Contents

[1. Introduction 4](#_Toc158126854)

[2. Understanding Git Concepts 5](#_Toc158126855)

[A. What is Git? 5](#_Toc158126856)

[B. The Git Object Database. 6](#_Toc158126857)

[1) Git Objects 6](#_Toc158126858)

[2) Git references 8](#_Toc158126859)

[C. The Git promotion model 11](#_Toc158126860)

[1) The working directory 12](#_Toc158126861)

[2) The staging area 13](#_Toc158126862)

[3) The local repository 14](#_Toc158126863)

[4) The remote repository 14](#_Toc158126864)

[D. Commands in Git 15](#_Toc158126865)

[Basic Git 17](#_Toc158126866)

[Inspecting Repositories 18](#_Toc158126867)

[E. Exploring the Object Database 18](#_Toc158126868)

[3. Git Configuration 24](#_Toc158126869)

[A. Initialize the Git Repository 24](#_Toc158126870)

[B. Configure git 24](#_Toc158126871)

[Writing a value 28](#_Toc158126872)

[Writing a value 28](#_Toc158126873)

[git config editor - core.editor 29](#_Toc158126874)

[Merge tools 29](#_Toc158126875)

[Aliases 29](#_Toc158126876)

[**Formatting & whitespace** 29](#_Toc158126877)

[Summary 30](#_Toc158126878)

[Git Config 30](#_Toc158126879)

[Changing your Editor 30](#_Toc158126880)

[Adding Aliases 31](#_Toc158126881)

[Commit Template 31](#_Toc158126882)

[Log Format 31](#_Toc158126883)

[4. Committing 32](#_Toc158126884)

[C. Git Add 32](#_Toc158126885)

[Git Status 33](#_Toc158126886)

[Git Commit 34](#_Toc158126887)

[Git rm 36](#_Toc158126888)

[Git rm Overview 37](#_Toc158126889)

[Usage 37](#_Toc158126890)

[How to undo git rm 38](#_Toc158126891)

[Discussion 39](#_Toc158126892)

[The scope of git rm 39](#_Toc158126893)

[Why use git rm instead of rm 39](#_Toc158126894)

[Examples 39](#_Toc158126895)

[How to remove files no longer in the filesystem 40](#_Toc158126896)

[Git rm summary 40](#_Toc158126897)

# Introduction

It is an introduction to Git that is currently the most widely used version control system in the world. The goals of this guide are:

* To shed some light on how Git works under the hood.
* To present the commands (basic and advanced) that you will need to know in order to use Git effectively.

# Understanding Git Concepts

## What is Git?

A version control system is a software designed to keep track of the changes made to files over time. There are a number of benefits to using VCS including the following:

* The ability to undo changes. You can recover an earlier version of you work
* A complete history of all the changes
* Documentation of why changes are made.
* Multiple streams of history.

One of the most popular VCS tools in use today is called Git. Git is a *Distributed* VCS, a category known as DVCS.



Centralized version control



Distributed version control

The essential difference between a Centralized Version Control System (CVCS) and a DVCS is the notion of a *repository instance.* With a DVCS there was no longer a central repository, everyone gets their own local repository and could develop at their own pace, store the updates locally, and put off merging conflicts until their convenience. The local nature of DVCSs also made development much faster, since you no longer had to perform actions over a network. The only time networking code gets involved is when the repository instances are being synchronized.

And, since each developer had a complete copy of the project, the risk of a server crash, a corrupted repository, or any other type of data loss was much lower than that of their CVCS predecessors.

Git is an open source distributed version control system created in 2005 to manage the entire Linux kernel. The Git project spread rapidly, and quickly became used to manage a number of other projects. Git is the technology behind the enormously popular “social coding” website GitHub,.

The key difference here is that, in a DVCS such as Git, users are performing the source management operations against a local copy of the server-side (remote) repository instead of making them against the actual server-side repository. Until users need to push the changes back to the remote, they do not even need to be connected to it. The connection between the local and the remote side is not constant. Rather, it is activated when updates need to be synchronized between the two repositories.

## The Git Object Database.

### Git Objects

The data model of Git is different from other common **version control systems** (**VCSs**) in the way Git handles its data. Traditionally, a VCS will store its data as an initial file, followed by a list of patches for each new version of the file. Git does not do this, it records a snapshot of all the files tracked and their paths relative to the repository root. Each commit in Git records the full tree state. If a file does not change between commits, Git will not store the file again.

Git is a version control system built on top of a key *value object store*. Git creates and stores a collection of objects when you commit. The object store is stored inside the Git *repository*. It exists entirely in a single .git directory in your project root. There is no central repository like in Subversion. The key is an SHA-1 hash of the object and the value is the object itself. The SHA1 hash is a cryptographic hash function. The key **SHA-1** ([https://en.wikipedia.org/wiki/SHA-1)](https://en.wikipedia.org/wiki/SHA-1)s) is an alphanumeric sequence of 40 characters representing a hexadecimal number.

This **hash**, as it is usually called, uniquely identifies the commit within the repository. What that means to us is that it is virtually impossible to find two different objects with the same name.

Git objects are the actual data of Git, the main thing that the repository is made up of.

All of these types of objects are stored in the Git Object Database, which is kept in the Git Directory. Each object is compressed (with Zlib) and referenced by the SHA-1 value of its contents plus a small header (SHA stands for Secure Hash Algorithm)

In Git, the contents of files are stored as blobs. It is important to note that it is the contents that are stored, not the files. The names and modes of the files are not stored with the blob, just the contents.

Directories in Git basically correspond to **trees**. A tree is a simple list of trees and blobs that the tree contains, along with the names and modes of those trees and blobs.

Git uses objects to track changes throughout the history of a repository. To achieve this tracking, Git uses four types of objects:

* **Blobs**. Git uses blobs to store the contents of a file. A blob is a **Binary Large OBject (BLOB)**. A blob is created when we commence the tracking of a file by using the git add command. Since the blob is entirely defined by its data, if two files in a directory tree (or in multiple different versions of the repository) have the same contents, they will share the same blob object. The object is totally independent of its location in the directory tree, and renaming a file does not change the object that file is associated with.
* T**rees**. A tree object in Git can be thought of as a directory. It contains a list of blobs (files) and other tree objects (sub-directories).
* **Commits**:A commit object is essentially a pointer that contains a few pieces of important metadata. The commit itself has a hash, which is built from a combination of the metadata that it contains:
  + - The hash of the tree (the root tree object) at the time of the commit.
    - The hash of any parent commits. This is what gives a repository its history: every commit has a parent commit, all the way back to the very first commit. This implicitly forms a graph of commits known as the commit graph. Specifically, it's a directed acyclic graph (or DAG).
    - The author’s name and email address, and the time that the changes were authored. The author is the name of the person responsible for this change.
    - The committer’s name and email address, and the time that the commit was made. The committer is the name of the person who actually created the commit, with the date it was done. This may be different from the author; for example, if the author wrote a patch and emailed it to another person who used the patch to create the commit
    - The commit message.
* **Annotated Tags** which point to a single commit object, and contain some metadata.it is a way to mark a specific commit as special in some way.

The final type of object you will find in a Git database is the **tag**. This is an object that provides a permanent shorthand name for a particular commit

So, the objects are tied together, blobs to trees, trees to other trees, and the root tree to the commit object, all connected by the SHA-1 identifier of the object. Almost all of Git is built around manipulating this simple structure of four different object types.

In short, the Git data model can be summarized as shown in the following diagram:



the Git object data is a *directed acyclic graph.*

The cheap references I’ve represented as the grey boxes, the immutable objects are the colored round cornered boxes.



Git keeps all of these objects in the folder .git/objects. This is Git’s object database. Each object, regardless of type, is stored as a file, using its SHA-1 checksum as the filename (sort of). But, instead of storing all objects in a single folder, they are split up using the first two characters of their ID as a directory name, resulting in an object database that looks something like the following.

$ find .git/objects

.git/objects

.git/objects/00

.git/objects/00/11f080776acf2d04fb99b0d5c70f85747420a9

.git/objects/01

.git/objects/01/9da3ea8f032c4ebf7825cc13b5eeecc7cf017d

.git/objects/01/c3abfb09d4c4b2b306de4b20188574d4e02914

.git/objects/02

.git/objects/02/2d0352de4df1478f1f6571d0cf52ff22611f9f

.git/objects/03

.git/objects/03/1d9f1c82db42c05df688aed50bcea31bf7554b

For example, an object with the following ID:

022d0352de4df1478f1f6571d0cf52ff22611f9f

is stored in a folder called 02, using the remaining characters (2d0352...) as a filename.

This highlights once again the simplicity of Git: no metadata, no internal databases, or useless complexity, but simple files and folders are enough to make it possible to manage any repository.

Git is amazingly smart and simple: to be quicker while searching through the filesystem, Git creates a set of folders where the name is two characters long, and those two characters represent the first two characters of a hash code; inside those folders, Git writes all the objects using as a name the other 38 characters of the hash, regardless of the kind of Git object. Git compresses them using the zlib library to reserve space on your disk.

If the sha of your object is ab04d884140f7b0cf8bbf86d6883869f16a46f65, then the file will be stored in the following path:

.git/objects/ab/04d884140f7b0cf8bbf86d6883869f16a46f65

It pulls the first two characters off and uses that as the subdirectory, so that there are never too many objects in one directory. The actual file name is the remaining 38 characters.

**Git calculates the hash on the content of the file, not in the file itself.** This teaches us an important lesson: if you have two different files with the same content, even if they have different names and paths, in Git you will end up having only one blob.

This is why we use the git cat-file –p command, which decompresses them on the fly for us

### Git references

Another important directory are is .refs, where Git stores all of its references. .git stores:

* **References**, which are pointers to a single object (usually a commit or tag object).
* Branches and tags point to a commit object and the HEAD object points to the **branch** that is currently checked out. So, for every commit, the full tree state and snapshot are identified by the root tree.

In Git, **a branch is nothing more than a label**, a *mobile label* placed on a commit

Branches, remote-tracking branches, and tags are all references to commits. All references are named with a slash-separated path name starting with "refs"; the names we've been using so far are actually shorthand:

- The branch "test" is short for "refs/heads/test".

- The tag "v2.6.18" is short for "refs/tags/v2.6.18".

- "origin/master" is short for "refs/remotes/origin/master

So, branches are nothing but labels that are on the tip commit, the last one. This commit, our leaf, must always be identified by a label

Every time we make a commit to a branch, the **reference** that identifies that branch will move accordingly to always stay associated with the tip commit

In fact, every leaf on a Git branch has to be labeled with a meaningful name to allow us to reach it and then move around, go back, merge, rebase, or discard some commits when needed.

References

In addition to the Git objects, which are immutable – that is, they cannot ever be changed, there are references also stored in Git. Unlike the objects, references can constantly change. They are simple pointers to a particular commit, something like a tag, but eas­ily moveable.

A branch in Git is nothing more than a file in the .git/refs/heads/ directory that con­tains the SHA-1 of the most recent commit of that branch

In fact, in Git the act of creating a new branch is simply writing a file in the .git/refs/heads directory that has the SHA-1 of the last commit for that branch.

How does Git actually retrieve these objects in practice?

Well, it gets the initial SHA-1 of the starting commit object by looking in the .git/refs directory for the branch, tag or remote you specify. Then it tra­verses the objects by walking the trees one by one, checking out the blobs under the names listed.

In fact, in Git the act of creating a new branch is simply writing a file in the .git/refs/heads directory that has the SHA-1 of the last commit for that branch.

Switching to that branch simply means having Git make your work­ing directory look like the tree that SHA-1 points to and updating the HEAD file so each commit from that point on moves that branch pointer forward (

The HEAD in Git is the pointer to the current branch reference, which is in turn a pointer to the last commit you made or the last commit that was checked out into your working directory. That also means it will be the parent of the next commit you do. It's generally simplest to think of it as **HEAD is the snapshot of your last commit**.

**$ ls –al**

**$ ls -al .git/**

**$ ls -al .git/objects**

**$ ls -al .git/objects/63**

Therefore, any content in **git** can be looked up by it's hash:

git cat-file -p 4bb6f98

HEAD is a special ref. It always points to the current object. You can see where it's currently pointing by checking the .git**/**HEAD file. Normally, HEAD points to another ref:

$cat .git**/**HEAD

ref: refs**/**heads**/**mainline

A ref is essentially a pointer. It's a name that points to an object. For example,

"master" --**>** 1a410e...

They are stored in `.git/refs/heads/ in plain text files.

$ **cat** .git**/**refs**/**heads**/**mainline

4bb6f98a223abc9345a0cef9200562333

Now, it's possible to navigate git purely by jumping around to different objects directly by their hashes. But this would be terribly inconvenient. A ref gives you a convenient name to refer to objects by. It's much easier to ask git to go to a specific place by name rather than by hash

$ cat .git/HEAD

ref: refs/heads/master

$ cat .git/refs/heads/master

e9a570524b63d2a2b3a7c3325acf5b89bbeb131e

$ git cat-file -p e9a570524b63d2a2b3a7c3325acf5b89bbeb131e

tree cfda3bf379e4f8dba8717dee55aab78aef7f4daf

author Scott Chacon 1301511835 -0700

committer Scott Chacon 1301511835 -0700

initial commit

$ git ls-tree -r cfda3bf379e4f8dba8717dee55aab78aef7f4daf

100644 blob a906cb2a4a904a152... README

100644 blob 8f94139338f9404f2... Rakefile

040000 tree 99f1a6d12cb4b6f19... lib

\*\*\*\*\*

* **repository** — A **repository** is a collection of commits, each of which is an archive of what the project’s working tree looked like at a past date, whether on your machine or someone else’s. It also defines HEAD (see below), which identifies the branch or commit the current working tree stemmed from. Lastly, it contains a set of branches and tags, to identify certain commits by name
* **the index** — Unlike other, similar tools you may have used, Git does not commit changes directly from the working tree into the repository. Instead, changes are first registered in something called **the index**. Think of it as a way of “confirming” your changes, one by one, before doing a commit (which records all your approved changes at once). Some find it helpful to call it instead as the “staging area”, instead of the index.

