Links

<https://www.alexkras.com/category/git/>

<https://www.atlassian.com/git/tutorials/resetting-checking-out-and-reverting>

# What is version control

Version control systems are a category of software tools that help a software team manage changes to source code over time. Version control software keeps track of every modification to the code in a special kind of database. If a mistake is made, developers can turn back the clock and compare earlier versions of the code to help fix the mistake while minimizing disruption to all team members

Version control protects source code from both catastrophe and the casual degradation of human error and unintended consequences.

Version Control Systems (VCS) have seen great improvements over the past few decades and some are better than others. VCS are sometimes known as SCM (Source Code Management) tools or RCS (Revision Control System). One of the most popular VCS tools in use today is called Git. Git is a Distributed VCS, a category known as DVCS, more on that later. Like many of the most popular VCS systems available today, Git is free and open source. Regardless of what they are called, or which system is used, the primary benefits you should expect from version control are as follows.

1. A complete long-term change history of every file. This means every change made by many individuals over the years. Changes include the creation and deletion of files as well as edits to their contents. Different VCS tools differ on how well they handle renaming and moving of files. This history should also include the author, date and written notes on the purpose of each change. Having the complete history enables going back to previous versions to help in root cause analysis for bugs and it is crucial when needing to fix problems in older versions of software. If the software is being actively worked on, almost everything can be considered an "older version" of the software.
2. Branching and merging. Having team members work concurrently is a no-brainer, but even individuals working on their own can benefit from the ability to work on independent streams of changes. Creating a "branch" in VCS tools keeps multiple streams of work independent from each other while also providing the facility to merge that work back together, enabling developers to verify that the changes on each branch do not conflict. Many software teams adopt a practice of branching for each feature or perhaps branching for each release, or both. There are many different workflows that teams can choose from when they decide how to make use of branching and merging facilities in VCS.
3. Traceability. Being able to trace each change made to the software and connect it to project management and bug tracking software such as [Jira](https://www.atlassian.com/software/jira), and being able to annotate each change with a message describing the purpose and intent of the change can help not only with root cause analysis and other forensics. Having the annotated history of the code at your fingertips when you are reading the code, trying to understand what it is doing and why it is so designed can enable developers to make correct and harmonious changes that are in accord with the intended long-term design of the system. This can be especially important for working effectively with legacy code and is crucial in enabling developers to estimate future work with any accuracy.

# What is Git

By far, the most widely used modern version control system in the world today is Git. Git is a mature, actively maintained open source project originally developed in 2005 by Linus Torvalds, the famous creator of the Linux operating system kernel

Having a distributed architecture, Git is an example of a DVCS (hence Distributed Version Control System). Rather than have only one single place for the full version history of the software as is common in once-popular version control systems like CVS or Subversion (also known as SVN), in Git, every developer's working copy of the code is also a repository that can contain the full history of all changes.

In addition to being distributed, Git has been designed with performance, security and flexibility in mind.

## Performance

The raw performance characteristics of Git are very strong when compared to many alternatives. Committing new changes, branching, merging and comparing past versions are all optimized for performance. The algorithms implemented inside Git take advantage of deep knowledge about common attributes of real source code file trees, how they are usually modified over time and what the access patterns are.

Unlike some version control software, Git is not fooled by the names of the files when determining what the storage and version history of the file tree should be, instead, Git focuses on the file content itself. After all, source code files are frequently renamed, split, and rearranged. The object format of Git's repository files uses a combination of delta encoding (storing content differences), compression and explicitly stores directory contents and version metadata objects.

Being distributed enables significant performance benefits as well.

