# Versioning Commits

In Git, files can have the following statuses:

**Untracked**: This a file that exists in the working tree whose changes are not being monitored by Git and aren't listed in the **gitignore** file.

**Unstaged**: This a file whose changes are being tracked by Git; the file has been changed since the last commit and has yet to be moved to the **index**.

**Staged**: This is a file whose changes are being tracked by Git; the file has been changed since the last commit and has been moved to the index. This file is a file that is ready for the Git commit. Staged files are the files in the index that are different from their corresponding files in the **HEAD**, that is, the version of the file in the most recent commit.

**git status** is a utility that is used by Git. It's used to retrieve the details of files that are untracked, unstaged, or staged. **git status** lists files in order of their statuses.

The **git status** output is lengthy in nature. To view a brief list and status, use the **-s** or **--short** option with the **git status** command.

you should be able to use **git status** to view untracked, unstaged, and staged files.

To ensure flexibility, atomic commits are preferred and emphasized in version control. The term atomic commits refers to units of change that can be treated as a single unit. This normally implies a single file or a small set of files. This enables the addition and removal of changes without affecting a large set of files.

The **git diff** command is used to compare one snapshot of changes to another. As the name suggests, this utility supports evaluating the differences between two snapshots of a repository.

### Comparing the Working Tree to the Index

To compare the entire working tree to the index, run the git diff command without specifying a path:

$ git diff

This command supports examining the differences of a specific file or directory by accepting a path:

git diff -- [path\_to\_a\_file\_or\_directory]

1. git diff -- src/lib/
2. git diff -- src/lib/compute.py

(1) and (2) compare the version in the working directory of the specified paths to the version that's present in the index.

### Comparing the Working Tree to an Arbitrary Commit or Branch

A comparison can be made between the working tree and a specific commit on the same branch, or even the tip of a given branch. To compare the working tree to a given commit, use the following syntax:

$ git diff [commit\_hash] -- [path\_to\_a\_file\_or\_directory]

1. $ git diff HEAD -- src/
2. $ git diff f4e4e8d5b292dc94468b6f88223cac4f55c03713 -- src/lib/
3. $ git diff master

(1) Compares the version in the working directory of the **src** directory, to the version of the most recent commit on the current branch.

(2) Compares the version in the working directory of the src directory to the version in the snapshot represented by the hash f4e4e8d5b292dc94468b6f88223cac4f55c03713.

(3) Compares the version in the working directory of the **src** directory to the version at the tip of the branch **master**.

### Comparing the Index to an Arbitrary Commit

To compare the files in the index to a specific commit, (for example, the most recent commit, also referred to as the tip of the branch), you can use the **--staged** or **--cached** option with the **git diff** command. A commit hash is required for this scenario. The **git diff** command defaults to **HEAD** in the absence of a specific commit hash:

 **git diff --cached [commit\_hash]** or

**git diff --cached [commit\_hash] -- [path\_to\_a\_file\_or\_directory]**

1 **git diff --cached HEAD -- /src/lib/compute.py**

(1) compares the version in the index of the **compute.py** file to the version at the tip of the current branch.

### Comparing Commits and Branches

**git diff** provides a variant of the command's usage that supports comparing commits and branches.

To compare two commits or the tips of two branches, use the following syntax:

**git diff [commit\_hash or branch\_name] [commit\_hash or branch\_name]** or

**git diff [commit\_hash or branch\_name] [commit\_hash or branch\_name] -- [path\_to\_a\_file\_or\_directory]** or

**git diff [commit\_hash or branch\_name]..[commit\_hash or branch\_name]**

1. **git diff ft-add-encapsulating-class master**
2. **git diff ft-add-encapsulating-class..master**
3. **git diff da39a3ee5e6b4b0d3255bfef95601890afd80709 6f7e437faa5a7fce15d1ddcb9eaeaea377667b**

(1) and (2) compare the differences between the tips of the specified branches.

(3) compares the differences between the files at the point referenced by the specified hashes

Using the **...** notation, **git diff** is capable of comparing the changes that have been made on branch A to branch B. This occurs from the point where the two branches share an ancestor to the most recent commit of branch B.

To achieve this, use the following syntax:

**git diff [branch\_A]...[branch\_B] -- [path\_to\_a\_file\_or\_directory]**

1. **git diff ft-add-encapsulating-class...master**

(1) lists the changes that have occurred in the branch **master**, since the **ft-add-encapsulating-class** branch was created from the branch **master**.

Note

When .. or … are used with the **git diff** command, it implies the comparison of two points in history, and not a range.