 **working tree** — A **working tree** is any directory on your filesystem which has a *repository* associated with it (typically indicated by the presence of a sub-directory within it named .git.). It includes all the files and sub-directories in that directory.

 **commit** — A **commit** is a snapshot of your working tree at some point in time. The state of HEAD (see below) at the time your commit is made becomes that commit’s parent. This is what creates the notion of a “revision history”.

 **branch** — A **branch** is just a name for a commit (and much more will be said about commits in a moment), also called a reference. It’s the parentage of a commit which defines its history, and thus the typical notion of a “branch of development”.

 **tag** — A **tag** is also a name for a commit, similar to a *branch*, except that it always names the same commit, and can have its own description text.

 **master** — The mainline of development in most repositories is done on a branch called “\*\*master\*\*”. Although this is a typical default, it is in no way special.

 **HEAD** — **HEAD** is used by your repository to define what is currently checked out:

* If you checkout a branch, HEAD symbolically refers to that branch, indicating that the branch name should be updated after the next commit operation.
* If you checkout a specific commit, HEAD refers to that commit only. This is referred to as a detached *HEAD*, and occurs, for example, if you check out a tag name.

$ git cat-file

$ git ls-files

the plumbing command, *cat-file*. You use two options here:

-t = type—shows the type of the object

-p = pretty—prints information about the object

## The Git promotion model

The following diagram describes the tree stages and the commands used to move between the stages. Git manages and manipulates three stages in its normal operation

A Git *repository* is the local collection of all the files related to a particular Git version control system and contains a .git subdirectory in its root. Git keeps track of the state of the files in the repository’s directory on disk. Git repositories store all their data on your local machine. Making commits, viewing history, and requesting differences between commits are all local operations that don’t require a network connection. This makes all these operations much faster in Git than with centralized version control systems such as Subversion.

A **repository** is a container for your entire project; every file or subfolder within it belongs to that repository, in a consistent manner. Physically, a repository is nothing other than a folder that contains a special .git folder, the folder where the magic happens





The distinction between the working directory, the staged snapshot, and committed snapshots is at the core of Git version control. Most Git commands operate on one of the three main components of a Git repository:

The Git model provides a local environment where you can work with a local copy of a server-side repository (this server-side repository is known as the *remote* in Git terminology). This copy resides within your workspace.

### The working directory

Starting at the bottom is the working directory where content is created, edited, deleted, and so on. Any new content must exist here before it can be put into (tracked by) Git. Files in this directory are often removed or replaced by Git as you switch branches. The working directory is simply a temporary checkout place where you can modify the files until your next commit. All subdirectories are considered part of the working directory’s scope, unless Git is specifically told to ignore them via a .gitignore file or they are part of a Git *submodule*.

This is where the content of files are placed into actual files on your filesystem so they're easily edited by you. **The Working Directory is your scratch space, used to easily modify file content.**

When you checkout a branch, it changes **HEAD** to point to the new commit, populates your **Index** with the snapshot of that commit, then checks out the contents of the files in your **Index** into your **Working Directory**.

The *checkout* command is used to retrieve content (as flat files) from the local repository into the working directory. This is usually done by supplying a branch name and telling Git to get the latest copy of content from that branch. Checkout also tells Git to switch the branch that you are currently working with.

Your working directory is temporary – everything is stored permanently in your git repository. Your working directory is a just a copy of a tree so you can edit it and commit changes

### The staging area

The staging area is an intermediate level between the working directory and the local repository. The staging area is a file, generally contained in your .git directory, that stores information about what will go into your next commit. When a file is moved to the staging area, the SHA-1 hash of the file is created and the blob object is written to Git's database. The staging area is one of the concepts in Git that many new users have difficulty understanding and appreciating. At first glance, it may seem like an unnecessary intermediate level that gets in the way of trying to promote content from the working directory to the local repository. In fact, it plays a significant role in several parts of Git’s functionality.

The easiest way to see what is in the index is with the git-status command. When you run git status, you can see which files are staged (currently in your index), which are modified but not yet staged, and which are completely untracked.

It's simplest to think of it as the Index is the snapshot of your next commit.

Git populates it with a list of all the file contents that were last checked out into your working directory and what they looked like when they were originally checked out

$ git ls-files -s

100644 a906cb2a4a904a152e80877d4088654daad0c859 0 README

100644 8f94139338f9404f26296befa88755fc2598c289 0 Rakefile

100644 47c6340d6459e05787f644c2447d2595f5d3a54b 0 lib/simplegit.rb

Again, here we’re using git ls-files, which is more of a behind the scenes command that shows you what your index currently looks like

What’s the Point of the Staging Area?

As its name implies, the staging area provides a place to *stage* changes before they are committed (promoted) into the local repository. The staging area can hold any set of content that has been promoted from the working directory and is a candidate for going into the local repository—from a single file to all of the eligible files. The staging area provides a place to collect or assemble individual changes into the set of things that will be committed. It allows finer-grained control over the set of things that make up a change. Now let’s look at the common use cases for it.

However, in a case where there are merge conflicts that Git cannot automatically resolve, Git puts those files in your working directory for you to fix, and stages any files that merged cleanly. What it is doing is starting to create a set of merged content to be committed once everything is resolved.

There is another side benefit of this arrangement. After the merge has been attempted, if there are conflicts, the merged files are grouped together in the staging area.

One other area where the staging operation is required is when you need to complete a merge operation that had conflicts. As discussed in the previous section, Git stages files that merged successfully. In order to complete the merge, files that have conflicts manually resolved must be staged. This creates a complete set of content to be committed to complete the merge operation.

The primary function of the git add command, is to promote pending changes in the working directory, to the git staging area. The staging area is one of Git's more unique features, and it can take some time to wrap your head around it if you’re coming from an SVN (or even a Mercurial) background. It helps to think of it as a buffer between the working directory and the project history. The staging area is considered one of the ["three trees" of Git](https://www.atlassian.com/git/tutorials/undoing-changes/git-reset), along with, the working directory, and the commit history.

The index was called the cache for a while, because that’s largely what it does. It is a staging area for changes that are made to files or trees that are not committed to your repository yet. It acts as sort of a middle ground between your working directory and your repository. When you run git commit, the resulting tree and commit object will be built based on the contents of the index.

Now that you *hopefully* understand what Git is designed to do at a fundamental level – how it tracks and stores content, how it stores branches and merges and tracks remote copies of the repository,

### The local repository

The .git is the directory that stores all Git's history and meta information for your project - including all of the objects (commits, trees, blobs, tags), all of the pointers to where different branches are and more. There is only one Git Directory per project (as opposed to one per subdirectory like with SVN or CVS), and that directory is (by default, though not necessarily) '.git' in the root of your project. If you look at the contents of that directory, you can see all of your important files:

The combination of the working directory, staging area, and local repository make up your local environment. These are the parts of the Git system that exist on your local machine—actually, within a special subdirectory of the root (top-level) directory of your working directory. This local environment exists for users to create and update content and get it in the form they want before making it available or visible to others, in the remote repository.

When you initialize a Git repository, either by cloning an existing one or creating a new one, the first thing Git does is create a Git directory. This is the directory that stores all the object data, tags, branches, hooks and more. Everything that Git permanently stores goes in this single directory. When you clone someone else’s reposi­tory, it basically just copies the contents of this directory to your computer.

When you run git init to initialize your repository, the Git directory is by default installed in the directory you are currently in as .git. The Git directory for our little project looks something like this:

For now, let’s go over some of the more important contents of this directory.

* .git/config

This is the main Git configuration file. It keeps your project specific Git options, such as your remotes, push configurations, tracking branches and more.

* .git/index

This is the default location of the index file for your Git project.

* .git/objects/

This is the main directory that holds the data of your Git objects – that is, all the contents of the files you have ever checked in, plus your commit, tree and tag objects.

The files are stored by their SHA-1 values. The first two characters make up the subdirectory and the last 38 is the filename

* .git/refs/

This directory normally has three subdirectories in it – *heads*, *remotes* and *tags*. Each of these directories will hold files that correspond to your local branches, remote branches and tags, respectively

* ..git/HEAD

This file holds a reference to the branch you are currently on. This tells Git what to use as the parent of your next commit

* .git/hooks

Contains shell scripts that are invoked after the git command

### The remote repository

The remote repository is a separate Git repository intended to collect and host content pushed to it from one or more local repositories. Like the Public level in the dev-test-prod model, its main purpose is to be a place to share and access content from multiple users. There are various forms of hosting and protocols

When moving content from the remote repository to the local environment, there are several ways the local repository and the working directory can receive content from the remote repository.

The *clone* command is used to create a new local environment from an existing remote repository. Essentially, it makes a local copy of the specified remote repository onto the local disk and checks out a flat copy of the files from a branch (typically master, although this is configurable) into the working directory.

The *fetch* command is used to update the local repository from the remote repository. More specifically, it is updating reference copies of the remote branches (*reference branches*) that are maintained in the local repository.

The result of the git init command is the creation of a .git folder, where Git stores all the files it needs to manage our repository

**$ git add .**

With this trick (the dot after the git add command), you can add all the new or modified files in one shot.

So, we can move this grocery folder wherever we want, and no data will be lost. Another important thing to highlight is that we don't need any server: we can create a repository locally and work with it whenever we want, even with no LAN or internet Connection

## Commands in Git

$ git <git-options> <command> <command-options> <operands>

$ git

$ git help glossary

$ git help –a //list of over 150 commands

$ git help –g //list of common guides

$ git help config

$ git config –h

$ git config --help

Git has a myriad of commands, some of which are practically never used by the average user; as by example, the previous git cat-file. These commands are called *plumbing commands*, while others, such as git add, git commit, are among the so-called *porcelain commands*.

The plumbing commands function at a lower level and are not expected to be used by the average user. These commands are typically targeted at extracting or modifying content and information more directly from the repository. An example would be the git cat-file or git ls-files commands that provide a way to look at the contents of a file or directory within the repository if you know how to reference those elements.

The porcelain commands are intended to be user-facing, more commonly used, and more convenient.

The porcelain commands are based on the plumbing commands. They aggregate the functionality of plumbing commands and certain options and sequences in order to make things simpler for the typical Git user

The general form of commands is a as follows:

$ git <command> <command-options> <operands>

|  |  |  |
| --- | --- | --- |
|  | Description | Examples |
| <command> | Git command to execute | $ git push |
| <command-options> | Options to the specified command | $ git commit -m “comment” |
| <operands> | Items for the command to operate on | $ git add \*.c |

The primary reason to specify both commit references and paths would be to select certain paths that are part of the tree associated with the commit. Because Git operates at the granularity of a tree, you may not always want to do the operation against all items in the tree. To indicate that the operation should only be done against certain files or paths in the scope of the snapshot, you need to add specific filenames or paths.

When both types are specified, if there is a possibility of Git not being able to tell the difference between a commit | branch | tag and one or more of the filenames or paths, then you can separate the two types using the special separation symbol “--”.Normally, this won’t be needed if a commit is expressed as a SHA1 value, but it may be needed if branch or tag names could be mistaken as names for files or paths.

As an example, the command git checkout a1b2c3d4 file1.txt might be clear enough, but git checkout my-tag-name -- my-file-name could be ambiguous enough when parsed to require the “--” separator symbol.

|  |  |
| --- | --- |
| Command | Purpose |
| add | Add files contents to the index |
| branch | List, create, or delete branches |
| checkout | Switch branches or restore working tree files. |
| cherry | Find commits yet to be applied to upstream (branch on the remote). |
| cherry-pick | Apply the changes introduced by some existing commits. |
| clone | Clone a repository into a new directory. |
| commit | Record changes to the repository |
| config | Get and set repository or global options. |
| diff | Show changes between commits, commits and working tree, and so on |
| fetch | Download objects and refs from another repository |
| grep | Print lines matching a pattern |
| help | Display help information |
| log | Show commit logs. |
| merge | Join two or more development histories together |
| mv | Move or rename a file, directory, or symlink. |
| pull | Fetch from, or integrate with, another repository or a local branch |
| push | Update remote refs along with associated objects. |
| rebase | Forward-port local commits to the updated upstream head |
| rerere | Reuse recorded resolution for merged conflicts. |
| reset | Reset current HEAD to the specified state. |
| revert | Revert some existing commits. |
| rm | Remove files from the working tree and from the index. |
| show | Show various types of objects. |
| status | Show the working tree status. |
| submodule | Initialize, update, or inspect submodules. |
| subtree | Merge subtrees and split repositories into subtrees. |
| tag | Create, list, delete, or verify a tagged object. |
| worktree | Manage multiple working tree |

Porcelain Commands in Git

Table shows the same categorization for the plumbing commands. These commands have names that indicate an action and an object to operate against as opposed to the simpler naming of the porcelain commands.

|  |  |
| --- | --- |
| cat-file | Provide content or type and size information for repository objects |
| commit-tree | Create a new commit object. |
| count-objects | Count an unpacked number of objects and their disk consumption. |
| diff-index | Compare a tree to the working tree or index. |
| for-each-ref | Output information on each ref. |
| hash-object | Compute object ID and optionally create a blob from a file. |
| ls-files | Show information about files in the index and the working tree. |
| merge-base | Find as good common ancestors as possible for a merge. |
| read-tree | Read tree information into the index. |
| rev-list | List commit objects in reverse chronological order. |
| rev-parse | Pick out and massage parameters. |
| show-ref | List references in a local repository. |
| symbolic-ref | Read, modify, and delete symbolic refs. |
| update-index | Register file contents in the working tree to the index. |
| update-ref | Update the object name stored in a ref safely. |
| verify-pack | Validate packed Git archive files. |
| write-tree | Create a tree object from the current index. |

Plumbing commands

Arguments supplied to Git commands can be abbreviated as a single letter or spelled out as words. One important note here is that if the argument is spelled out, you must precede it with two hyphens, as in --global. If the argument is abbreviated, only one hyphen is required, as in -a. Abbreviated arguments may be passed together, as in -am instead of -a -m. When arguments are combined in this way, the ordering is important. If the first argument requires a value, then the second argument may be taken as the required value instead of an additional argument.