For example, say a developer, Alice, makes changes to source code, adding a feature for the upcoming 2.0 release, then commits those changes with descriptive messages. She then works on a second feature and commits those changes too. Naturally these are stored as separate pieces of work in the version history. Alice then switches to the version 1.3 branch of the same software to fix a bug that affects only that older version. The purpose of this is to enable Alice's team to ship a bug fix release, version 1.3.1, before version 2.0 is ready. Alice can then return to the 2.0 branch to continue working on new features for 2.0 and all of this can occur without any network access and is therefore fast and reliable. She could even do it on an airplane. When she is ready to send all of the individually committed changes to the remote repository, Alice can "push" them in one command.

## Security

Git has been designed with the integrity of managed source code as a top priority. The content of the files as well as the true relationships between files and directories, versions, tags and commits, all of these objects in the Git repository are secured with a cryptographically secure hashing algorithm called SHA1. This protects the code and the change history against both accidental and malicious change and ensures that the history is fully traceable.

With Git, you can be sure you have an authentic content history of your source code.

Some other version control systems have no protections against secret alteration at a later date. This can be a serious information security vulnerability for any organization that relies on software development.

## Flexibility

One of Git's key design objectives is flexibility. Git is flexible in several respects: in support for various kinds of nonlinear development workflows, in its efficiency in both small and large projects and in its compatibility with many existing systems and protocols.

Git has been designed to support branching and tagging as first-class citizens (unlike SVN) and operations that affect branches and tags (such as merging or reverting) are also stored as part of the change history. Not all version control systems feature this level of tracking.

## Version control with Git

Git is the best choice for most software teams today. While every team is different and should do their own analysis, here are the main reasons why version control with Git is preferred over alternatives:

### **Git is good**

Git has the functionality, performance, security and flexibility that most teams and individual developers need. These attributes of Git are detailed above. In side-by-side comparisons with most other alternatives, many teams find that Git is very favorable.

### **Git is a de facto standard**

Git is the most broadly adopted tool of its kind. This is makes Git attractive for the following reasons. At Atlassian, nearly all of our project source code is managed in Git.

Vast numbers of developers already have Git experience and a significant proportion of college graduates may have experience with only Git. While some organizations may need to climb the learning curve when migrating to Git from another version control system, many of their existing and future developers do not need to be trained on Git.

In addition to the benefits of a large talent pool, the predominance of Git also means that many third party software tools and services are already integrated with Git including IDEs, and our own tools like DVCS desktop client [Sourcetree](https://www.atlassian.com/software/sourcetree), issue and project tracking software, [Jira](https://www.atlassian.com/software/jira), and code hosting service, [Bitbucket](https://bitbucket.org/).

If you are an inexperienced developer wanting to build up valuable skills in software development tools, when it comes to version control, Git should be on your list.

### **Git is a quality open source project**

Git is a very well supported open source project with over a decade of solid stewardship. The project maintainers have shown balanced judgment and a mature approach to meeting the long term needs of its users with regular releases that improve usability and functionality. The quality of the open source software is easily scrutinized and countless businesses rely heavily on that quality.

Git enjoys great community support and a vast user base. Documentation is excellent and plentiful, including books, tutorials and dedicated web sites. There are also podcasts and video tutorials.

Being open source lowers the cost for hobbyist developers as they can use Git without paying a fee. For use in open-source projects, Git is undoubtedly the successor to the previous generations of successful open source version control systems, SVN and CVS.

### **Criticism of Git**

One common criticism of Git is that it can be difficult to learn. Some of the terminology in Git will be novel to newcomers and for users of other systems, the Git terminology may be different, for example, revert in Git has a different meaning than in SVN or CVS. Nevertheless, Git is very capable and provides a lot of power to its users. Learning to use that power can take some time, however once it has been learned, that power can be used by the team to increase their development speed.

For those teams coming from a non-distributed VCS, having a central repository may seem like a good thing that they don't want to lose. However, while Git has been designed as a distributed version control system (DVCS), with Git, you can still have an official, canonical repository where all changes to the software must be stored. With Git, because each developer's repository is complete, their work doesn't need to be constrained by the availability and performance of the "central" server. During outages or while offline, developers can still consult the full project history. Because Git is flexible as well as being distributed, you can work the way you are accustomed to but gain the additional benefits of Git, some of which you may not even realise you're missing.