**git diff** defaults to HEAD when a commit hash or branch name is not specified.

The **git add** command uses the following syntax:

**git add [options] [path\_to\_files]**

The **[options]** used with **git add** include the following:

**-n or --dry-run**

This option simulates the behavior of **git add** for the specified file:

**-f or --force**

This option adds ignored files to the index:

**-i or --interactive**

This option creates an interactive prompt that can be used to add files from the working tree to the index:

**-p or --patch**

The **--patch** option caters to adding portions of a file to the index

The **git commit** command saves the files in the index. This commit operation stores a message along with the commit. The message describes the additions or alterations associated with the created snapshot.

The syntax of this command is as follows: **git commit [options].**

The **git commit** command requires that a message be provided for each commit operation.

The options supported by this command include **-m [text] or --message [text].**

This message is used to associate the index file with the commit action:

**-a or -all**

This option instructs the **git commit** utility to stage tracked files that are unstaged, that is, the tracked files have been added to the index if the files are yet to be staged. Untracked files are not added to the index.

**-p or --patch**

This launches the interactive patch tool. The options are akin to those that are available through the **git add** command. See the preceding table for more information on this.

**-C [commit hash]** or **--reuse-message=[commit hash]**

This instructs **git commit** to reuse a commit message and the author information of the specified commit hash.

**-F [file]** or **--file=[file]**

This command specifies a file from which a commit message should be obtained.

**-t [file]** or **--template [file]**

This command specifies the commit message template file.

**-e** or **--edit**

This command edits the provided commit message. This refers to the message provided by the **-F**, **-t**, and **-m** options.

**--no-edit**

This command uses the specified message as is. Do not launch an editor to edit the message.

**--author=[author]**

This command overrides the details of a commit author, and takes the following form:

**git commit --author="Kifeh Polyswarm <kifeh@poly-swarm.com>"**

**--date=[date]**

As you can see, this overrides the date details used in a commit.

**-q** or **--quiet**

This command suppresses the summary message that's returned after running the **git commit** command

**git commit -C 474b5caaf480f7a367c1c456a53868c7fe32b9df --no-edit**

### *****git rm*****

The **git rm** command performs two roles. These roles are used to remove files from the working directory and the index.

**-r**

This option is applicable when using the **git rm** command in a directory. It removes the directory's contents recursively. This means that the directory and its contents are removed.

**--cached**

This option removes the specified files from the index only.

**-f** or **--force**

**git rm** checks the files marked for removal for matches, with the files in HEAD, at the tip of the current branch. This check is conducted before the file(s) are removed. The **-f** option overrides this check.

The ***rm*** command removes the specified file from the working tree only. The ***git rm*** command, on the other hand, removes the file from the index and the working tree. This provides a shorter process for deleting files, since with ***rm***, you need to run ***git add*** to impact the deletion process in the index.

### **git mv**

In the event that you need to update the index for both, old and new paths automatically, the **git mv** command serves that purpose.

This command has two forms of implementation:

1. git mv [options] [source] [destination]
2. git mv [options] [source] … [destination]

(1) is used to rename a file.

(2) is used to move a file.

## History and Logs

The **git log** command lists the history of a branch and the repository, by extension. It uses options and a range to define the duration for which the logs should be retrieved:

**git log [options] [version range] [path\_to\_file\_or\_directory] [version range]**

The **git log** command can display the history of a branch when given a range of version hashes:

**git log [hash\_1]..[hash\_2]**

### **Options**

### The **--follow** command retrieves and displays the history of a file, beyond rename events:

The **--decorate[=short** or **full** or **no]** command displays the **ref** name of the listed commits as seen in the following screenshots:

**git log --decorate=short**

### *The short option omits the* ***ref/heads/,ref/remotes/,*** *and* ***ref/tags/*** *prefixes from the* ***ref*** *name that is displayed.*

### *The full option displays the full ref name. It includes the* ***ref/heads/, ref/remotes/,*** *and* ***ref/tags/*** *prefixes in the ref name that is displayed.*

The **-L [start],[end]:[path\_to\_file]** command views the changes that have been made to a section of a file, from line number X to line number Y:

**git log -L 6,12:./src/lib/compute.py**

The **--[number]** , -**n [number]** ,and **--max-count=[number]** the specified number of commits only

### **git log -3 or git log -n 3**

The **--skip=[number]** command skips the specified commits and displays the rest:

**git log --skip=4**

The **--since=[date]** or **--after=[date]** commits that have been created after a given date:

**git log --after=25/08/2018**

The **--until=[date]** or **--before=[date]** commits that precede a given date:

**git log --before=24/08/2018**

The **--pretty=[format]** command displays the history of a branch using a prescribed format:

**git log --pretty=oneline**

### ***git log --pretty=medium***

**git log --pretty=format:[format string]**

The format is a string that's in the form of **%placeholder1 %placeholder2 %placeholderN**.