## Basic Git

|  |  |
| --- | --- |
| git config | Sets configuration values for things like your user name, email, and gpg key, your preferred diff algorithm, file formats to use, proxies, remotes and tons of other stuff. For a full list, see the git-config docs |
| git init | Initializes a git repository – creates the initial ‘.git’ directory in a new or existing project. |
| git clone | Copies a Git repository from another place and adds the original location as a remote you can fetch from again and possibly push to if you have permission. |
| git add | Adds changes in files in your working directory to your index, or staging area |
| git rm | Removes files from your index and your working directory so they will stopped being tracked |
| git commit | Takes all of the changes staged in the index (that have been ‘git add’ed), creates a new commit object pointing to it, and advances the branch to point to that new commit. |
| git status | Shows you the status of files in your index versus your working directory. It will list out files that are untracked (only in your working directory), modified (tracked but not yet updated in your index), and staged (added to your index and ready for committing). |
| git branch | Lists existing branches, including remote branches if ‘-a’ is provided. Creates a new branch if a branch name is provided. Branches can also be created with ‘-b’ option to ‘git checkout’. |
| git checkout | Checks out a different branch – makes your working directory look like the tree of the commit that branch points to and updates your HEAD to point to this branch now, so your next commit will modify it. |
| git merge | Merges one or more branches into your current branch and auto­matically creates a new commit if there are no conflicts. |
| git reset | Resets your index and working directory to the state of your last commit, in the event that something screwed up and you just want to go back. |
| git rebase | An alternative to merge that rewrites your commit history to move commits since you branched off to apply to the current head instead. A bit dangerous as it discards existing commit objects. |
| git stash | Temporarily saves changes that you don’t want to commit immedi­ately for later. Can re-apply the saved changes at any time |
| git tag | Tags a specific commit with a simple, human readable handle that never moves. |
| git fetch | Fetches all the objects that a remote version of your repository has that you do not yet so you can merge them into yours or simply inspect them. |
| git pull | Runs a ‘git fetch’ then a ‘git merge’. |
| git push | Pushes all the objects that you have that a remote version does not yet have to that repository and advances its branches. |
| git remote | Lists all the remote versions of your repository, or can be used to add and delete them. |

## Inspecting Repositories

|  |  |
| --- | --- |
| git log | Shows a listing of commits on a branch or involving a specific file and optionally details about what changed between it and its par­ents. |
| git show | Shows information about a git object, normally used to view commit information. |
| git ls-tree | Shows a tree object, including the mode and name of each node and the SHA-1 value of the blob or tree that it points to. Can also be run recursively to see all subtrees as well. |
| git cat-file | Used to view the type of an object if you only have the SHA-1 value, or used to redirect contents of files or view raw information about any object. |
| git grep | Lets you search through your trees of content for words and phrases without having to actually check them out. |
| git diff | Generates patch files or statistics of differences between paths or files in your git repository, or your index or your working directory. |
| gitk | Graphical Tcl/Tk based interface to a local Git repository |
| git instaweb | Wrapper script to quickly run a web server with an interface into your repository and automatically directs a web browser to it. |

## Exploring the Object Database

For Windows users, Git installation will install a special command shell called *Git Bash*. To test your installation, open a new command prompt and run

$ git --version.

**[15] ~/grocery (master)**

**$ git cat-file -p a57d7**

**tree a31c31cb8d7cc16eeae1d2c15e61ed7382cebf40**

this plumbing command lets you peek into the Git objects; with the -p option (which means *pretty-print* here), we ask Git to show an easier way to read what the contents of the object are.

**$ git cat-file -p 637a0**

**banana**

You can see the contents of any commit like this:

$ **git cat-file** commit 5bac93

if you have the hash of a blob, you can look at it's contents.

$ git cat-file -t 54196cc2

$ git cat-file -s 54196cc2

$ git cat-file -p 54196cc2

You can examine the contents of any tree using ls-tree

$ git ls-tree 92b8b694

All of these objects are stored under their SHA1 names inside the git directory and the contents of these files is just the compressed data plus a header identifying their length and their type. The type is either a blob, a tree, a commit, or a tag.

$ find .git/objects/

The simplest commit to find is the HEAD commit, which we can find from .git/HEAD:

$ cat .git/HEAD

ref: refs/heads/master

$ cat .git/refs/heads/master

c4d59f390b9cfd4318117afde11d601c1085f241

$ git cat-file -t c4d59f39

commit

We can list all the heads in this repository with linkgit:git-show-ref

$ git show-ref --heads

$ git show-ref --tags

Computes the object ID value for an object with specified type with the contents of the named file

$ git hash-object

The ever-versatile git-show command can also be used to examine tree objects, but : git-ls-tree will give you more details

The "commit" object links a physical state of a tree with a description of how we got there and why. You can use the --pretty=raw option to git-show or git-log to examine your favorite commit.

A commit is usually created by git-commit, which creates a commit whose parent is normally the current HEAD, and whose tree is taken from the content currently stored in the index.

The Git command git cat-file -p will print the object given as an input. Normally, it is not used in everyday Git commands, but it is quite useful to investigate how it ties the objects together

$ git cat-file -p HEAD

We can now see the commit object, consisting of the root tree (tree), the parent commit object's ID (parent), the author and timestamp information (author), the committer and timestamp information (committer), and the commit message.

A tag object contains an object name (called simply 'object'), object type, tag name, the name of the person ("tagger") who created the tag, and a message, as can be seen using git-cat-file.

$ git cat-file tag v1.5.0

git-tag can also be used to create "lightweight tags", which are not tag objects at all, but just simple references whose names begin with "refs/tags/"

$ ls -al

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

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

**$ git checkout -**

**Switched to branch 'berries**

New trick: using the dash (-), you actually are saying to Git: "*Move me to the branch I was before switching*"; and Git obeys, moving us to the berries branch

The index is a binary file (generally kept in .git/index) containing a sorted list of path names, each with permissions and the SHA1 of a blob object; linkgit:git-ls-files[1] can show you the contents of the index:

$ git ls-files --stage

The **git commit** command does a few things:

1. Create blobs and trees to represent your project directory - stored in .git**/**objects

2. Creates a new commit object with your author information, commit message, and the root **tree** from step 1 - also stored in .git**/**objects

3. Updates the HEAD ref in .git**/**HEAD to the hash of the newly-created commit

We can ask git about particular objects with the cat-file command. Note that you can shorten the shas to only a few characters to save yourself typing all 40 hex digits:

I made a commit without first making git add; the *trick* is in the -a (--add) option added to the git commit command, which means *add to this commit all the modified files that I have already committed at least one time before*. In our case, this option allowed us to go faster and skip the git add command.

**$ ls -al .git/**

**$ ls -al .git/refs**

**$ ls -al .git/refs/heads**

**$ cat .git/refs/heads/master**

**0e8b5cf1c1b44110dd36dea5ce0ae29ce22ad4b8**

Git manages all this articulated reference system... with a trivial text file! It contains the hash of the last commit made on the branch

As branches are, HEAD is a **reference**. It represents a pointer to the place on where we are right now, nothing more, nothing less. In practice instead, it is just another plain text file:

**$ cat .git/HEAD**

**ref: refs/heads/berries**

The difference between the HEAD file and branches text file is that the HEAD file usually refers to a branch, and not directly to a commit as branches do. The ref: part is the convention Git uses internally to declare a pointer to another branch, while refs/heads/berries is of course the relative path to the berries branch text file

*In Subversion, we usually have different folders for each different branch.* When you switch a branch, Git goes to the commit the branch is pointing to, and following the parent relationship and analyzing trees and blobs, rebuilds the content on the **working directory** accordingly, getting hold of that files and folders

This gives us an object ID, but before we can inspect items in the object database, we need to know what type of object it is. Again, we can use the -t flag:

$ git cat-file -t 022d0352de4

tree

Of course, change the object ID to an object from your database (don’t forget to combine the folder name with the filename to get the full ID). This will output the type of commit, which we can then pass to a normal call to git cat-file.

git cat-file blob 7a52bb8

My object was a blob, but yours may be different. If it’s a tree, remember to use git ls-tree to turn that ugly binary data into a pretty directory listing.

$ git ls-tree 022d0352de4

100644 blob 1d09ca3ac33e045ccde753b47f81a9e980c90774 .gitconfig-template

100644 blob 139597f9cb07c5d48bed18984ec4747f4b4f3438 .gitignore

100644 blob 812e4df6163374ffb1ffbd1dac2cf8ec5460684e Basic-Tutorials-master.zip

040000 tree b9e065e8380804fcc424dc08128b3163ecf3ae6b Chapters

100644 blob 42ca1cef8e65effe2ad5bae228bcfc406c058f85 Links.txt

100644 blob 3aa637263b63eaaa57a2f34419374cfe7fc0701a Separating Collated Code with Branching Strategies.docx

040000 tree ba6aabea140dfdd02d0b4d84844dbe95410a78dd book-svg

100644 blob 2c6500fd68bb673ba4a5f571ced1e9cbdee31bf9 borderl.txt

100644 blob 6f6305ebd02adcfc9967d396246d5b0bc0183e37 instaLL.rtf

040000 tree e950201e1a88e43226e5d497656125936a514286 ppt

100644 blob 87701586a37307d58549e39dcdfc70bd8db51936 script.sh

100644 blob 0fe633e30c461e9e5e08d545fe06d909243955dd txt1.txt

Any file is compressed and transformed into a blob before archiving it into a Git repository. Each file is marked with a *hash*; this hash uniquely identifies the file within our repository, and it is thanks to this ID that Git can then retrieve it when needed, and detect any changes when the same file is altered (files with different content will have different hashes).

SHA-1 hashes are unique

**$ echo "banana" | git hash-object --stdin**

**637a09b86af61897fb72f26bfb874f2ae726db82**

The git hash-object command is the plumbing command to calculate the hash of any object; in this example, we used the --stdin option to pass as a command argument the result of the preceding command, echo "banana"; in a few words, we calculated the hash of the string "banana",

You can initialize a Git repository anywhere with the git init command. Take a look inside the .git folder to get a glimpse of what a repository looks like.

$ git init

Initialized empty Git repository in C:/temp/demo/.git/

$ ls -la .git

total 11

drwxr-xr-x 1 asaki 1049089 0 Jun 22 13:50 ./

drwxr-xr-x 1 asaki 1049089 0 Jun 22 13:49 ../

-rw-r--r-- 1 asaki 1049089 130 Jun 22 13:50 config

-rw-r--r-- 1 asaki 1049089 73 Jun 22 13:49 description

-rw-r--r-- 1 asaki 1049089 23 Jun 22 13:49 HEAD

drwxr-xr-x 1 asaki 1049089 0 Jun 22 13:49 hooks/

drwxr-xr-x 1 asaki 1049089 0 Jun 22 13:49 info/

drwxr-xr-x 1 asaki 1049089 0 Jun 22 13:50 objects/

drwxr-xr-x 1 asaki 1049089 0 Jun 22 13:49 refs/

The special HEAD pointer that refers to the branch/commit currently being checked out

* The tree object:

There are many ways to see the objects in the Git database. The git ls-tree command can easily show the content of trees and subtrees, and git show can show the Git objects, but in a different way.

We can also specify that we want the tree object from the commit pointed to by HEAD by specifying:

$ git cat-file -p HEAD^{tree}

The special notation HEAD^{tree} means that from the reference given, HEAD recursively dereferences the object at the reference until a tree object is found. The first tree object is the root tree object found from the commit pointed to by the master branch, which is pointed to by HEAD.

A generic form of the notation is <rev>^<type>, and will return the first object of <type>, searching recursively from <rev>.

* The blob object:

The branch object

we can take a look at the branch inside the .git folder where the whole Git repository is stored. If we open the text file .git/refs/heads/master, we can actually see the commit ID that the master branch points to. We can do this using cat, as follows:

**$ cat .git/refs/heads/master**

**13dcada077e446d3a05ea9cdbc8ecc261a94e42d**

We can also see that HEAD is pointing to the active branch by using cat with the .git/HEAD file:

**$ cat .git/HEAD**

The branch object is simply a pointer to a commit, identified by its SHA-1 hash.

The tag object

There are three different kinds of tag: a lightweight (just a label) tag, an annotated tag, and a signed tag. In the example repository, there are two annotated tags:

**$ git cat-file -p v1.0**

As you can see, the tag consists of an object—which, in this case, is the latest commit on the master branch—the object's type (commits, blobs, and trees can be tagged), the tag name, the tagger and timestamp, and finally the tag message.

There are many ways to see the objects in the Git database. The git ls-tree command can easily show the content of trees and subtrees, and git show can show the Git objects, but in a different way.

***The Treeish***

Besides branch heads, there are a number of shorthand ways to refer to particular objects in the Git data store. These are often referred to as a *treeish*. Any Git command that takes an object – be it a commit, tree or blob – as an argument can take one of these shorthand versions as well.

* Full SHA-1

dae86e1950b1277e545cee180551750029cfe735

* PARTIAL SHA-1

dae86e

the full SHA-1 can be referenced fine with the first 6 or 7 characters. Git is smart enough to figure out a partial SHA-1 as long as it’s unique.

* Branch or tag name

Anything in *.git/refs/heads* or *.git/refs/tags* can be used to refer to the commit it points to.

* date spec

master@{yesterday}

master@{1 month ago}

* ordinal spec

master@{5}

This indicates the 5th prior value of the master branch. Like the *Date Spec*, this depends on special files in the *.git/log* directory that are written during commits, and is specific to *your* repository

* Carrot parent

dae86e^N

this refers to the Nth parent of that commit. Only really helpful for commits that merged two or more commits

* Tilde spec

dae86e~N

refers to the Nth generation grandparent of that commit

dae86e~5 ⬄ dae86e^^^^^

* tree pointer

e65s46^{tree}

This points to the tree of that commit. Any time you add a ^{tree} to any commit-ish, it resolves to its tree.



* Tree pointer

dae86e^{tree}

This points to th tree of that commit

blob spec

master:/path/to/file

This is very helpful for referring to a blob under a particular commit or tree.