Now that you understand what version control is, what Git is and why software teams should use it, read on to discover the benefits Git can provide across the whole organization.

## Git for developers

### **Feature Branch Workflow**

One of the biggest advantages of Git is its branching capabilities. Unlike centralized version control systems, Git branches are cheap and easy to merge. This facilitates the feature branch workflow popular with many Git users.

Feature branches provide an isolated environment for every change to your codebase. When a developer wants to start working on something—no matter how big or small—they create a new branch. This ensures that the master branch always contains production-quality code.

Using feature branches is not only more reliable than directly editing production code, but it also provides organizational benefits. They let you represent development work at the same granularity as the your [agile backlog](https://www.atlassian.com/agile/backlogs). For example, you might implement a policy where each [Jira](https://www.atlassian.com/software/jira) ticket is addressed in its own feature branch.

### **Distributed Development**

In SVN, each developer gets a working copy that points back to a single central repository. Git, however, is a distributed version control system. Instead of a working copy, each developer gets their own local repository, complete with a full history of commits.

Having a full local history makes Git fast, since it means you don’t need a network connection to create commits, inspect previous versions of a file, or perform diffs between commits.

Distributed development also makes it easier to scale your engineering team. If someone breaks the production branch in SVN, other developers can’t check in their changes until it’s fixed. With Git, this kind of blocking doesn’t exist. Everybody can continue going about their business in their own local repositories.

And, similar to feature branches, distributed development creates a more reliable environment. Even if a developer obliterates their own repository, they can simply clone someone else’s and start anew.

### **Pull Requests**

Many source code management tools such as [Bitbucket](https://bitbucket.org/) enhance core Git functionality with pull requests. A pull request is a way to ask another developer to merge one of your branches into their repository. This not only makes it easier for project leads to keep track of changes, but also lets developers initiate discussions around their work before integrating it with the rest of the codebase.

Since they’re essentially a comment thread attached to a feature branch, pull requests are extremely versatile. When a developer gets stuck with a hard problem, they can open a pull request to ask for help from the rest of the team. Alternatively, junior developers can be confident that they aren’t destroying the entire project by treating pull requests as a formal code review.

### **Community**

In many circles, Git has come to be the expected version control system for new projects. If your team is using Git, odds are you won’t have to train new hires on your workflow, because they’ll already be familiar with distributed development.

In addition, Git is very popular among open source projects. This means it’s easy to leverage 3rd-party libraries and encourage others to fork your own open source code.

### **Faster Release Cycle**

The ultimate result of feature branches, distributed development, pull requests, and a stable community is a faster release cycle. These capabilities facilitate an [agile workflow](https://www.atlassian.com/agile/workflow) where developers are encouraged to share smaller changes more frequently. In turn, changes can get pushed down the deployment pipeline faster than the monolithic releases common with centralized version control systems.

As you might expect, Git works very well with continuous integration and continuous delivery environments. Git hooks allow you to run scripts when certain events occur inside of a repository, which lets you automate deployment to your heart’s content. You can even build or deploy code from specific branches to different servers.

For example, you might want to configure Git to deploy the most recent commit from the develop branch to a test server whenever anyone merges a pull request into it. Combining this kind of build automation with peer review means you have the highest possible confidence in your code as it moves from development to staging to production.

# 1 Setting up a repository

[git init](https://www.atlassian.com/git/tutorials/setting-up-a-repository/git-init) [git clone](https://www.atlassian.com/git/tutorials/setting-up-a-repository/git-clone) [git config](https://www.atlassian.com/git/tutorials/setting-up-a-repository/git-config)

This tutorial provides an overview of how to set up a repository (repo) under Git version control. This resource will walk you through initializing a Git repository for a new or existing project. Included below are workflow examples of repositories both created locally and cloned from remote repositories. This guide assumes a basic familiarity with a command-line interface.