The supported placeholders include the following and are shown in the following screenshots:

* **%H**: The commit hash
* **%h**: The abbreviated commit hash
* **%T**: A tree hash
* **%t**: An abbreviated tree hash
* **%P**: The parent hash
* **%p**: The abbreviated parent hash
* **%an**: Author name
* **%ae**: Author email
* **%ad**: Author date
* **%ar**: A relative author date
* **%at**: The Unix timestamp version of the author date
* **%s**: The subject of the commit
* **%b**: The body of the commit
* **%n**: A newline

**git log --pretty=format:"%H %an"**

### **git log --pretty=format:"%H %an %ae"**

### **git log --pretty=format:"%H %an %ae %n %s %n %b"**

### Amending a Single Most Recent Commit

The most recent commit can be edited by using **--amend** in the **git commit** command.

**Amending Multiple Commits**

The **git rebase** command provides the **reword** and **edit** options to edit the commits. The **reword** option allows you to edit a message, while the **edit** option supports editing a commit message, as well as the contents of a commit.

## Fetching and Delivering Code

the **git remote** utility, to explore how we can manage the connection between the local and upstream repositories.

The **git remote** utility includes commands that help with managing the remote/upstream repositories that are associated with a local repository.

add the remote tracked repository by using the **git remote add origin git@github. com:kifeh-polyswarm/remote-demo.git**

View the remote configuration by using the **git remote -v command**

Rename the remote configuration from **origin** to **source-truth** by using the **git remote rename origin source-truth command**

View the specifics of the **source-truth remote** by using the **git remote show source-truth** command:

*The* ***prune*** *command removes local branches that correspond to branches that have been deleted from the remote repository:*

**$ git remote prune source-truth**

The configured remote can be removed by using the **remove** command. For example, to remove the **source-truth** remote, run **git remote remove source-truth:**

### Fetching, Pushing, and Pulling Changes

To enable collaboration in a distributed version control system, Git provides the means to retrieve and publish your contributions to the shared repository. To demonstrate this, we will explore **git fetch, git push,** and **git pull**.

**git fetch**

To navigate changes that were made to a repository, you need to utilize references to the changes made to the repository, and consequently a branch. This command allows you to explore the changes before integrating them into your work.

The **git fetch** command downloads remote-tracking branches and tags from the remote repository. These branches and tags indicate changes that have been made to the remote repository.

The content retrieved by this command is isolated from the content in the local repository, and, when you do this, the work that is being undertaken locally isn't affected by the downloaded content.

*Remote-tracking branches keep track of changes occurring on the branches in the remote repository. Remote-tracking branches can be viewed by using the* ***git branch -r*** *command. The* ***refs*** *for remote-tracking branches are stored in* ***/.git/refs/remotes/[remote\_name]****; for example,* ***/.git/refs/remotes/origin.***

you can retrieve or download all of the branches of the repository specified by **[remote]** with **git fetch [remote]** e.g. **git fetch origin:**

**git fetch [remote] [branch]**

You can fetch the **[branch]** from the upstream repository specified by **[remote]** with **git fetch origin master:**

**git fetch --all**

You can fetch branches from all of the remote connections defined for a repository. For example, if you have a remote **origin** and **upstream**, as you will see in the Chapter 4: Branches, this command will fetch branches from the two repositories identified by **origin** and **upstream**:

**git fetch --prune**

You can remove remote-tracking references that have ceased to exist in the remote repository, and then proceed to fetch the branches and their corresponding commits, files, and **refs** with: **+refs/heads/\*:refs/remotes/origin/\***.

#### Note

The **git fetch** command uses the **refspec** defined in the repository-level config. This is defined in the **remote.[remote\_name].fetch** config value. The **refspec** can be retrieved by running the **git config --local --list** command.

The **refspec** dictates that **refs** stored in the remote repository in **refs/heads/** are tracked locally, under **refs/remotes/origin/.**

The **+** indicates that references should be updated, including in scenarios where the commit is not a **fast-forward.**

The **git push** command picks local commits and updates the remote branch with the local commits. By default, the **git push** command only supports pushing commits in a fast-forward mode. If the commits being pushed are non-fast-forward, you're required to either push changes by force, or update the local repository by merging the commits from the remote branch

This command takes the form of **git push [remote\_name]**

You can push commits from the current branch to the remote branch configured on the repository with **git push origin**

**git push**

This is similar to **git push [remote\_name]**

**git push [remote\_name] [branch\_name]**

This variant will push commits from the local branch to the specified branch on the specified remote repository. You can use **git push origin develop** for this