# Git Configuration

## Initialize the Git Repository

$ mkdir myrepo

$ cd myrepo

$ git init

Git is designed to be as unobtrusive as possible. Notice that there is now a .git directory in myrepo that stores all the tracking data for our repository. The .git folder is the only difference between a Git repository and an ordinary folder, so deleting it will turn your project back into an unversioned collection of files

git init can be run without any arguments to create the local Git repository in the current directory. Under the myrepo repository directory, a .git is created with various files and directories in the current directory.

$ find .git

.git/config //Contains the configuration of the local repository

.git/HEAD //Head pointer

.git/hooks

...

.git/objects // Object storage

.git/objects/info

.git/objects/pack

.git/refs

.git/refs/heads //Contains the branch pointers

.git/refs/tags //Contains the tag pointers

## Configure git

Git comes with a long list of configurations options from your name to your favorite merge tool. You can set options with the git config or by manually editing a file called .gitconfig in your home directory.

In addition to configuring a remote repo URL, you may also need to set global Git configuration options such as username, or email. The git config command lets you configure your Git installation (or an individual repository) from the command line. This command can define everything from user info, to preferences, to the behavior of a repository. Several common configuration options are listed below.

Git stores configuration options in three separate files, which lets you scope options to individual repositories (local), user (Global), or the entire system (system):

* Local: <repo>/.git/config – Repository-specific settings.
* Global: /.gitconfig – User-specific settings. This is where options set with the --global flag are stored.
* System: $(prefix)/etc/gitconfig – System-wide settings.

Once you’ve installed Git, the first thing you need to do is to tell Git your name and email (particularly before creating any commits). Rather than usernames, Git uses a name and an email address to identify the author of a commit. We can do this with the git config command:

$ git config --global user.name "Your Name"

$ git config --global user.email [your.email@example.com](mailto:your.email@example.com)

The --global option tells Git to use this configuration as a default for all of your repositories. Omitting it lets you specify different user information for individual repositories, which will come in handy later on.

Git’s command line relies on a text editor for most of its input. You can forece Git to use your editor of choice with the core.editor option

$ git config --global core.editor notepad

Git supports aliasing commands

$ git config --global alias.st status

To tell Git to ignore certain files (meaning not to track them), you just need to list them in a *Git ignore file*. This is a text file named *.gitignore* that is placed at the root (top level directory) of the local environment

This is because in Git, every modification you make in a repository has to be signed with the name and email of the author. So, before doing anything else, we have to tell Git this information.

You can configure Git to use your own preferred editor, but if you don't do it, this is what you have to deal with. Vim is powerful, but for newcomers, it can be a pain to use. It has a strange way of dealing with text. To start typing, you have to press *I* for inserting text, as shown in the following

Once you have typed your commit message, you can press *Esc* to get out of editing mode. Then, you can type the :w command to write changes and the :q command to quit. You can also type the command in pairs as :wq,

So, I prefer setting up usernames and emails per repository; in Git, you can set up your config variables at three levels: *repository* (with the --local option, the default one), *user* (with the --global option), and *system-wide* (with the --system option).

**$ git config user.name "Ferdinando Santacroce"**

**$ git config user.email** [ferdinando.santacroce@gmail.com](mailto:ferdinando.santacroce@gmail.com)

**==🡺**

Define the author name to be used for all commits in the current repository. Typically, you’ll want to use the --global flag to set configuration options for the current user.

git config --global user.name <name>

Adding the --local option or not passing a config level option at all, will set the user.name for the current local repository.

git config --local user.email <email>

Define the author email to be used for all commits by the current user.

git config --global alias.<alias-name> <git-command>

Create a shortcut for a Git command. This is a powerful utility to create custom shortcuts for commonly used git commands. A simplistic example would be:

git config --global alias.ci commit

This creates a ci command that you can execute as a shortcut to git commit. To learn more about git aliases visit the [git config page](https://www.atlassian.com/git/tutorials/setting-up-a-repository/git-config).

git config --system core.editor <editor>

Define the text editor used by commands like git commit for all users on the current machine. The <editor> argument should be the command that launches the desired editor (e.g., vi). This example introduces the --system option. The --system option will set the configuration for the entire system, meaning all users and repos on a machine

git config --global --edit

Open the global configuration file in a text editor for manual editing.

All configuration options are stored in plaintext files, so the git config command is really just a convenient command-line interface. Typically, you’ll only need to configure a Git installation the first time you start working on a new development machine, and for virtually all cases, you'll want to use the --global flag. One important exception is to override the author email address. You may wish to set your personal email address for personal and open source repositories, and your professional email address for work-related repositories.

Git stores configuration options in three separate files, which lets you scope options to individual repositories, users, or the entire system:

* <repo>/.git/config – Repository-specific settings.
* ~/.gitconfig – User-specific settings. This is where options set with the --global flag are stored.
* $(prefix)/etc/gitconfig – System-wide settings.

When options in these files conflict, local settings override user settings, which override system-wide. If you open any of these files, you’ll see something like the following:

The first thing you’ll want to do after installing Git is tell it your name/email and customize some of the default settings. A typical initial configuration might look something like the following:

Tell Git who you are git config

git --global user.name "John Smith" git config --global user.email john@example.com

Select your favorite text editor

git config --global core.editor vim

This will produce the ~ /.gitconfig file from the previous section

The git config command is a convenience function that is used to set Git configuration values on a global or local project level. These configuration levels correspond to .gitconfig text files. Executing git config will modify a configuration text file.

Overall, git config is a helper tool that provides a shortcut to editing raw git config files on disk.

The most basic use case for git config is to invoke it with a configuration name, which will display the set value at that name. Configuration names are dot delimited strings composed of a 'section' and a 'key' based on their hierarchy.

$ git config user.email

The git config command can accept arguments to specify which configuration level to operate on. The following configuration levels are available:

* **--local**

By default, git config will write to a local level if no configuration option is passed. Local level configuration is applied to the context repository git config gets invoked in. Local configuration values are stored in a file that can be found in the repo's .git directory: .git/config

* **--global**

Global level configuration is user-specific, meaning it is applied to an operating system user. Global configuration values are stored in a file that is located in a user's home directory C:\Users\<username>\.gitconfig on windows

* **--system**

System-level configuration is applied across an entire machine. This covers all users on an operating system and all repos. The system level configuration file lives in a gitconfig file off the system root path. $(prefix)/etc/gitconfig on unix systems. On windows this file can be found at C:\Documents and Settings\All Users\Application Data\Git\config on Windows XP, and in C:\ProgramData\Git\config on Windows Vista and newer.

Thus the order of priority for configuration levels is: local, global, system.

### Writing a value

Expanding on what we already know about git config, let's look at an example in which we write a value:

git config --global user.email "your\_email@example.com"

This example writes the value your\_email@example.com to the configuration name user.email. It uses the --global flag so this value is set for the current operating system user.

### Writing a value

Expanding on what we already know about git config, let's look at an example in which we write a value:

git config --global user.email "your\_email@example.com"

This example writes the value your\_email@example.com to the configuration name user.email. It uses the --global flag so this value is set for the current operating system user.

## git config editor - core.editor

Many Git commands will launch a text editor to prompt for further input. One of the most common use cases for git config is configuring which editor Git should use. Listed below is a table of popular editors and matching git config commands:

## Merge tools

In the event of a merge conflict, Git will launch a "merge tool." By default, Git uses an internal implementation of the common Unix diff program. The internal Git diff is a minimal merge conflict viewer. There are many external third party merge conflict resolutions that can be used instead. For an overview of various merge tools and configuration, see our guide on [tips and tools to resolve conflits with Git](https://developer.atlassian.com/blog/2015/12/tips-tools-to-solve-git-conflicts/).

git config --global merge.tool kdiff3

## Aliases

You may be familiar with the concept of aliases from your operating system command-line; if not, they're custom shortcuts that define which command will expand to longer or combined commands. Aliases save you the time and energy cost of typing frequently used commands. Git provides its own alias system. A common use case for Git aliases is shortening the commit command. Git aliases are stored in Git configuration files. This means you can use the git config command to configure aliases.

git config --global alias.ci commit

This example creates a ci alias for the git commit command. You can then invoke git commit by executing git ci. Aliases can also reference other aliases to create powerful combos.

git config --global alias.amend ci --amend

This example creates an alias amend which composes the ci alias into a new alias that uses --amend flag.

**Formatting & whitespace**

Git has several "whitespace" features that can be configured to highlight whitespace issues when using git diff. The whitespace issues will be highlighted using the configured color color.diff.whitespace

The following features are enabled by default:

* blank-at-eol highlights orphan whitespaces at the line endings
* space-before-tab highlights a space character that appears before a tab character when indenting a line
* blank-at-eof highlights blank lines inserted at the end of a file

The following features are disabled by default

* indent-with-non-tab highlights a line that is indented with spaces instead of tabs
* tab-in-indent highlights an initial tab indent as an error
* trailing-space is shorthand for both blank-at-eol and blank-at-eof
* cr-at-eol highlights a carriage-return at the line endings
* tabwidth=<n> defines how many character positions a tab occupies. The default value is 8. Allowed values are 1-63

## Summary

In this article, we covered the use of the git config command. We discussed how the command is a convince method for editing raw git config files on the filesystem. We looked at basic read and write operations for configuration options. We took a look at common config patterns:

* How to configure the Git editor
* How to override configuration levels
* How to reset configuration defaults

### Git Config

The first thing you're going to want to do is set up your name and email address for Git to use to sign your commits.

$ git config --global user.name "Scott Chacon"

$ git config --global user.email "schacon@gmail.com"

That will set up a file in your home directory which may be used by any of your projects. By default that file is ~/.gitconfig and the contents will look like this:

If you want to override those values for a specific project (to use a work email address, for example), you can run the git config command without the --global option while in that project. This will add a [user] section like the one shown above to the .git/config file in your project's root directory.

At a Windows command prompt enter the commands:  
git config --global diff.tool bc  
git config --global difftool.bc.path "c:/Program Files/Beyond Compare 4/bcomp.exe"

git config --global merge.tool bc  
git config --global mergetool.bc.path "c:/Program Files/Beyond Compare 4/bcomp.exe"

### Changing your Editor

$ git config --global core.editor emacs

### Adding Aliases

$ git config --global alias.last 'cat-file commit HEAD'

$ git last

### Commit Template

$ git config commit.template '/etc/git-commit-template'

### Log Format

$ git config format.pretty oneline

A Git repository can be configured to ignore specific files or directories. This will prevent Git from saving changes to any ignored content. Git has multiple methods of configuration that manage the ignore list. Git ignore configure is discussed in further detail on the [git ignore](https://www.atlassian.com/git/tutorials/saving-changes/gitignore) page.

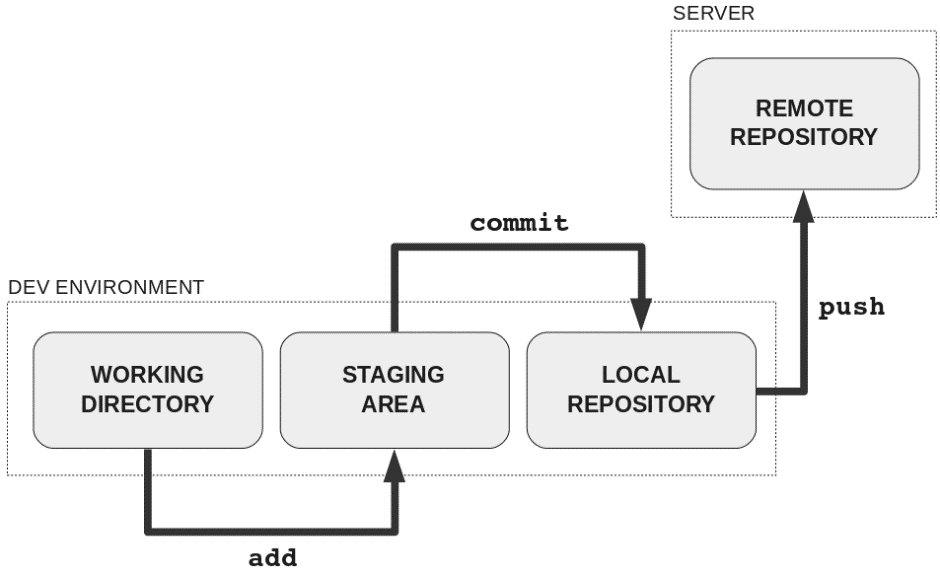
#### Ignoring Files

Untracked files typically fall into two categories. They're either files that have just been added to the project and haven't been committed yet, or they're compiled binaries like .pyc, .obj, .exe, etc. While it's definitely beneficial to include the former in the git status output, the latter can make it hard to see what’s actually going on in your repository.

For this reason, Git lets you completely ignore files by placing paths in a special file called [.gitignore](https://www.atlassian.com/git/tutorials/gitignore). Any files that you'd like to ignore should be included on a separate line, and the \* symbol can be used as a wildcard. For example, adding the following to a .gitignore file in your project root will prevent compiled Python modules from appearing in git status:

\*.pyc

# Committing



cd /path/to/project

echo "test content for git tutorial" >> CommitTest.txt

git add CommitTest.txt

git commit -m "added CommitTest.txt to the repo"

After executing this example, your repo will now have CommitTest.txt added to the history and will track future updates to the file.

This example introduced two additional git commands: add and commit.

## Git Add

The commands: git add, [git status](https://www.atlassian.com/git/tutorials/inspecting-a-repository), and [git commit](https://www.atlassian.com/git/tutorials/saving-changes/git-commit) are all used in combination to save a snapshot of a Git project's current state.

The git add command adds a change in the working directory to the staging area. It tells Git that you want to include updates to a particular file in the next commit. However, git add doesn't really affect the repository in any significant way—changes are not actually recorded until you run [git commit](https://www.atlassian.com/git/tutorials/saving-changes/git-commit).

In conjunction with these commands, you'll also need [git status](https://www.atlassian.com/git/tutorials/inspecting-a-repository) to view the state of the working directory and the staging area.

The git add and [git commit](https://www.atlassian.com/git/tutorials/saving-changes) commands compose the fundamental Git workflow. These are the two commands that every Git user needs to understand. They are the means to record versions of a project into the repository’s history.