The high level points this guide will cover are:

* Initializing a new Git repo
* Cloning an existing Git repo
* Committing a modified version of a file to the repo
* Configuring a Git repo for remote collaboration
* Common Git version control commands

By the end of this module, you should be able to create a Git repo, use common Git commands, commit a modified file, view your project’s history and configure a connection to a Git hosting service (Bitbucket).

## Initializing a new repository: git init

To create a new repo, you'll use the git init command. git init is a one-time command you use during the initial setup of a new repo. Executing this command will create a new .git subdirectory in your current working directory. This will also create a new master branch.

### Versioning an existing project with a new git repository

This example assumes you already have an existing project folder that you would like to create a repo within. You'll first cd to the root project folder and then execute the git init command.

cd /path/to/your/existing/code

git init

Pointing git init to an existing project directory will execute the same initialization setup as mentioned above, but scoped to that project directory.

git init <project directory>

Visit the [git init](http://www.atlassian.com/git/tutorials/setting-up-a-repository/git-init) page for a more detailed resource on git init.

## Cloning an existing repository: git clone

If a project has already been set up in a central repository, the clone command is the most common way for users to obtain a local development clone. Like git init, cloning is generally a one-time operation. Once a developer has obtained a working copy, all [version control](http://bitbucket-marketing.atlassian.com/product/version-control-software) operations are managed through their local repository.

git clone <repo url>

## Saving changes to the repository: git add and git commit

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.

## Repo-to-repo collaboration: git push

It’s important to understand that Git’s idea of a “working copy” is very different from the working copy you get by checking out source code from an SVN repository. Unlike SVN, Git makes no distinction between the working copies and the central repository—they're all full-fledged [Git repositories](http://bitbucket-marketing.atlassian.com/product/code-repository).

This makes collaborating with Git fundamentally different than with SVN. Whereas SVN depends on the relationship between the central repository and the working copy, Git’s collaboration model is based on repository-to-repository interaction. Instead of checking a working copy into SVN’s central repository, you push or pull commits from one repository to another.

Of course, there’s nothing stopping you from giving certain Git repos special meaning. For example, by simply designating one Git repo as the “central” repository, it’s possible to replicate a centralized workflow using Git. This is accomplished through conventions rather than being hardwired into the VCS itself.

### Bare vs. cloned repositories

If you used git clone in the previous "Initializing a new Repository" section to set up your local repository, your repository is already configured for remote collaboration. git clone will automatically configure your repo with a remote pointed to the Git URL you cloned it from. This means that once you make changes to a file and commit them, you can git push those changes to the remote repository.

If you used git init to make a fresh repo, you'll have no remote repo to push changes to. A common pattern when initializing a new repo is to go to a hosted Git service like Bitbucket and create a repo there. The service will provide a Git URL that you can then add to your local Git repository and git push to the hosted repo. Once you have created a remote repo with your service of choice you will need to update your local repo with a mapping. We discuss this process in the Configuration & Set Up guide below.

If you prefer to host your own remote repo, you'll need to set up a "Bare Repository." Both git init and git clone accept a --bare argument. The most common use case for bare repo is to create a remote central Git repository

## Configuration & set up: git config

Once you have a remote repo setup, you will need to add a remote repo url to your local git config, and set an upstream branch for your local branches. The git remote command offers such utility.

git remote add <remote\_name> <remote\_repo\_url>

This command will map remote repository at <remote\_repo\_url> to a ref in your local repo under <remote\_name>. Once you have mapped the remote repo you can push local branches to it.

git push -u <remote\_name> <local\_branch\_name>

This command will push the local repo branch under <local\_branc\_name> to the remote repo at <remote\_name>.

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.