**git push origin [local\_branch]:[remote\_branch]**

This command creates a new remote branch, bearing the commits in the **[local\_branch]**. This command is useful when the local branch and the remote branch do not have matching names. You can use **git push origin feature-video:video-experiment:**

**git push origin :[branch\_name]**

This variant deletes the specified branch. You can use **git push origin :bug-logout-mobile** for this

There are several options that are commonly used with the **git push** command. These include the following options:

The **--all** local branches. These branches are the branches that correspond to the **heads** stored in **.git/refs/heads/.** You can use **git push --all** for this:

**--force or -f**

This forces **git push** to update the remote branch in the event non-fast-forward changes are encountered. You can use **git push -f origin master** for this**:**

**git push --force origin maste**

You can set a tracking reference to the upstream branch for the current branch by using **git push -u origin master.**

### Dealing with Non-Fast-Forward Commits

As we explained earlier, **git push** utilizes a fast-forward mode to merge commits. Non-fast-forward commits are therefore rejected, in order to avoid the loss of source code. To deal with this, it's advised that you use **git fetch** to retrieve the updated remote branch, and then run **git merge [remote\_name]/[branch\_name]** to merge the changes from the remote branch. Finally, run **git push** to upload your changes to the remote branch.

The **git rebase** command can also be used to integrate changes from the remote branch.

Git pull

The **git pull** command is an alternative to using the **git fetch** mechanism to retrieve and integrate changes in the remote repository.

The **git pull** command, in its default mode, runs a combination of **git fetch** and **git merge**. The **git pull** utility may use a **rebase** mechanism, if it's specified as the synchronization mechanism in the place of a **merge**

The **git pull** syntax is as follows:

**git pull [options] [remote\_name] [branch\_name or refspec]**

**git pull [remote]**

The preceding code retrieves changes from the remote branch that's been configured as the remote-tracking branch for the current branch and merges the changes into the local branch. This can be done with **git pull origin.**

**git pull**

This is similar to **git pull origin.**

You can fetch the remote branch **develop** and merge it into the current branch with **git pull origin develop.**

--no-commit

This option instructs **git merge** to integrate the changes from the remote repository, and to not perform a commit; for example, **git merge --no-commit origin master**

#### Note

In the event that the commits being merged are resolved to be fast-forward, the **--no-ff** option should be used to override the default behavior, which is to update the branch pointer without creating a merge commit; for example, **git merge --no-commit --no-ff origin master.**

**--no-ff**

This option creates a merge commit, including in a scenario where commits resolve as being fast-forward; for example, **git merge --no-ff origin master.**

**--ff**

This is the opposite of **--no-ff**. It updates the branch pointer of the current branch to the tip of the branch with the incoming changes commits resolved as being fast-forward; for example, **git merge --ff origin master.**

**--edit**

The **git pull** command fetches and merges the changes, and launches an editor to allow for the editing of the automatically generated commit message; for example, **git merge --edit --no-ff origin master**

**--no-edit**

This instructs the fetch and merge process of **git pull** to accept the auto-generated commit message; for example: **git merge --no-edit --no-ff origin master--strategy=[strategy].**

This option specifies the merge strategy to be used for the **git merge** step of **git pull**.

The strategies used by git are **resolve**, **recursive**, **octopus**, **ours**, and **subtree**.

**--strategy-option=[option]**

This sets a strategy option that's specific to the strategy specified by the **--strategy** command option. The options for the **recursive** strategy include **ours**, **theirs**, **no-renames**, **ignore-all-space**, **ignore-space-at-eol**, and **ignore-cr-at-eol**.

### Reversing Commits

Git provides multiple approaches to reversing changes that have been introduced to a file.

The **git revert** command uses the following syntax:

**git revert [options] [commit(s)]**

The options supported by **git revert** are as follows:

**--edit**

This option provides support for editing the commit message for a given reversal; for example, **git revert --edit [commit hash].**

**--no-edit**

The **--no-edit** option overrides the **--edit** option. It performs the revert with the default auto-generated commit message.

**--no-commit**

With this option, the **git revert** command makes changes to the index and the working tree, in order to reverse the modifications made by the given commit. It does this without creating a commit for the reversal.

**--mainline [parent-number]**

Two sets of changes exist in a merge commit, referenced by each of the two parents that constitute a merge commit. This option dictates which of the two sets should be used in the reversal process. We will explore this further in the cherry-pick section.

### Other Possible Uses

**git revert [least\_recent\_commit\_hash]..[most\_recent\_commit\_hash]**

**git reset**

The preceding code shows how the **git revert** command is a forward-moving **undo** operation that provides an effective means of undoing changes.