In addition to git add and git commit, a third command [git push](https://www.atlassian.com/git/tutorials/syncing) is essential for a complete collaborative Git workflow. git push is utilized to send the committed changes to remote repositories for collaboration. This enables other team members to access a set of saved changes.

Instead of committing all of the changes you've made since the last commit, the stage lets you group related changes into highly focused snapshots before actually committing it to the project history. This means you can make all sorts of edits to unrelated files, then go back and split them up into logical commits by adding related changes to the stage and commit them piece-by-piece. As in any revision control system, it’s important to create atomic commits so that it’s easy to track down bugs and revert changes with minimal impact on the rest of the project.

|  |  |
| --- | --- |
| git add <file> | Stage all changes in <file> for the next commit. |
| git add <directory> | Stage all changes in <directory> for the next commit. |
| git add -p | Begin an interactive staging session that lets you choose portions of a file to add to the next commit. This will present you with a chunk of changes and prompt you for a command. Use y to stage the chunk, n to ignore the chunk, s to split it into smaller chunks, e to manually edit the chunk, and q to exit. |

In review, git add is the first command in a chain of operations that directs Git to "save" a snapshot of the current project state, into the commit history. When used on its own, git add will promote pending changes from the working directory to the staging area. The [git status](https://www.atlassian.com/git/tutorials/inspecting-a-repository) command is used to examine the current state of the repository and can be used to confirm a git add promotion. The [git reset](https://www.atlassian.com/git/tutorials/undoing-changes/git-reset) command is used to undo a git add. The [git commit](https://www.atlassian.com/git/tutorials/saving-changes/git-commit) command is then used to Commit a snapshot of the staging directory to the repositories commit history.

git add adds the file to the index. The index is a list that contains every file that Git has been told to keep track of. It is stored as a file at .git/index.

## Git Status

The git status command displays the state of the working directory and the staging area. It lets you see which changes have been staged, which haven’t, and which files aren’t being tracked by Git.

While the git status output is pretty comprehensive, it’s also quite wordy. Git also has a short status flag so you can see your changes in a more compact way. If you run git status -s or git status --short you get a far more simplified output from the command:

$ git status –s

New files that aren’t tracked have a ?? next to them, new files that have been added to the staging area have an A, modified files have an M and so on. There are two columns to the output - the lefthand column indicates the status of the staging area and the right-hand column indicates the status of the working tree. So for example in that output, the README file is modified in the working directory but not yet staged, while the lib/simplegit.rb file is modified and staged. The Rakefile was modified, staged and then modified again, so there are changes to it that are both staged and unstaged.

To see what you’ve changed but not yet staged, type git diff with no other arguments. That command compares what is in your working directory with what is in your staging area. The result tells you the changes you’ve made that you haven’t yet staged.

If you want to see what you’ve staged that will go into your next commit, you can use git diff --staged. This command compares your staged changes to your last commit:

It’s important to note that git diff by itself doesn’t show all changes made since your last commit — only changes that are still unstaged. If you’ve staged all of your changes, git diff will give you no output.

Now you can use git diff to see what is still unstaged and git diff --cached to see what you’ve staged so far (--staged and --cached are synonyms):

We will continue to use the git diff command in various ways throughout the rest of the book. There is another way to look at these diffs if you prefer a graphical or external diff viewing program instead. If you run git difftool instead of git diff, you can view any of these diffs in software like emerge, vimdiff and many more (including commercial products). Run git difftool --tool-help to see what is available on your system

git status list which files are staged, unstaged, and untracked.

The git status command is a relatively straightforward command. It simply shows you what's been going on with git add and git commit. Status messages also include relevant instructions for staging/unstaging files. Sample output showing the three main categories of a git status call is included below:

It's good practice to check the state of your repository before committing changes so that you don't accidentally commit something you don't mean to.

## Git Commit

The git commit command captures a snapshot of the project's currently staged changes. Committed snapshots can be thought of as “safe” versions of a project—Git will never change them unless you explicitly ask it to. Prior to the execution of git commit, The [git add](https://www.atlassian.com/git/tutorials/saving-changes) command is used to promote or 'stage' changes to the project that will be stored in a commit. These two commands git commit and git add are two of the most frequently used.

In Git, repositories are distributed, Snapshots are committed to the local repository, and this requires absolutely no interaction with other Git repositories. Git commits can later be pushed to arbitrary remote repositories.

Aside from the practical distinctions between SVN and Git, their underlying implementation also follows entirely divergent design philosophies. Whereas SVN tracks differences of a file, Git’s version control model is based on snapshots. For example, a SVN commit consists of a diff compared to the original file added to the repository. Git, on the other hand, records the entire contents of each file in every commit.

This makes many Git operations much faster than SVN, since a particular version of a file doesn’t have to be “assembled” from its diffs—the complete revision of each file is immediately available from Git's internal database.

Git's snapshot model has a far-reaching impact on virtually every aspect of its version control model, affecting everything from its branching and merging tools to its collaboration work-flows.

|  |  |
| --- | --- |
| git commit | Commit the staged snapshot. This will launch a text editor prompting you for a commit message. After you’ve entered a message, save the file and close the editor to create the actual commit. |
| git commit -a | Commit a snapshot of all changes in the working directory. This only includes modifications to tracked files (those that have been added with git add at some point in their history). |
| git commit -m "commit message" | A shortcut command that immediately creates a commit with a passed commit message. By default, git commit will open up the locally configured text editor, and prompt for a commit message to be entered. Passing the -m option will forgo the text editor prompt in-favor of an inline message. |
| git commit -am "commit message" | A power user shortcut command that combines the -a and -m options. This combination immediately creates a commit of all the staged changes and takes an inline commit message. |
| git commit --amend | This option adds another level of functionality to the commit command. Passing this option will modify the last commit. Instead of creating a new commit, staged changes will be added to the previous commit. This command will open up the system's configured text editor and prompt to change the previously specified commit message. |

Git doesn't require commit messages to follow any specific formatting constraints, but the canonical format is to summarize the entire commit on the first line in less than 50 characters, leave a blank line, then a detailed explanation of what’s been changed. For example:

Change the message displayed by hello.py

- Update the sayHello() function to output the user's name

- Change the sayGoodbye() function to a friendlier message

It is a common practice to use the first line of the commit message as a subject line, similar to an email. The rest of the log message is considered the body and used to communicate details of the commit change set. Note that many developers also like to use the present tense in their commit messages. This makes them read more like actions on the repository, which makes many of the history-rewriting operations more intuitive.

git add hello.py

git commit –amend

This will once again, open up the configured text editor. This time, however, it will be pre-filled with the commit message we previously entered. This indicates that we are not creating a new commit, but editing the last.

The git commit command is one of the core primary functions of Git. Prior use of the git add command is required to select the changes that will be staged for the next commit. Then git commit is used to create a snapshot of the staged changes along a timeline of a Git projects history. Learn more about [git add](https://www.atlassian.com/git/tutorials/saving-changes) usage on the accompanying page. The [git status](https://www.atlassian.com/git/tutorials/inspecting-a-repository) command can be used to explore the state of the staging area and pending commit.

If you want to skip the staging area, Git provides a simple shortcut. Adding the -a option to the git commit command makes Git automatically stage every file that is already tracked before doing the commit,

$ git commit -am 'added new benchmarks'

$ git commit

This will again prompt you for a message describing the change, and then record a new version of the project.

Alternatively, instead of running git add beforehand, you can use

$ git commit -a

Git commits can be captured and built up locally, then pushed to a remote server as needed using the git push -u origin master command.

The commit command has three steps. It creates a tree graph to represent the content of the version of the project being committed. It creates a commit object. It points the current branch at the new commit object

* Git records the current state of the project by creating a tree graph from the index. This tree graph records the location and content of every file in the project.

The graph is composed of two types of object: blobs and trees.

Blobs are stored by git add. They represent the content of files.

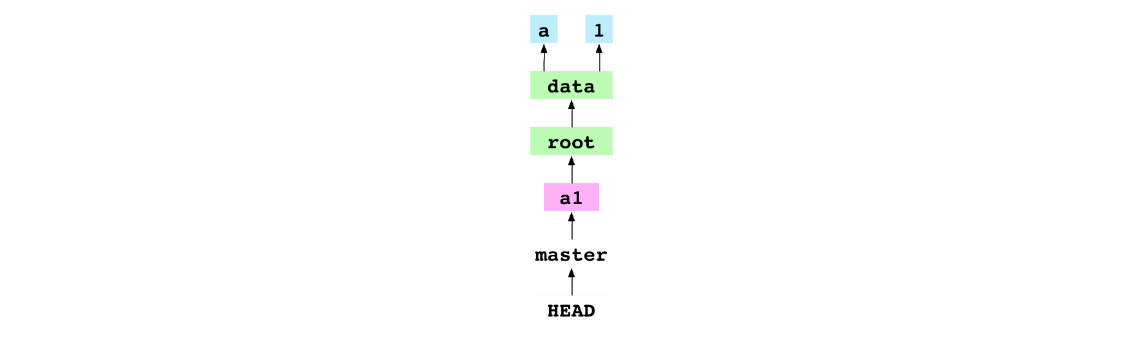
Trees are stored when a commit is made. A tree represents a directory in the working copy.

* git commit creates a commit object after creating the tree graph. The commit object is just another text file in .git/objects/:
* Finally, the commit command points the current branch at the new commit object. Which is the current branch? Git goes to the HEAD file at .git/HEAD and finds:

ref: refs/heads/master

This says that HEAD is pointing at master. master is the current branch.

HEAD and master are both refs. A ref is a label used by Git or the user to identify a specific commit.



HEAD pointing at master and master pointing at the a1 commit

## Git rm

To remove a file from Git, you have to remove it from your tracked files (more accurately, remove it from your staging area) and then commit. The git rm command does that, and also removes the file from your working directory so you don’t see it as an untracked file the next time around.

The next time you commit, the file will be gone and no longer tracked. If you modified the file and added it to the staging area already, you must force the removal with the -f option. This is a safety feature to prevent accidental removal of data that hasn’t yet been recorded in a snapshot and that can’t be recovered from Git.

Another useful thing you may want to do is to keep the file in your working tree but remove it from your staging area. In other words, you may want to keep the file on your hard drive but not have Git track it anymore. This is particularly useful if you forgot to add something to your .gitignore file and accidentally staged it, like a large log file or a bunch of .a compiled files. To do this, use the --cached option:

$ git rm --cached README

That means you can do things such as:

$ git rm log/\*.log

$ git rm /\*~

**Moving Files**

Unlike many other VCS systems, Git doesn’t explicitly track file movement. If you rename a file in Git, no metadata is stored in Git that tells it you renamed the file. However, Git is pretty smart about figuring that out after the fact — we’ll deal with detecting file movement a bit later. Thus it’s a bit confusing that Git has a mv command. If you want to rename a file in Git, you can run something like:

$ git mv file\_from file\_to

However, this is equivalent to running something like this:

$ mv README.md README

$ git rm README.md

$ git add README

Git figures out that it’s a rename implicitly, so it doesn’t matter if you rename a file that way or with the mv command. The only real difference is that git mv is one command instead of three — it’s a convenience function. More importantly, you can use any tool you like to rename a file, and address

the add/rm later, before you commit.

A common question when getting started with Git is "How do I tell Git not to track a file (or files) any more?" The git rm command is used to remove files from a Git repository. It can be thought of as the inverse of the [git add](https://www.atlassian.com/git/tutorials/saving-changes) command.

## Git rm Overview

The git rm command can be used to remove individual files or a collection of files. The primary function of git rm is to remove tracked files from the Git index. Additionally, git rm can be used to remove files from both the staging index and the working directory. There is no option to remove a file from only the working directory. The files being operated on must be identical to the files in the current HEAD. If there is a discrepancy between the HEAD version of a file and the staging index or working tree version, Git will block the removal. This block is a safety mechanism to prevent removal of in-progress changes.

Note that git rm does not remove branches. Learn more about [using git branches](https://www.atlassian.com/git/tutorials/using-branches)

## Usage

<file>…​

Specifies the target files to remove. The option value can be an individual file, a space delimited list of files file1 file2 file3, or a wildcard file glob (~./directory/\*).

-f  
--force

The -f option is used to override the safety check that Git makes to ensure that the files in HEAD match the current content in the staging index and working directory.

-n  
--dry-run

The "dry run" option is a safeguard that will execute the git rm command but not actually delete the files. Instead it will output which files it would have removed.

-r

The -r option is shorthand for 'recursive'. When operating in recursive mode git rm will remove a target directory and all the contents of that directory.

--

The separator option is used to explicitly distinguish between a list of file names and the arguments being passed to git rm. This is useful if some of the file names have syntax that might be mistaken for other options.

--cached

The cached option specifies that the removal should happen only on the staging index. Working directory files will be left alone.

--ignore-unmatch

This causes the command to exit with a 0 sigterm status even if no files matched. This is a Unix level status code. The code 0 indicates a successful invocation of the command. The --ignore-unmatch option can be helpful when using git rm as part of a greater shell script that needs to fail gracefully.

-q  
--quiet

The quiet option hides the output of the git rm command. The command normally outputs one line for each file removed.

## How to undo git rm

Executing git rm is not a permanent update. The command will update the staging index and the working directory. These changes will not be persisted until a new commit is created and the changes are added to the commit history. This means that the changes here can be "undone" using common Git commands.

git reset HEAD

A reset will revert the current staging index and working directory back to the HEAD commit. This will undo a git rm.

git checkout .

A checkout will have the same effect and restore the latest version of a file from HEAD.

In the event that git rm was executed and a new commit was created which persist the removal, git reflog can be used to find a ref that is before the git rm execution. Learn more about [using git reflog](https://www.atlassian.com/git/tutorials/rewriting-history/git-reflog).

## Discussion

The <file> argument given to the command can be exact paths, wildcard file glob patterns, or exact directory names. The command removes only paths currently commited to the Git repository.