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

# 2 git init

This page will explore the git init command in depth. By the end of this page you will be informed on the core functionality and extended feature set of git init. This exploration includes:

* git init options and usage
* .git directory overview
* custom git init directory environment values
* git init vs. git clone
* git init bare repositories
* git init templates

The git init command creates a new Git repository. It can be used to convert an existing, unversioned project to a Git repository or initialize a new, empty repository. Most other Git commands are not available outside of an initialized repository, so this is usually the first command you'll run in a new project.

Executing git init creates a .git subdirectory in the current working directory, which contains all of the necessary Git metadata for the new repository. This metadata includes subdirectories for objects, refs, and template files. A HEAD file is also created which points to the currently checked out commit.

Aside from the .git directory, in the root directory of the project, an existing project remains unaltered (unlike SVN, Git doesn't require a .git subdirectory in every subdirectory).

By default, git init will initialize the Git configuration to the .git subdirectory path.

## Usage

Compared to SVN, the git init command is an incredibly easy way to create new version-controlled projects. Git doesn’t require you to create a repository, import files, and check out a working copy. Additionally, Git does not require any pre-existing server or admin privileges. All you have to do is cd into your project subdirectory and run git init, and you'll have a fully functional Git repository.

git init

Transform the current directory into a Git repository. This adds a .git subdirectory to the current directory and makes it possible to start recording revisions of the project.

git init <directory>

Create an empty Git repository in the specified directory. Running this command will create a new subdirectory called containing nothing but the .git subdirectory.

If you've already run git init on a project directory and it contains a .git subdirectory, you can safely run git init again on the same project directory. It will not override an existing .git configuration.

### git init vs. git clone

A quick note: git init and git clone can be easily confused. At a high level, they can both be used to "initialize a new git repository." However, git clone is dependent on git init. git clone is used to create a copy of an existing repository. Internally, git clone first calls git init to create a new repository. It then copies the data from the existing repository, and checks out a new set of working files. Learn more on the [git clone page](https://www.atlassian.com/git/tutorials/setting-up-a-repository/git-clone).

## Bare repositories --- git init --bare

git init --bare <directory>

Initialize an empty Git repository, but omit the working directory. Shared repositories should always be created with the --bare flag (see discussion below). Conventionally, repositories initialized with the --bare flag end in .git. For example, the bare version of a repository called my-project should be stored in a directory called my-project.git.

The --bare flag creates a repository that doesn’t have a working directory, making it impossible to edit files and commit changes in that repository. You would create a bare repository to git push and git pull from, but never directly commit to it. Central repositories should always be created as bare repositories because pushing branches to a non-bare repository has the potential to overwrite changes. Think of --bare as a way to mark a repository as a storage facility, as opposed to a development environment. This means that for virtually all Git workflows, the central repository is bare, and developers local repositories are non-bare.

## Configuration

All configurations of git init <directory> take a <directory> argument. If you provide the <directory>, the command is run inside it. If this directory does not exist, it will be created. In addition to the options and configuration already discussed, Git init has a few other command line options. A full list of them follows:

# 3 git clone

Here we'll examine the git clone command in depth. git clone is a Git command line utility which is used to target an existing repository and create a clone, or copy of the target repository. In this page we'll discuss extended configuration options and common use cases of git clone. Some points we'll cover here are:

* Cloning a local or remote repository
* Cloning a bare repository
* Using shallow options to partially clone repositories
* Git URL syntax and supported protocols

On the [setting up a repository guide](https://www.atlassian.com/git/tutorials/setting-up-a-repository), we covered a basic use case of git clone. This page will explore more complex cloning and configuration scenarios.

## Purpose: repo-to-repo collaboration development copy

If a project has already been set up in a central repository, the git clone command is the most common way for users to obtain a development copy. Like git init, cloning is generally a one-time operation. Once a developer has obtained a working copy, all version control operations and collaborations are managed through their local repository

### Repo-to-repo collaboration

It’s important to understand that Git’s idea of a “working copy” is very different from the working copy you get by checking out code from an SVN repository. Unlike SVN, Git makes no distinction between the working copy and the central repository—they're all full-fledged Git repositories.