The **git reset** command is used to roll back a file, directory, or repository at large to a given point in its history. This entails updating the **index**, **working tree,** and **commit history** of a repository where applicable, based on the options passed in the command.

Git uses objects to track changes throughout the history of a repository. To achieve this tracking, Git uses four types of objects. The objects are **commits**, **trees**, **blobs,** and **tags**. These objects are stored in **.git/objects**.

>ls -al

A commit object stores the hash of the directory tree object that the commit corresponds to, the parent commit hash, the author, the committer date and time, and the commit message:

**git cat-file -t 11b8b15**

**git cat-file -p 11b8b15**

**BLOBs**

Git uses blobs to store the contents of a file at a given point in time. A blob is a **Binary Large OBject (BLOB)**. It's Git's methodology of storing the contents of a file at a given point in its lifetime. A blob is created when we commence the tracking of a file by using the **git add** command:

**Tags**

A **tag** is a pointer to a specific commit. The tag object contains the hash of the tagged object, the type of the object that the tag was created for, the tag name, and the the author details, such as the author name, date, and message:

**git tag -a v1.4 ab30a24 -m "This is demo commit"**

The **git reset** syntax follows the following form:

**git reset [tree-ish] [path(s)]**

**git reset [mode] [commit]**

To demonstrate how **git reset works**, we will explore the aforementioned variants and how each works to examine and establish how each affects a repository:

**git reset [tree-ish] [path(s)]**

In Git, **tree-ish** refers to identifiers that reference a tree object, that is, a directory or subdirectory at a given point in a repository's history.

This variant of **git reset** resets the **[path(s)]** index to its state at **[tree-ish]**. The files and directories in the working tree are not affected:

For example. **git reset master:./test.txt test.txt**

**git reset 5d2045c test.txt**

**git reset HEAD:README README**

**git reset :/Adding the word 'boat' -- test.txt**

The text **Adding the word 'boat'** represents a commit message.

**--patch**

The **--patch** option allows you to choose portions of the differences between the index and **[tree-ish]** to reset to their respective states at **[tree-ish]**; for example, **git reset --patch 5d2045c test.txt.**

**git reset [mode] [commit]**

This variant resets the head of the current branch to the state at **[commit]**. Additionally, the index and working tree are reset, depending on the mode used to conduct the reset.

The supported modes are as follows:

**--soft**

This resets the head to **[commit]**. The index and the working tree are not altered by this option.

**--hard**

The **--hard** mode resets the head to **[commit]**. The index and working tree are reset to **[commit]**. Changes introduced to the working tree are discarded.

**--mixed**

This mode resets the head to **[commit]**. Additionally, it resets, the index to **[commit]**. The changes made to the working tree are not altered or discarded.

The commit object stores a snapshot of the directories and files that constitute a repository at a given point in time. In addition, the commit specifies auxiliary information, which includes the parent commit of the created commit, the author, the committer date and time, and the commit message:

A branch is therefore a pointer to a snapshot of the repository. This pointer refers to the commit at the tip of the branch. These tips are the commit hashes stored in **.git/refs/heads/. HEAD** is the pointer that references the commit at the tip of the current branch. This commit is imperative because it's based on the fact that git is able to navigate the history of a repository with the help of the parent-child association between commits. The creation of a branch, in turn, creates a pointer and the head, which bears a branch's name. Navigating from one branch to another updates the **HEAD** to refer to the tip of the branch you switch to – or in Git terms, check out to.

### Creating:

**git branch [branch\_name]**

**git branch --set-upstream-to [remote\_branch\_name]**

e.g. **git branch --set-upstream-to origin/ft-support-exponents**

**git branch --unset-upstream [branch\_name]**

**git branch [branch\_name] [start\_point]**

**Renaming:**

**git branch -m [old\_branch\_name] [new\_branch\_name]**

**git branch -M [old\_branch\_name] [new\_branch\_name]**

This is similar to invoking **git branch** with the **--move** and **--force** options.

**git branch -c [old\_branch\_name] [new\_branch\_name]**

**Copy:**

**git branch -C [old\_branch\_name] [new\_branch\_name]**

This is similar to invoking **git branch** with the **--copy** and **--force options.**

**Deleting:**

**git branch -d [branch\_name]**

Delete a branch, granted that it's fully merged into its upstream branch or the **HEAD**, in the event that the upstream branch is not specified.

**git branch -D [branch\_name]**

This forces the deletion of a branch.