Wildcard file globbing matches across directories. It is important to be cautious when using wildcard globs. Consider the examples: directory/\* and directory\*. The first example will remove all sub files of directory/ whereas the second example will remove all sibling directories like directory1 directory2 directory\_whatever which may be an unexpected result.

## The scope of git rm

The git rm command operates on the current branch only. The removal event is only applied to the working directory and staging index trees. The file removal is not persisted to the repository history until a new commit is created.

## Why use git rm instead of rm

A Git repository will recognize when a regular shell rm command has been executed on a file it is tracking. It will update the working directory to reflect the removal. It will not update the staging index with the removal. An additional git add command will have to be executed on the removed file paths to add the changes to the staging index. The git rm command acts a shortcut in that it will update the working directory and the staging index with the removal.

## Examples

git rm Documentation/\\*.txt

This example uses a wildcard file glob to remove all \*.txt files that are children of the Documentation directory and any of its subdirectories.

Note that the asterisk \* is escaped with slashes in this example; this is a guard that prevents the shell from expanding the wildcard. The wildcard then expands the pathnames of files and subdirectories under the Documentation/ directory.

git rm -f git-\*.sh

This example uses the force option and targets all wildcard git-\*.sh files. The force option explicitly removes the target files from both the working directory and staging index.

## How to remove files no longer in the filesystem

As stated above in "Why use git rm instead of rm" , git rm is actually a convenience command that combines the standard shell rm and git add to remove a file from the working directory and promote that removal to the staging index. A repository can get into a cumbersome state in the event that several files have been removed using only the standard shell rm command.

If intentions are to record all the explicitly removed files as part of the next commit, git commit -a will add all the removal events to the staging index in preparation of the next commit.

If however, intentions are to persistently remove the files that were removed with the shell rm, use the following command:

git diff --name-only --diff-filter=D -z | xargs -0 git rm --cached

This command will generate a list of the removed files from the working directory and pipe that list to git rm --cached which will update the staging index.

## Git rm summary

git rm is a command that operates on two of the primary Git [internal state management trees](https://www.atlassian.com/git/tutorials/undoing-changes/git-reset): the working directory, and staging index. git rm is used to remove a file from a Git repository. It is a convenience method that combines the effect of the default shell rm command with git add. This means that it will first remove a target from the filesystem and then add that removal event to the staging index. The command is one of many that can be used for [undoing changes in Git.](https://www.atlassian.com/git/tutorials/undoing-changes)

# Branches

### Create a branch

git branch deputy

The user creates a new branch called deputy. This just creates a new file at .git/refs/heads/deputy that contains the hash that HEAD is pointing at: the hash of the a3 commit.

Merging two branches means merging two commits

### Merge an ancestor

For this merge, Git does nothing. It reports it is Already up-to-date.

### Merge a descendent

git checkout master

git merge deputy

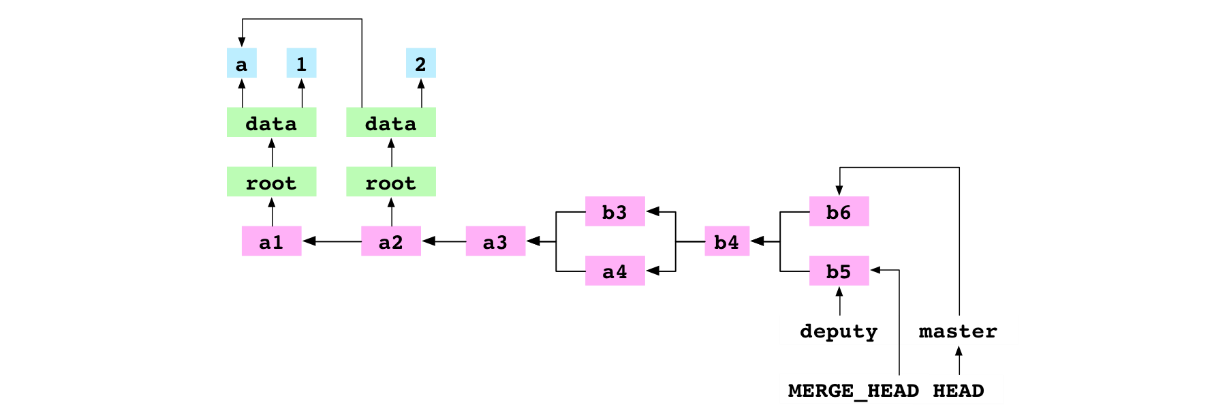
They merge deputy into master. Git discovers that the receiver commit, a2, is an ancestor of the giver commit, a3. It can do a fast-forward merge

### Merge two commits from different lineages that both modify the same file

$ git merge deputy

The user merges deputy into master. There is a conflict and the merge is paused.

First, Git writes the hash of the giver commit to a file at .git/MERGE\_HEAD



Adding a conflicted file tells Git that the conflict is resolved

It deletes the file at .git/MERGE\_HEAD. This completes the merge.

~/alpha $ cd ..

~ $ cp -R alpha bravo

The user copies the contents of the alpha/ repository to the bravo/ directory. This produces the following directory structure:

# Inspecting

# Git diff

## Comparing changes with git diff

Diffing is a function that takes two input data sets and outputs the changes between them. git diff is a multi-use Git command that when executed runs a diff function on Git data sources. These data sources can be commits, branches, files and more. This document will discuss common invocations of git diff and diffing work flow patterns. The git diff command is often used along with git status and git log to analyze the current state of a Git repo.

## Reading diffs: outputs

### Raw output format

The following examples will be executed in a simple repo. The repo is created with the commands below:

$:> mkdir diff\_test\_repo

$:> cd diff\_test\_repo

$:> touch diff\_test.txt

$:> echo "this is a git diff test example" > diff\_test.txt

$:> git init .

Initialized empty Git repository in /Users/kev/code/test/.git/

$:> git add diff\_test.txt

$:> git commit -am"add diff test file"

[master (root-commit) 6f77fc3] add diff test file

1 file changed, 1 insertion(+)

create mode 100644 diff\_test.txt

If we execute git diff at this point, there will be no output. This is expected behavior as there are no changes in the repo to diff. Once the repo is created and we've added the diff\_test.txt file, we can change the contents of the file to start experimenting with diff output.

$:> echo "this is a diff example" > diff\_test.txt

Executing this command will change the content of the diff\_test.txt file. Once modified, we can view a diff and analyze the output. Now executing git diff will produce the following output:

diff --git a/diff\_test.txt b/diff\_test.txt

index 6b0c6cf..b37e70a 100644

--- a/diff\_test.txt

+++ b/diff\_test.txt

@@ -1 +1 @@

-this is a git diff test example

+this is a diff example

Let us now examine a more detailed breakdown of the diff output.

### 1. Comparison input

diff --git a/diff\_test.txt b/diff\_test.txt

This line displays the input sources of the diff. We can see that a/diff\_test.txt and b/diff\_test.txt have been passed to the diff.

### 2. Meta data

index 6b0c6cf..b37e70a 100644

This line displays some internal Git metadata. You will most likely not need this information. The numbers in this output correspond to Git object version hash identifiers.

### 3. Markers for changes

--- a/diff\_test.txt

+++ b/diff\_test.txt

These lines are a legend that assigns symbols to each diff input source. In this case, changes from a/diff\_test.txt are marked with a --- and the changes from b/diff\_test.txt are marked with the +++ symbol.

### 4. Diff chunks

The remaining diff output is a list of diff 'chunks'. A diff only displays the sections of the file that have changes. In our current example, we only have one chunk as we are working with a simple scenario. Chunks have their own granular output semantics.

@@ -1 +1 @@

-this is a git diff test example

+this is a diff example

The first line is the chunk header. Each chunk is prepended by a header inclosed within @@ symbols. The content of the header is a summary of changes made to the file. In our simplified example, we have -1 +1 meaning line one had changes. In a more realistic diff, you would see a header like:

@@ -34,6 +34,8 @@

In this header example, 6 lines have been extracted starting from line number 34. Additionally, 8 lines have been added starting at line number 34.

The remaining content of the diff chunk displays the recent changes. Each changed line is prepended with a + or - symbol indicating which version of the diff input the changes come from. As we previously discussed, - indicates changes from the a/diff\_test.txt and + indicates changes from b/diff\_test.txt.

## Highlighting changes

### 1. git diff --color-words

git diff also has a special mode for highlighting changes with much better granularity: ‐‐color-words. This mode tokenizes added and removed lines by whitespace and then diffs those.

$:> git diff --color-words

diff --git a/diff\_test.txt b/diff\_test.txt

index 6b0c6cf..b37e70a 100644

--- a/diff\_test.txt

+++ b/diff\_test.txt

@@ -1 +1 @@

this is agit difftest example

Now the output displays only the color-coded words that have changed.

### 2. git diff-highlight

If you clone the git source, you’ll find a sub-directory called contrib. It contains a bunch of git-related tools and other interesting bits and pieces that haven’t yet been promoted to git core. One of these is a Perl script called diff-highlight. Diff-highlight pairs up matching lines of diff output and highlights sub-word fragments that have changed.

$:> git diff | /your/local/path/to/git-core/contrib/diff-highlight/diff-highlight

diff --git a/diff\_test.txt b/diff\_test.txt

index 6b0c6cf..b37e70a 100644

--- a/diff\_test.txt

+++ b/diff\_test.txt

@@ -1 +1 @@

-this is a git diff test example

+this is a diff example

Now we’ve pared down our diff to the smallest possible change.

## Diffing binary files

In addition to the text file utilities we have thus far demonstrated, git diff can be run on binary files. Unfortunately, the default output is not very helpful.

$:> git diff

Binary files a/script.pdf and b/script.pdf differ

Git does have a feature that allows you to specify a shell command to transform the content of your binary files into text prior to performing the diff. It does require a little set up though. First, you need to specify a textconv filter describing how to convert a certain type of binary to text. We're using a simple utility called pdftohtml (available via homebrew) to convert my PDFs into human readable HTML. You can set this up for a single repository by editing your .git/config file, or globally by editing ~ /.gitconfig

[diff "pdfconv"]

textconv=pdftohtml -stdout

Then all you need to do is associate one or more file patterns with our pdfconv filter. You can do this by creating a .gitattributes file in the root of your repository.

\*.pdf diff=pdfconv

Once configured, git diff will first run the binary file through the configured converter script and diff the converter output. The same technique can be applied to get useful diffs from all sorts of binary files, for example: zips, jars and other archives: using unzip -l (or similar) in place of pdf2html will show you paths that have been added or removed between commits images: exiv2 can be used to show metadata changes such as image dimensions documents: conversion tools exist for transforming .odf, .doc and other document formats to plain text. In a pinch, strings will often work for binary files where no formal converter exists.

## Comparing files: git diff file

The git diff command can be passed an explicit file path option. When a file path is passed to git diff the diff operation will be scoped to the specified file. The below examples demonstrate this usage.

git diff HEAD ./path/to/file

This example is scoped to ./path/to/file when invoked, it will compare the specific changes in the working directory, against the index, showing the changes that are not staged yet. By default git diff will execute the comparison against HEAD. Omitting HEAD in the example above git diff ./path/to/file has the same effect.

git diff --cached ./path/to/file

When git diff is invoked with the --cached option the diff will compare the staged changes with the local repository. The --cached option is synonymous with --staged.

## Comparing all changes

Invoking git diff without a file path will compare changes across the entire repository. The above, file specific examples, can be invoked without the ./path/to/file argument and have the same output results across all files in the local repo.

## Changes since last commit

By default git diff will show you any uncommitted changes since the last commit.

git diff

## Comparing files between two different commits

git diff can be passed Git refs to commits to diff. Some example refs are, HEAD, tags, and branch names. Every commit in Git has a commit ID which you can get when you execute GIT LOG. You can also pass this commit ID to git diff.

$:> git log --prety=oneline

957fbc92b123030c389bf8b4b874522bdf2db72c add feature

ce489262a1ee34340440e55a0b99ea6918e19e7a rename some classes

6b539f280d8b0ec4874671bae9c6bed80b788006 refactor some code for feature

646e7863348a427e1ed9163a9a96fa759112f102 add some copy to body

$:> git diff 957fbc92b123030c389bf8b4b874522bdf2db72c ce489262a1ee34340440e55a0b99ea6918e19e7a

## Comparing branches

### Comparing two branches

Branches are compared like all other ref inputs to git diff

git diff branch1..other-feature-branch

This example introduces the dot operator. The two dots in this example indicate the diff input is the tips of both branches. The same effect happens if the dots are omitted and a space is used between the branches. Additionally, there is a three dot operator:

git diff branch1...other-feature-branch

The three dot operator initiates the diff by changing the first input parameter branch1. It changes branch1 into a ref of the shared common ancestor commit between the two diff inputs, the shared ancestor of branch1 and other-feature-branch. The last parameter input parameter remains unchanged as the tip of other-feature-branch.

## Comparing files from two branches

To compare a specific file across branches, pass in the path of the file as the third argument to git diff

git diff master new\_branch ./diff\_test.txt

# Inspecting a repository

## git log

The git log command displays committed snapshots. It lets you list the project history, filter it, and search for specific changes. While git status lets you inspect the working directory and the staging area, git log only operates on the committed history.

A diagram of different types of objects

Description automatically generated

Log output can be customized in several ways, from simply filtering commits to displaying them in a completely user-defined format. Some of the most common configurations of git log are presented below.

The git log command is Git's basic tool for exploring a repository’s history. It’s what you use when you need to find a specific version of a project or figure out what changes will be introduced by merging in a feature branch.

commit 3157ee3718e180a9476bf2e5cab8e3f1e78a73b7

Author: John Smith

Most of this is pretty straightforward; however, the first line warrants some explanation. The 40-character string after commit is an SHA-1 checksum of the commit’s contents. This serves two purposes. First, it ensures the integrity of the commit—if it was ever corrupted, the commit would generate a different checksum. Second, it serves as a unique ID for the commit.

This ID can be used in commands like git log <since>..<until> to refer to specific commits. For instance, git log 3157e..5ab91 will display everything between the commits with ID's 3157e and 5ab91. Aside from checksums, branch names (discussed in the [Branch Module](https://www.atlassian.com/git/tutorials/using-branches)) and the HEAD keyword are other common methods for referring to individual commits. HEAD always refers to the current commit, be it a branch or a specific commit.

