactly who you’re looking for, you can use a plain old string instead of a regular expression:

git log --author="John"

This displays all commits whose author includes the name John. The author name doesn’t need to be an exact match—it just needs to contain the specified phrase.

You can also use regular expressions to create more complex searches. For example, the following command searches for commits by either Mary or John.

git log --author="John\|Mary"

Note that the author’s email is also included with the author’s name, so you can use this option to search by email, too.

If your workflow separates committers from authors, the --committer flag operates in the same fashion.

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:

git log --grep="JRA-224:"

You can also pass in the -i parameter to git log to make it ignore case differences while pattern matching.

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 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 Content

It’s also possible to search for commits that introduce or remove a particular line of source code. This is called a pickaxe, and it takes the form of -S"<string>". For example, if you want to know when the string Hello, World! was added to any file in the project, you would use the following command:

git log -S"Hello, World!"

If you want to search using a regular expression instead of a string, you can use the -G"<regex>" flag instead.

This is a very powerful debugging tool, as it lets you locate all of the commits that affect a particular line of code. It can even show you when a line was copied or moved to another file.

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>..<until>

This command is particularly useful when you use branch references as the parameters. It’s a simple way to show the differences between 2 branches. Consider the following command:

git log master..feature

The master..feature range contains all of the commits that are in the feature branch, but aren’t in the master branch. In other words, this is how far feature has progressed since it forked off of master. You can visualize this as follows:

Git Stash: Temporarily shelves the changes you have made to your working copy so you can work on something else, and then come-back and re-apply them later on.

Git stash

Git stash list

Git stash show stashid

Git stash pop

<https://www.atlassian.com/git/tutorials/saving-changes/git-stash>

git fetch

git fork

git diff: To see the changes done for the file in the working directory which is not present in the local directory. It compares the working directory with the local repo

Git diff –stage “name of file” : comparing the staging area with the local repo