This makes collaborating with Git fundamentally different than with SVN. Whereas SVN depends on the relationship between the central repository and the working copy, Git’s collaboration model is based on repository-to-repository interaction. Instead of checking a working copy into SVN’s central repository, you [push](https://www.atlassian.com/git/tutorials/syncing/git-push) or [pull](https://www.atlassian.com/git/tutorials/syncing/git-pull) commits from one repository to another

Of course, there’s nothing stopping you from giving certain Git repos special meaning. For example, by simply designating one Git repo as the “central” repository, it’s possible to replicate a [centralized workflow](https://www.atlassian.com/git/tutorials/comparing-workflows) using Git. The point is, this is accomplished through conventions rather than being hardwired into the VCS itself.

## Usage

git clone is primarily used to point to an existing repo and make a clone or copy of that repo at in a new directory, at another location. The original repository can be located on the local filesystem or on remote machine accessible supported protocols. The git clone command copies an existing Git repository. This is sort of like SVN checkout, except the “working copy” is a full-fledged Git repository—it has its own history, manages its own files, and is a completely isolated environment from the original repository.

As a convenience, cloning automatically creates a remote connection called "origin" pointing back to the original repository. This makes it very easy to interact with a central repository. This automatic connection is established by creating Git refs to the remote branch heads under refs/remotes/origin and by initializing remote.origin.url and remote.origin.fetch configuration variables.

git clone ssh://john@example.com/path/to/my-project.git

cd my-project

# Start working on the project

The first command initializes a new Git repository in the my-project folder on your local machine and populates it with the contents of the central repository. Then, you can cd into the project and start editing files, committing snapshots, and interacting with other repositories. Also note that the .git extension is omitted from the cloned repository. This reflects the non-bare status of the local copy.

### Cloning to a specific folder

git clone <repo> <directory>

Clone the repository located at <repo> into the folder called ~<directory>! on the local machine.

### Cloning a specific tag

git clone -branch <tag> <repo>

Clone the repository located at <repo> and only clone the ref for <tag>.

### Shallow clone

git clone -depth=1 <repo>

Clone the repository located at <repo> and only clone the   
history of commits specified by the option depth=1. In this example a clone of <repo> is made and only the most recent commit is included in the new cloned Repo. Shallow cloning is most useful when working with repos that have an extensive commit history. An extensive commit history may cause scaling problems such as disk space usage limits and long wait times when cloning. A Shallow clone can help alleviate these scaling issues.

## Configuration options

### git clone -branch

The -branch argument lets you specify a specific a branch to clone instead of the branch the remote HEAD is pointing to, usually the master branch. In addition you can pass a tag instead of branch for the same effect.

git clone -branch new\_feature git://remoterepository.git

This above example would clone only the new\_feature branch from the remote Git repository. This is purely a convince utility to save you time from downloading the HEAD ref of the repository and then having to additionally fetch the ref you need.

### git clone -mirror vs. git clone -bare

#### git clone --bare

Similar to git init --bare, when the -bare argument is passed to git clone, a copy of the remote repository will be made with an omitted working directory. This means that a repository will be set up with the history of the project that can be pushed and pulled from, but cannot be edited directly. In addition, no remote branches for the repo will be configured with the -bare repository. Like git init --bare, this is used to create a hosted repository that developers will not edit directly.

#### git clone --mirror

Passing the --mirror argument implicitly passes the --bare argument as well. This means the behavior of --bare is inherited by --mirror. Resulting in a bare repo with no editable working files. In addition, --mirror will clone all the extended refs of the remote repository, and maintain remote branch tracking configuration. You can then run git remote update on the mirror and it will overwrite all refs from the origin repo. Giving you exact 'mirrored' functionality.