The ~ character is useful for making relative references to the parent of a commit. For example, 3157e~1 refers to the commit before 3157e, and HEAD~3 is the great-grandparent of the current commit.

The idea behind all of these identification methods is to let you perform actions based on specific commits. The git log command is typically the starting point for these interactions, as it lets you find the commits you want to work with.

### Usage

$ git log

Display the entire commit history using the default formatting. If the output takes up more than one screen, you can use Space to scroll and q to exit.

$ git log -n <limit>

Limit the number of commits by <limit>. For example, git log -n 3 will display only 3 commits.

$ git log --oneline

Condense each commit to a single line. This is useful for getting a high-level overview of the project history.

$ git log --stat

Along with the ordinary git log information, include which files were altered and the relative number of lines that were added or deleted from each of them.

$ git log -p

Display the patch representing each commit. This shows the full diff of each commit, which is the most detailed view you can have of your project history.

$ git log --author="<pattern>"

Search for commits by a particular author. The <pattern> argument can be a plain string or a regular expression.

$ git log --grep="<pattern>"

Search for commits with a commit message that matches <pattern>, which can be a plain string or a regular expression.

$ git log <since>..<until>

Show only commits that occur between <since> and <until>. Both arguments can be either a commit ID, a branch name, HEAD, or any other kind of [revision reference](http://www.kernel.org/pub/software/scm/git/docs/gitrevisions.html).

$ git log <file>

Only display commits that include the specified file. This is an easy way to see the history of a particular file.

$ git log --graph --decorate --oneline

A few useful options to consider. The --graph flag that will draw a text based graph of the commits on the left hand side of the commit messages. --decorate adds the names of branches or tags of the commits that are shown. --oneline shows the commit information on a single line making it easier to browse through commits at-a-glance.

### Discussion

### Example

The Usage section provides many examples of git log, but keep in mind that several options can be combined into a single command:

git log --author="John Smith" -p hello.py

This will display a full diff of all the changes John Smith has made to the file hello.py.

The .. syntax is a very useful tool for comparing branches. The next example displays a brief overview of all the commits that are in some-feature that are not in master.

git log --oneline master..some-feature

# Git blame

* + The high-level function of git blame is the display of author metadata attached to specific committed lines in a file. This is used to explore the history of specific code and answer questions about what, how, and why the code was added to a repository.

The git blame command is a versatile troubleshooting utility that has extensive usage options. The high-level function of git blame is the display of author metadata attached to specific committed lines in a file. This is used to examine specific points of a file's history and get context as to who the last author was that modified the line. This is used to explore the history of specific code and answer questions about what, how, and why the code was added to a repository.

Git blame is often used with a GUI display. Online Git hosting sites like [Bitbucket](http://bitbucket-marketing.atlassian.com/product) offer blame views which are UI wrappers to git blame. These views are referenced in collaborative discussions around pull requests and commits. Additionally, most IDE's that have Git integration also have dynamic blame views.

## How It Works

In order to demonstrate git blame we need a repository with some history. We will use the open source project [git-blame-example](https://bitbucket.org/kevzettler/git-blame-example). This open source project is a simple repository that contains a README.md file which has a few commits from different authors. The first step of our git blame usage example is to git clone the example repository.

git clone https://kevzettler@bitbucket.org/kevzettler/git-blame-example.git && cd git-blame-example

Now that we have a copy of the example code we can start exploring it with git blame. The state of the example repo can be examined using [git log](https://www.atlassian.com/git/tutorials/git-log). The commit history should look like the following:

$ git log

commit 548dabed82e4e5f3734c219d5a742b1c259926b2

Author: Juni Mukherjee <jmukherjee@atlassian.com>

Date: Thu Mar 1 19:55:15 2018 +0000

Another commit to help git blame track the who, the what, and the when

commit eb06faedb1fdd159d62e4438fc8dbe9c9fe0728b

Author: Juni Mukherjee <jmukherjee@atlassian.com>

Date: Thu Mar 1 19:53:23 2018 +0000

Creating the third commit, along with Kev and Albert, so that Kev can get git blame docs.

commit 990c2b6a84464fee153253dbf02e845a4db372bb

Merge: 82496ea 89feb84

Author: Albert So <aso@atlassian.com>

Date: Thu Mar 1 05:33:01 2018 +0000

Merged in albert-so/git-blame-example/albert-so/readmemd-edited-online-with-bitbucket-1519865641474 (pull request #2)

README.md edited online with Bitbucket

commit 89feb84d885fe33d1182f2112885c2a64a4206ec

Author: Albert So <aso@atlassian.com>

Date: Thu Mar 1 00:54:03 2018 +0000

README.md edited online with Bitbucket

git blame only operates on individual files. A file-path is required for any useful output. The default execution of git blame will simply output the commands help menu. For this example, we will operate on the README.MD file. It is a common open source software practice to include a README file in the root of a git repository as documentation source for the project.

git blame README.MD

Executing the above command will give us our first sample of blame output. The following output is a subset of the full blame output of the README. Additionally, this output is static is reflective of the state of the repo at the time of this writing.

$ git blame README.md

82496ea3 (kevzettler 2018-02-28 13:37:02 -0800 1) # Git Blame example

82496ea3 (kevzettler 2018-02-28 13:37:02 -0800 2)

89feb84d (Albert So 2018-03-01 00:54:03 +0000 3) This repository is an example of a project with multiple contributors making commits.

82496ea3 (kevzettler 2018-02-28 13:37:02 -0800 4)

82496ea3 (kevzettler 2018-02-28 13:37:02 -0800 5) The repo use used elsewhere to demonstrate `git blame`

82496ea3 (kevzettler 2018-02-28 13:37:02 -0800 6)

89feb84d (Albert So 2018-03-01 00:54:03 +0000 7) Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod TEMPOR incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum

89feb84d (Albert So 2018-03-01 00:54:03 +0000 8)

eb06faed (Juni Mukherjee 2018-03-01 19:53:23 +0000 9) Annotates each line in the given file with information from the revision which last modified the line. Optionally, start annotating from the given revision.

eb06faed (Juni Mukherjee 2018-03-01 19:53:23 +0000 10)

548dabed (Juni Mukherjee 2018-03-01 19:55:15 +0000 11) Creating a line to support documentation needs for git blame.

548dabed (Juni Mukherjee 2018-03-01 19:55:15 +0000 12)

548dabed (Juni Mukherjee 2018-03-01 19:55:15 +0000 13

This is a sample of the first 13 lines of the README.md file. To better understand this output lets break down a line. The following table displays the content of line 3 and the columns of the table indicate the column content.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Id | Author | Timestamp | Line Number | Line Content |
| 89feb84d | Albert So | 2018-03-01 00:54:03 +0000 | 3 | This repository is an example of a project with multiple contributors making commits. |

If we review the blame output list, we can make some observations. There are three authors listed. In addition to the project's maintainer Kev Zettler, Albert So, and Juni Mukherjee are also listed. Authors are generally the most valuable part of git blame output. The timestamp column is also primarily helpful. What the change was is indicated by line content column.

## Common Options

git blame -L 1,5 README.md

The -L option will restrict the output to the requested line range. Here we have restricted the output to lines 1 through 5.

git blame -e README.md

The -e option shows the authors email address instead of username.

git blame -w README.md

The -w option ignores whitespace changes. If a previous author has modified the spacing of a file by switching from tabs to spaces or adding new lines this, unfortunately, obscures the output of git blame by showing these changes.

git blame -M README.md

The -M option detects moved or copied lines within in the same file. This will report the original author of the lines instead of the last author that moved or copied the lines.

git blame -C README.md

The -C option detects lines that were moved or copied from other files. This will report the original author of the lines instead of the last author that moved or copied the lines.

## Git Blame vs Git Log

While git blame displays the last author that modified a line, often times you will want to know when a line was originally added. This can be cumbersome to achieve using git blame. It requires a combination of the -w, -C, and -M options. It can be far more convenient to use the [git log](https://www.atlassian.com/git/tutorials/git-log) command.

To list all original commits in-which a specific code piece was added or modified execute git log with the -S option. Append the -S option with the code you are looking for. Let's take one of the lines from the README output above to use as an example. Let us take the text "CSS3D and WebGL renderers" from Line 12 of the README output.

$ git log -S"CSS3D and WebGL renderers." --pretty=format:'%h %an %ad %s'

e339d3c85 Mario Schuettel Tue Oct 13 16:51:06 2015 +0200 reverted README.md to original content

509c2cc35 Daniel Tue Sep 8 13:56:14 2015 +0200 Updated README

cb20237cc Mr.doob Mon Dec 31 00:22:36 2012 +0100 Removed DOMRenderer. Now with the CSS3DRenderer it has become irrelevant.

This output shows us that content from the README was added or modified 3 times by 3 different authors. It was originally added in commit cb20237cc by Mr.doob. In this example, git log has also been prepended with the --pretty-format option. This option converts the default output format of git log into one that matches the format of git log. For more information on usage and configuration options visit the [git log](https://www.atlassian.com/git/tutorials/git-log) page.

## Summary

The git blame command is used to examine the contents of a file line by line and see when each line was last modified and who the author of the modifications was. The output format of git blame can be altered with various command line options. Online Git hosting solutions like Bitbucket offer blame views, which offer a superior user experience to command line git blame usage. git blame and git log can be used in combination to help discover the history of a file's contents. The git log command has some similar blame functionality, to learn more visit the [git log](https://www.atlassian.com/git/tutorials/git-log) overview page.

=======

## Reviewing History - Git Log

The linkgit:git-log[1] command can show lists of commits. On its own, it shows all commits reachable from the parent commit;

$ git log v2.5.. # commits since (not reachable from) v2.5

$ git log test..master # commits reachable from master but not test

$ git log master..test # commits reachable from test but not master

$ git log master...test # commits reachable from either test or

# master, but not both

$ git log --since="2 weeks ago" # commits from the last 2 weeks

$ git log Makefile # commits that modify Makefile

$ git log fs/ # commits that modify any file under fs/

$ git log -S'foo()' # commits that add or remove any file data

# matching the string 'foo()'

$ git log --no-merges # dont show merge commits

And of course you can combine all of these; the following finds commits since v2.5 which touch the Makefile or any file under fs:

$ git log v2.5.. Makefile fs/

You can also ask git log to show patches:

$ git log -p

If you pass the --stat option to 'git log', it will show you which files have changed in that commit and how many lines were added and removed from each.

You can also format the log output almost however you want. The '--pretty' option can take a number of preset formats, such as 'oneline':

$ git log --pretty=oneline

$ git log --pretty=short

$ git log --pretty=full

$ git log --pretty=fuller

If those formats aren't exactly what you need, you can also create your own format with the '--pretty=format' option (see the git-log[1] docs for all the formatting options).

$ git log --pretty=format:'%h was %an, %ar, message: %s'

Another interesting thing you can do is visualize the commit graph with the '--graph' option,

you can reverse the order of the log with the '--reverse' option

## Comparing Commits - Git Diff

You can generate diffs between any two versions of your project using

$ git diff master..test

That will produce the diff between the tips of the two branches. If you'd prefer to find the diff from their common ancestor to test, you can use three dots instead of two:

$ git diff master...test

You will commonly use linkgit:git-diff[1] for figuring out differences between your last commit, your index, and your current working directory. A common use is to simply run

$ git diff

which will show you changes in the working directory that are not yet staged for the next commit. If you want to see what isstaged for the next commit, you can run

$ git diff --cached

$ git diff HEAD

which shows changes in the working directory since your last commit; what you would be committing if you run "git commit -a".

If you want to see how your current working directory differs from the state of the project in another branch, you can run something like

$ git diff test

This will show you what is different between your current working directory and the snapshot on the 'test' branch ou can also limit the comparison to a specific file or subdirectory by adding a *path limiter*:

$ git diff HEAD -- ./lib

That command will show the changes between your current working directory and the last commit (or, more accurately, the tip of the current branch), limiting the comparison to files in the 'lib' subdirectory.

If you don't want to see the whole patch, you can add the '--stat' option

**Viewing the Commit History**

The most basic and powerful tool to do this is the git log command.

One of the more helpful options is -p or --patch, which shows the difference (the *patch* output) introduced in each commit. You can also limit the number of log entries displayed, such as using -2 to show only the last two entries.

$ git log -p -2

if you want to see some abbreviated stats for each commit, you can use the --stat option

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

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

$ git log --pretty=oneline

The most interesting option is format, which allows you to specify your own log output format. This is especially useful when you’re generating output for machine parsing — because you specify the format explicitly, you know it won’t change with updates to Git:

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

ca82a6d - Scott Chacon, 6 years ago : changed the version number

085bb3b - Scott Chacon, 6 years ago : removed unnecess

**Option Description of Output**

%H Commit hash

%h Abbreviated commit hash

%T Tree hash

%t Abbreviated tree hash

%P Parent hashes

%p Abbreviated parent hashes

%an Author name

%ae Author email

%ad Author date (format respects the --date=option)

%ar Author date, relative

%cn Committer name

%ce Committer email

%cd Committer date

%cr Committer date, relative

%s Subject

The oneline and format options are particularly useful with another log option called --graph. This option adds a nice little ASCII graph showing your branch and merge history:

*Table 2. Common options to* git log

**Option Description**

-p Show the patch introduced with each commit.

--stat Show statistics for files modified in each commit.

--shortstat Display only the changed/insertions/deletions line from the --stat command.

--name-only Show the list of files modified after the commit information.

--name-status Show the list of files affected with added/modified/deleted information as well.

--abbrev-commit Show only the first few characters of the SHA-1 checksum instead of all 40.

--relative-date Display the date in a relative format (for example, “2 weeks ago”) instead of using the full date format.

--graph Display an ASCII graph of the branch and merge history beside the log output.

--pretty Show commits in an alternate format. Options include oneline, short, full,fuller, and format (where you specify your own format).

--oneline Shorthand for --pretty=oneline --abbrev-commit used together.