It's similar to using **--delete --force.**

**Listing:**

**git branch --list**

**git branch --list [pattern]**

For example, you can use **git branch --list 'ft\*'.**

**git branch --contains [commit]**

For example, you can use **git branch --contains 8354043.**

**git branch --no-contains [commit]**

**git branch --merged [commit]**

For example, you can use **git git branch --merged 8354043.**

This lists branches that have been merged into a given commit, that is, commits whose tip is reachable from the given commit.

**git branch --no-merged [commit]**

This is used for branches that are not merged into the given commit.

**git branch -a**

This is used to get all branches:

**git branch -r**

This is used for the remote tracking of branches.

**Switching to New and Existing Branches**

The process of moving from one branch to another is done by switching to **[branch]** and then setting the files in the index and working tree to reflect [branch]'s latest commit. Lastly, you must set the **HEAD** to branch:

**git checkout [branch\_name]**

**git checkout -b [branch\_name]**

**git branch -B [branch\_name] [start\_point]**

Create a new branch and set its tip to **[start\_point]**. If the branch exists, then reset it to **[start\_point]**.

**git checkout -b sample**

**git branch sample**

**git tag v1.0**

**git checkout [commit\_hash]**

**git checkout [tag]**

**git checkout --detach [branch]**

**Switching to a Specific Version of a File**

When switching, the git check out command takes the following syntax:

**git checkout [commit] -- [path]**

Other uses of **git checkout are as follows:**

**git checkout -b --orphan [new\_branch] [start\_point]**

This creates a branch whereby the first commit has no parent. This is necessary when certain information contained in the repository's history needs to remain unexposed for privacy reasons.

**Incorporating Changes with Stashing**

In the book of development work, emerging requests are a typical occurrence, including in scenarios where you are attending a planned work stream over a specific period of time. The book of action in this scenario, normally, is to put aside what you're working on and attend to this request, be it an emergency or not.

How does Git enable you to "put away" what you're working on without losing the progress you'd achieved on a certain task? Ask no more.

In comes **git stash**. The **git stash** command temporarily moves staged, unstaged, or untracked modifications made to a repository, to and from the index and working directory.

To achieve this with **git stash**, use the following subcommands:

**git stash push -m [message] or git stash push —message [message]**

This saves modifications to a stash list and reverts the index and the working tree to the state reflected by HEAD.

The **—keep-index** option retains changes made to the index. This means that the modifications in the index are not reverted.

The **—include-untracked** option includes untracked files in the stash entry made to the stash list.

The **—all** option stashes ignored files in the stash entry made to the stash list.

#### Note

**git stash save** was deprecated in favor of this command.

**git stash list**

This command lists the entries in the stash list. These are all of the created stashes:

**Git stash show [stash\_id]**

This command displays the changes introduced by the stash identified by **[stash\_id].**

**git stash apply [stash\_id]**

This updates the working directory with the changes stored in **[stash\_id].**

**git stash pop [stash\_id]**

This updates the working directory with the stash **[stash\_id]** and removes it from the stash list.

**git stash drop [stash\_id]**

This removes the specified stash from the stash list.

**git stash clear**

This removes all stashes from the stash list.

**git stash branch [branchname] [stash\_id]**

#### Note

The stash list is available from every branch

**Merging**

A merge takes one of two modes, namely:

1. **Fast-forward merge**
2. **Three-way merge**

Merging is achieved by using the **git merge** command. This command uses the following syntax:

**git merge [options] [branch\_name]**

**--no-commit**

This option merges changes into the current branch. However, the command does not create a merge commit in order to leave room for evaluating the result of the merge.

**--edit**

This option launches the editor to allow for editing of the generated commit message.

**--no-edit**

This conducts the merge using the message generated by the command.

**--no-ff**

This option creates a merge commit in all merge scenarios, including when the merge resolves to a fast-forward merge.

**--squash**

This option instructs **git merge** to update the index and working directory to reflect the incoming changes without creating a commit. Using this option enables you to create a commit as part of the current branch, thus referencing the incoming changes. With this option, the **HEAD** is not changed and the **MERGE\_HEAD** ref is not recorded. As a result, the subsequent commit is not a merge commit.

**--strategy=[strategy]**

This specifies the merge strategy to be used for the merge.

The supported [strategy] includes

**--strategy-option=[strategy\_option]**

This sets the option that's specific to the provided strategy.

### Cherry-Pick

The **git cherry-pick** command takes a commit from one branch and applies the specified commit to another branch.

**git cherry-pick** is useful when you wish to check the effects of certain changes that have been introduced to a branch you're working on.

The syntax of the **git cherry-pick** command is a follows:

**git cherry-pick [options] [commit].**

**git cherry-pick** supports options that dictate how the introduced commits are handled. This includes the following:

**-x:** This option adds a standardized message to the commit of the form **"cherry picked from commit ..."** to specify the commit that introduces the incoming change.