# 4 git config

In this document, we'll take an in-depth look at the git config command. We briefly discussed git config usage on our [Setting up a Repository](https://www.atlassian.com/git/tutorials/setting-up-a-repository) page. 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. We'll be covering common configuration settings like email, username, and editor. We'll discuss Git aliases, which allow you to create shortcuts for frequently used Git operations. Becoming familiar with git config and the various Git configuration settings will help you create a powerful, customized Git workflow.

## Usage

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. For example: user.email

git config user.email

In this example, email is a child property of the user configuration block. This will return the configured email address, if any, that Git will associate with locally created commits.

### git config levels and files

Before we further discuss git config usage, let's take a moment to cover configuration levels. 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. ~ /.gitconfig on unix systems and 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. This means when looking for a configuration value, Git will start at the local level and bubble up to the system level.

### 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
* How to customize git colors

Overall, git config is a helper tool that provides a shortcut to editing raw git config files on disk. We covered in depth personal customization options. Basic knowledge of git configuration options is a prerequisite for [setting up a repository](https://www.atlassian.com/git/tutorials/setting-up-a-repository). See our guide there for a demonstration of the basics.

# 5 Git Alias

This section will focus on Git aliases. To better understand the value of Git aliases we must first discuss what an alias is. The term alias is synonymous with a shortcut. Alias creation is a common pattern found in other popular utilities like `bash` shell. Aliases are used to create shorter commands that map to longer commands. Aliases enable more efficient workflows by requiring fewer keystrokes to execute a command. For a brief example, consider the git checkout command. The checkout command is a frequently used git command, which adds up in cumulative keystrokes over time. An alias can be created that maps git co to git checkout, which saves precious human fingertip power by allowing the shorter keystroke form: git co to be typed instead.

## Git Alias Overview

It is important to note that there is no direct git alias command. Aliases are created through the use of the [git config](https://www.atlassian.com/git/tutorials/setting-up-a-repository/git-config) command and the Git configuration files. As with other configuration values, aliases can be created in a local or global scope.  
  
To better understand Git aliases let us create some examples.

$ git config --global alias.co checkout  
$ git config --global alias.br branch  
$ git config --global alias.ci commit  
$ git config --global alias.st status

The previous code example creates globally stored shortcuts for common git commands. Creating the aliases will not modify the source commands. So git checkout will still be available even though we now have the git co alias. These aliases were created with the --global flag which means they will be stored in Git's global operating system level configuration file. On linux systems, the global config file is located in the User home directory at /.gitconfig.

    [alias]  
        co = checkout  
            br = branch  
            ci = commit  
            st = status

This demonstrates that the aliases are now equivalent to the source commands.

## Usage

Git aliasing is enabled through the use of git config, For command-line option and usage examples please review the [git config](https://www.atlassian.com/git/tutorials/setting-up-a-repository/git-config) documentation.

## Examples

### Using aliases to create new Git commands

A common Git pattern is to remove recently added files from the staging area. This is achieved by leveraging options to the git reset command. A new alias can be created to encapsulate this behavior and create a new alias-command-keyword which is easy to remember:

git config --global alias.unstage 'reset HEAD --'

The preceding code example creates a new alias unstage. This now enables the invocation of git unstage. git unstage which will perform a reset on the staging area. This makes the following two commands equivalent.

git unstage fileA  
$ git reset HEAD -- fileA

## Discussion

### How do I create Git Aliases?

Aliases can be created through two primary methods:

#### Directly editing Git config files

The global or local config files can be manually edited and saved to create aliases. The global config file lives at $HOME/.gitconfig file path. The local path lives within an active git repository at /.git/config  
  
The config files will respect an [alias] section that looks like:

[alias]  
 co = checkout

This means that co is a shortcut for checkout

#### Using the git config to create aliases

As previously demonstrated the git config command is a convenient utility to quickly create aliases. The git config command is actually a helper utility for writing to the global and local Git config files.

git config --global alias.co checkout

Invoking this command will update the underlying global config file just as it had been edited in our previous example.