**Limiting Log Output**

In addition to output-formatting options, git log takes a number of useful limiting options

* -<n>, where n is any integer to show the last n commits
* --since and --until the time-limiting options.

$ git log --since=2.weeks

This command works with lots of formats — you can specify a specific date like "2008-01-15", or a relative date such as "2 years 1 day 3 minutes ago"

* The --author option allows you to filter on a specific author,
* the --grep option lets you search for keywords in the commit messages

Another really helpful filter is the -S option (colloquially referred to as Git’s “pickaxe” option), which takes a string and shows only those commits that changed the number of occurrences of that string.

$ git log -S FindNextStructure

The last really useful option to pass to git log as a filter is a path. If you specify a directory or file name, you can limit the log output to commits that introduced a change to those files. This is always the last option and is generally preceded by double dashes (--) to separate the paths from the options.

**Option Description**

-<n> Show only the last n commits

--since, --after Limit the commits to those made after the specified date.

--until, --before Limit the commits to those made before the specified date.

--author Only show commits in which the author entry matches the specified string.

--committer Only show commits in which the committer entry matches the specified string.

--grep Only show commits with a commit message containing the string

-S Only show commits adding or removing code matching the string

$ git log --pretty="%h - %s" --author='Junio C Hamano' --since="2008-10-01"

--before="2008-11-01" --no-merges -- testData/

To prevent the display of merge commits cluttering up your log history, simply add the log option --no-merges

## Advanced

## Stash

Git has an additional saving mechanism called 'the stash'. The stash is an ephemeral storage area for changes that are not ready to be committed. The stash operates on the working directory, the first of [the three trees](https://www.atlassian.com/git/tutorials/undoing-changes/git-reset) and has extensive usage options. To learn more visit the [git stash](https://www.atlassian.com/git/tutorials/saving-changes/git-stash) page.

git stash temporarily shelves (or stashes) changes you've made to your working copy so you can work on something else, and then come back and re-apply them later on. Stashing is handy if you need to quickly switch context and work on something else, but you're mid-way through a code change and aren't quite ready to commit.

## Stashing your work

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

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

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

You can reapply previously stashed changes with 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:

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

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

## Stashing untracked or ignored files

By default, running git stash will stash:

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

But it will **not** stash:

* new files in your working copy that have not yet been staged
* files that have been [ignored](https://www.atlassian.com/git/tutorials/gitignore)

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

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

You can include changes to [ignored](https://www.atlassian.com/git/tutorials/gitignore) files as well by passing the -a option (or --all) when running git stash.

A screenshot of a git stash options

Description automatically generated

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

$ git stash list

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

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

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

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

$ git stash save "add style to our site"

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

HEAD is now at 5002d47 our new homepage

$ git stash list

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

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

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

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

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

$ git stash pop stash@{2}

## Viewing stash diffs

You can view a summary of a stash with git stash show:

$ git stash show

index.html | 1 +

style.css | 3 +++

2 files changed, 4 insertions(+)

Or pass the -p option (or --patch) to view the full diff of a stash:

$ git stash show -p

diff --git a/style.css b/style.css

new file mode 100644

index 0000000..d92368b

--- /dev/null

+++ b/style.css

@@ -0,0 +1,3 @@

+\* {

+ text-decoration: blink;

+}

diff --git a/index.html b/index.html

index 9daeafb..ebdcbd2 100644

--- a/index.html

+++ b/index.html

@@ -1 +1,2 @@

+<link rel="stylesheet" href="style.css"/>

## Partial stashes

You can also choose to stash just a single file, a collection of files, or individual changes from within files. If you pass the -p option (or --patch) to git stash, it will iterate through each changed "hunk" in your working copy and ask whether you wish to stash it:

$ git stash -p

diff --git a/style.css b/style.css

new file mode 100644

index 0000000..d92368b

--- /dev/null

+++ b/style.css

@@ -0,0 +1,3 @@

+\* {

+ text-decoration: blink;

+}

Stash this hunk [y,n,q,a,d,/,e,?]? y

diff --git a/index.html b/index.html

index 9daeafb..ebdcbd2 100644

--- a/index.html

+++ b/index.html

@@ -1 +1,2 @@

+<link rel="stylesheet" href="style.css"/>

Stash this hunk [y,n,q,a,d,/,e,?]? n

A screenshot of a computer

Description automatically generated

## Creating a branch from your stash

If the changes on your branch diverge from the changes in your stash, you may run into conflicts when popping or applying your stash. Instead, you can use git stash branch to create a new branch to apply your stashed changes to:

$ git stash branch add-stylesheet stash@{1}

Switched to a new branch 'add-stylesheet'

On branch add-stylesheet

Changes to be committed:

new file: style.css

Changes not staged for commit:

modified: index.html

Dropped refs/stash@{1} (32b3aa1d185dfe6d57b3c3cc3b32cbf3e380cc6a)

This checks out a new branch based on the commit that you created your stash from, and then pops your stashed changes onto it.

## Cleaning up your stash

If you decide you no longer need a particular stash, you can delete it with git stash drop:

$ git stash drop stash@{1}

Dropped stash@{1} (17e2697fd8251df6163117cb3d58c1f62a5e7cdb)

Or you can delete all of your stashes with:

$ git stash clear

## How git stash works

If you just wanted to know how to use git stash, you can stop reading here. But if you're curious about how Git (and git stash) works under the hood, read on!

Stashes are actually encoded in your repository as commit objects. The special ref at .git/refs/stash points to your most recently created stash, and previously created stashes are referenced by the stash ref's reflog. This is why you refer to stashes by stash@{n}: you're actually referring to the nth reflog entry for the stash ref. Since a stash is just a commit, you can inspect it with git log:

$ git log --oneline --graph stash@{0}

\*-. 953ddde WIP on master: 5002d47 our new homepage

|\ \

| | \* 24b35a1 untracked files on master: 5002d47 our new homepage

| \* 7023dd4 index on master: 5002d47 our new homepage

|/

\* 5002d47 our new homepage

Depending on what you stashed, a single git stash operation creates either two or three new commits. The commits in the diagram above are:

* stash@{0}, a new commit to store the tracked files that were in your working copy when you ran git stash
* stash@{0}'s first parent, the pre-existing commit that was at HEAD when you ran git stash
* stash@{0}'s second parent, a new commit representing the index when you ran git stash
* stash@{0}'s third parent, a new commit representing untracked files that were in your working copy when you ran git stash. This third parent only created if:
  + your working copy actually contained untracked files; and
  + you specified the --include-untracked or --all option when invoked git stash.

How git stash encodes your worktree and index as commits:

* Before stashing, your worktree may contain changes to tracked files, untracked files, and ignored files. Some of these changes may also be staged in the index.

A screenshot of a diagram

Description automatically generated

Invoking git stash encodes any changes to tracked files as two new commits in your DAG: one for unstaged changes, and one for changes staged in the index. The special refs/stash ref is updated to point to them.

A screenshot of a diagram

Description automatically generated

Using the --include-untracked option also encodes any changes to untracked files as an additional commit.

A diagram of a diagram

Description automatically generated

Using the --all option includes changes to any ignored files alongside changes to untracked files in the same commit.

A diagram of a process

Description automatically generated

When you run git stash pop, the changes from the commits above are used to update your working copy and index, and the stash reflog is shuffled to remove the popped commit. Note that the popped commits aren't immediately deleted, but do become candidates for future garbage collection.

======

## Git tag

* + Tags are ref's that point to specific points in Git history. git tag is generally used to capture a point in history that is used for a marked version release (i.e. v1.0.1).

## Tagging

This document will discuss the Git concept of tagging and the git tag command. Tags are ref's that point to specific points in Git history. Tagging is generally used to capture a point in history that is used for a marked version release (i.e. v1.0.1). A tag is like a branch that doesn’t change. Unlike branches, tags, after being created, have no further history of commits. For more info on branches visit the git branch page. This document will cover the different kind of tags, how to create tags, listing all tags, deleting tags, sharing tags, and more.

## Creating a tag

To create a new tag execute the following command:

git tag <tagname>

Replace <tagname> with a semantic identifier to the state of the repo at the time the tag is being created. A common pattern is to use version numbers like git tag v1.4. Git supports two different types of tags, annotated and lightweight tags. The previous example created a lightweight tag. Lightweight tags and Annotated tags differ in the amount of accompanying meta data they store. A best practice is to consider Annotated tags as public, and Lightweight tags as private. Annotated tags store extra meta data such as: the tagger name, email, and date. This is important data for a public release. Lightweight tags are essentially 'bookmarks' to a commit, they are just a name and a pointer to a commit, useful for creating quick links to relevant commits.

## Annotated Tags

Annotated tags are stored as full objects in the Git database. To reiterate, They store extra meta data such as: the tagger name, email, and date. Similar to commits and commit messages Annotated tags have a tagging message. Additionally, for security, annotated tags can be signed and verified with GNU Privacy Guard (GPG). Suggested best practices for git tagging is to prefer annotated tags over lightweight so you can have all the associated meta-data.

git tag -a v1.4

Executing this command will create a new annotated tag identified with v1.4. The command will then open up the configured default text editor to prompt for further meta data input.

git tag -a v1.4 -m "my version 1.4"

Executing this command is similar to the previous invocation, however, this version of the command is passed the -m option and a message. This is a convenience method similar to git commit -m that will immediately create a new tag and forgo opening the local text editor in favor of saving the message passed in with the -m option.

## Lightweight Tags

git tag v1.4-lw

Executing this command creates a lightweight tag identified as v1.4-lw. Lightweight tags are created with the absence of the -a, -s, or -m options. Lightweight tags create a new tag checksum and store it in the .git/ directory of the project's repo.

## Listing Tags

To list stored tags in a repo execute the following:

git tag

This will output a list of tags:

v0.10.0

v0.10.0-rc1

v0.11.0

v0.11.0-rc1

v0.11.1

v0.11.2

v0.12.0

v0.12.0-rc1

v0.12.1

v0.12.2

v0.13.0

v0.13.0-rc1

v0.13.0-rc2

To refine the list of tags the -l option can be passed with a wild card expression:

$ git tag -l \*-rc\*

v0.10.0-rc1

v0.11.0-rc1

v0.12.0-rc1

v0.13.0-rc1

v0.13.0-rc2

v0.14.0-rc1

v0.9.0-rc1

v15.0.0-rc.1

v15.0.0-rc.2

v15.4.0-rc.3

This previous example uses the -l option and a wildcard expression of -rc which returns a list of all tags marked with a -rc prefix, traditionally used to identify release candidates.

## Tagging Old Commits

The previous tagging examples have demonstrated operations on implicit commits. By default, git tag will create a tag on the commit that HEAD is referencing. Alternatively git tag can be passed as a ref to a specific commit. This will tag the passed commit instead of defaulting to HEAD. To gather a list of older commits execute the git log command.

$ git log --pretty=oneline

15027957951b64cf874c3557a0f3547bd83b3ff6 Merge branch 'feature'

a6b4c97498bd301d84096da251c98a07c7723e65 add update method for thing

0d52aaab4479697da7686c15f77a3d64d9165190 one more thing

6d52a271eda8725415634dd79daabbc4d9b6008e Merge branch 'experiment'

Executing git log will output a list of commits. In this example we will pick the top most commit Merge branch 'feature' for the new tag. We will need to reference to the commit SHA hash to pass to Git:

git tag -a v1.2 15027957951b64cf874c3557a0f3547bd83b3ff6

Executing the above git tag invocation will create a new annotated commit identified as v1.2 for the commit we selected in the previous git log example.

## ReTagging/Replacing Old Tags

If you try to create a tag with the same identifier as an existing tag, Git will throw an error like:

fatal: tag 'v0.4' already exists

Additionally if you try to tag an older commit with an existing tag identifier Git will throw the same error.

In the event that you must update an existing tag, the -f FORCE option must be used.

git tag -a -f v1.4 15027957951b64cf874c3557a0f3547bd83b3ff6

Executing the above command will map the 15027957951b64cf874c3557a0f3547bd83b3ff6 commit to the v1.4 tag identifier. It will override any existing content for the v1.4 tag.

## Sharing: Pushing Tags to Remote

Sharing tags is similar to pushing branches. By default, git push will not push tags. Tags have to be explicitly passed to git push.

$ git push origin v1.4

Counting objects: 14, done.

Delta compression using up to 8 threads.

Compressing objects: 100% (12/12), done.

Writing objects: 100% (14/14), 2.05 KiB | 0 bytes/s, done.

Total 14 (delta 3), reused 0 (delta 0)

To git@bitbucket.com:atlasbro/gittagdocs.git

\* [new tag] v1.4 -> v1.4

To push multiple tags simultaneously pass the --tags option to git push command. When another user clones or pulls a repo they will receive the new tags.

## Checking Out Tags

You can view the state of a repo at a tag by using the [git checkout](https://www.atlassian.com/git/tutorials/using-branches/git-checkout) command.

git checkout v1.4

The above command will checkout the v1.4 tag. This puts the repo in a detached HEAD state. This means any changes made will not update the tag. They will create a new detached commit. This new detached commit will not be part of any branch and will only be reachable directly by the commits SHA hash. Therefore it is a best practice to create a new branch anytime you're making changes in a detached HEAD state.

## Deleting Tags

Deleting tags is a straightforward operation. Passing the -d option and a tag identifier to git tag will delete the identified tag.

$ git tag

v1

v2

v3

$ git tag -d v1

$ git tag

v2

v3

In this example git tag is executed to display a list of tags showing v1, v2, v3, Then git tag -d v1 is executed which deletes the v1 tag.

## Summary

To recap, Tagging is an additional mechanism used to create a snap shot of a Git repo. Tagging is traditionally used to create semantic version number identifier tags that correspond to software release cycles. The git tag command is the primary driver of tag: creation, modification and deletion. There are two types of tags; annotated and lightweight. Annotated tags are generally the better practices as they store additional valuable meta data about the tag. Additional Git commands covered in this document were [git push](https://www.atlassian.com/git/tutorials/syncing), and [git checkout](https://www.atlassian.com/git/tutorials/using-branches/git-checkout). Visit their corresponding pages for discussion on their extended use.