**--edit:** This allows you to edit the commit message for the incoming changes.

**--no-commit:** You may wish to integrate changes from a specific commit without creating a corresponding commit. The **--no-commit** command integrates changes from a commit without creating a commit.

**--mainline [parent\_number]:** Since a parent commit possesses two parents, running a cherry-pick against a merge commit requires that a parent is specified. The given parent is compared to the merge tree of the merge commit and the resulting difference is introduced into the branch where **git cherry-pick** is invoked from.

**git cherry-pick -m 1 merge\_commit**

By specifying -m 1, we choose parent-1.

**Merge Conflicts**

A **merge conflict** is a term that depicts an issue whereby modifications made in separate branches can't be amalgamated into one unit of change or modification.

A merge conflict will occur when:

1. Modifications are made on the same line of a file.
2. Changes are made to a file on one branch and the same file is deleted on another branch.

Merge conflict resolution encompasses picking which of the differing sets of changes should be used in a merge process.

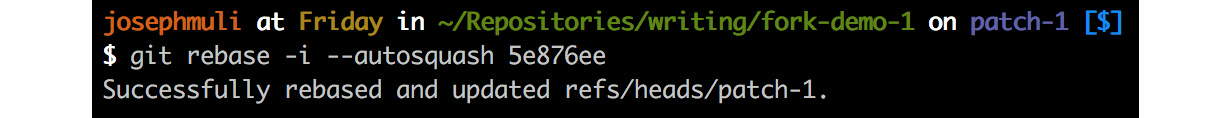
**Fixup and Squash Commits**

### Utilizing the Autosquash Feature

To **fixup** the previous commits using the autosquash facility in GitHub.

Instead of a normal commit, include a **--fixup** argument and a commit hash to the command as follows: **git commit --fixup dd4b896**

1. To clean this up, rebase and autosquash the commit, as shown in the following screenshot:



###### Figure 5.42: Rebasing and autosquashing the commit

#### Note

Be sure to select the commit hash that's just before the commit you want to autosquas

If you are running VIM as I am, select **I** on your keyboard to use the **Insert** option and edit the command. To drop and escape and select **:wq** without the quotes to write and quit, as shown in the following screenshot:

### Submodules

### Submodules require a **.gitmodules** file and dependencies to be installed through the **git submodule** command.

Git submodules, often referred to as gitmodules, enable the **separation of concern** toward project dependencies. In other words, they help organize code based on a **single responsibility principle**.

When handling submodules for the first time, be sure to do the following:

* Add submodules through the **git submodule add <repository url>** command.
* Initialize the submodules through the **git submodule init** command.

**git submodule update** will fetch submodules that are not present in your project path, as shown in the following screenshot:

**git clone --recurse-submodules: https:github.com/<username>/abacus-scripts** clones the repository and submodules defined in a **.gitmodules** file.

**git blame**

**git blame** helps identify and present occurrences from a revision that has modified a block of code. Usually, this is done line by line. This means that through the command, we can identify who made a change, the commit used, and what line(s) were affected. This also makes Git a very effective audit tool.

**git blame** includes several other options, such as showing the filename in the original commit, whereby the default filename is displayed if a change comes from a different file, with a different name, as shown in the following screenshot:

git blame –f totodo.py

**git bisect**, as described on Git's official documentation, is used to find a commit that has introduced an anomaly through a binary search. This is most helpful when trying to identify a commit to test a failing feature after previous success.

$ git bisect start

$ git bisect bad XXXX

$git bisect good XXXX

**git reflog**

**git reflog** is short for Git **reference logs**. Reflogs keep track of changes to **HEAD** over a defined period of time. These changes can be best defined as events, as they are saturated, which is basically all activities, that is, checking out branches, rebase events, and branch updates from remote URLs.

To get the reflog on a specific branch, pass the branch name as an argument, as shown in the following screenshot:



## Housekeeping

In this subtopic, we will highlight a number of best practices that can be applied toward maintaining a clean and operable repository through the following commands:

* **git clean**
* **git gc**
* **git prune**

**git clean**

**git clean** recursively removes untracked files from a working tree. This emphasizes that any file that is not staged to be tracked or reset is rid of, maintaining a versioned only directory. Normally, **git clean** purges files through a list defined from a **.gitignore** file, but in special cases, these rules can be ignored and any untracked file is cleared.

*The* ***-n option*** *will enable a dry run accompanying* ***git clean***

To remove the file, the **-f** and **-i** options can be utilized

*The* ***-i*** *option presents an interactive mode that presents an interactive session*

**git gc**

**git gc** is responsible for garbage collection on targeted repositories. This process handles the deletion of staged and committed objects from unreachable branches, particularly those holding a reflog. Depending on the repository activity, **git gc** helps optimize disk space and maintains a decent repository.

To verify whether any housekeeping is required, **git gc** appends the **--auto** option and if not necessary, no output shall be displayed, as demonstrated in the following screenshot:

To proceed without necessarily requiring housekeeping, run the following command, and optionally append **--force**, as shown in the following screenshot:

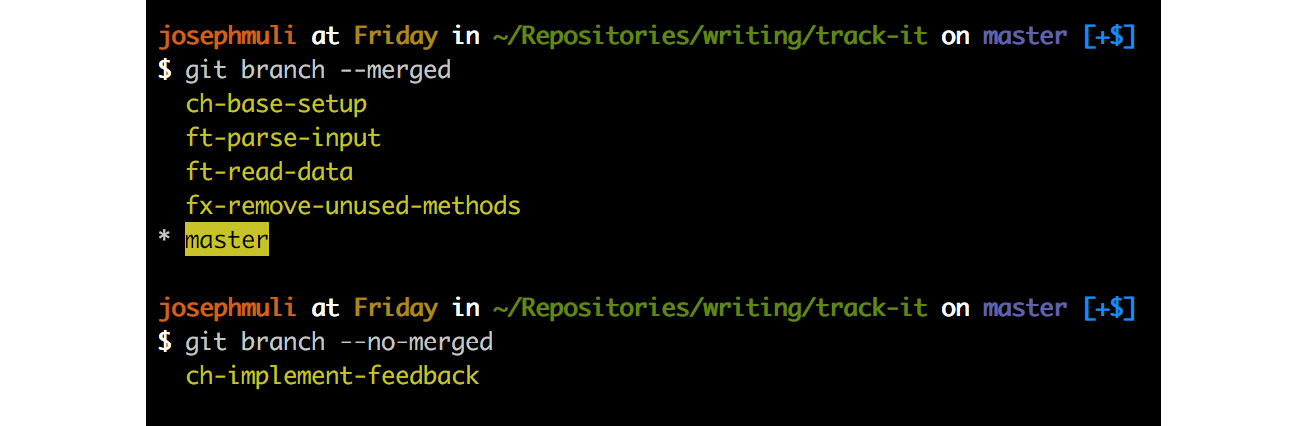
**git prune**

The **git prune** command, similar to **git gc**, gets rid of all unreachable objects, that is, basically objects without references, such as ones from deleted branches. According to the documentation, it's best to go for **git gc,** which calls **git prune**, killing two birds with one stone.

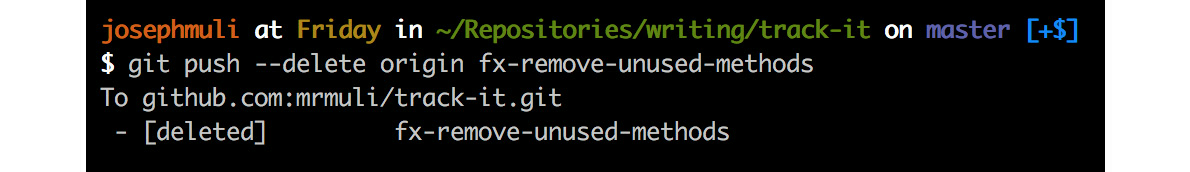
**Removing Merged Local and Remote Branches**

Manual deletion locally.

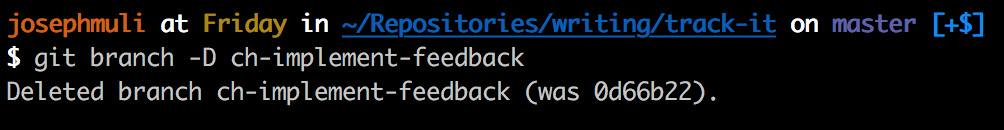
Before we start taking out branches, we need to be aware of merged and not unmerged branches. The following commands help us identify this:



To delete the remote branch reference, run the following command:



To delete a branch that is not merged, run the command with a **-D** option, as shown in the following screenshot:



**…or create a new repository on the command line**

echo "# test3" >> README.md

git init

git add README.md

git commit -m "first commit"

git remote add origin https://github.com/pvtreservoir/test3.git

git push -u origin master

**…or push an existing repository from the command line**

git remote add origin https://github.com/pvtreservoir/test3.git

git push -u origin master

**git log -5 --pretty=format:"%H %an %ae %n %s %n %b"**

**git rebase -i HEAD~4**