## Git Alias Summary

Git aliases are a powerful workflow tool that create shortcuts to frequently used Git commands. Using Git aliases will make you a faster and more efficient developer. Aliases can be used to wrap a sequence of Git commands into new faux Git command. Git aliases are created through the use of the git config command which essentially modifies local or global Git config files. Learn more on the [git config](https://www.atlassian.com/git/tutorials/setting-up-a-repository/git-config) page.

# 6 Saving changes

[git add](https://www.atlassian.com/git/tutorials/saving-changes) [git commit](https://www.atlassian.com/git/tutorials/saving-changes/git-commit) [git diff](https://www.atlassian.com/git/tutorials/saving-changes/git-diff) [git stash](https://www.atlassian.com/git/tutorials/saving-changes/git-stash) [.gitignore](https://www.atlassian.com/git/tutorials/saving-changes/gitignore)

When working in Git, or other version control systems, the concept of "saving" is a more nuanced process than saving in a word processor or other traditional file editing applications. The traditional software expression of "saving" is synonymous with the Git term "committing". A commit is the Git equivalent of a "save". Traditional saving should be thought of as a file system operation that is used to overwrite an existing file or write a new file. Alternatively, Git committing is an operation that acts upon a collection of files and directories.

Saving changes in Git vs SVN is also a different process. SVN Commits or 'check-ins' are operations that make a remote push to a centralized server. This means an SVN commit needs Internet access in order to fully 'save' project changes. 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 difference between the two methods is a fundamental difference between architecture designs. Git is a distributed application model whereas SVN is a centralized model. Distributed applications are generally more robust as they do not have a single point of failure like a centralized server.

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.

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.

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.

## git add

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.

## How it works

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, regardless of their team’s collaboration model. They are the means to record versions of a project into the repository’s history.

Developing a project revolves around the basic edit/stage/commit pattern. First, you edit your files in the working directory. When you’re ready to save a copy of the current state of the project, you stage changes with git add. After you’re happy with the staged snapshot, you commit it to the project history with git commit. The [git reset](https://www.atlassian.com/git/tutorials/undoing-changes/git-reset) command is used to undo a commit or staged snapshot.

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.

The git add command should not be confused with svn add, which adds a file to the repository. Instead, git add works on the more abstract level of changes. This means that git add needs to be called every time you alter a file, whereas svn add only needs to be called once for each file. It may sound redundant, but this workflow makes it much easier to keep a project organized.

## The staging area

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.

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.

## Common options

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.

## Examples

When you’re starting a new project, git add serves the same function as svn import. To create an initial commit of the current directory, use the following two commands:

git add .

git commit

Once you’ve got your project up-and-running, new files can be added by passing the path to git add:

git add hello.py

git commit

The above commands can also be used to record changes to existing files. Again, Git doesn’t differentiate between staging changes in new files vs. changes in files that have already been added to the repository.

## Summary

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 diff

[git add](https://www.atlassian.com/git/tutorials/saving-changes) [git commit](https://www.atlassian.com/git/tutorials/saving-changes/git-commit) [git diff](https://www.atlassian.com/git/tutorials/saving-changes/git-diff) [git stash](https://www.atlassian.com/git/tutorials/saving-changes/git-stash) [.gitignore](https://www.atlassian.com/git/tutorials/saving-changes/gitignore)

## 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

## Summary

This page disscused the Git diffing process and the git diff command. We discussed how to read git diff output and the various data included in the output. Examples were provided on how to alter the git diff output with highlighting and colors. We discussed different diffing strategies such as how to diff files in branches and specific commits. In addition to the git diff command, we also used git log and git checkout.

# Git stash

[git add](https://www.atlassian.com/git/tutorials/saving-changes) [git commit](https://www.atlassian.com/git/tutorials/saving-changes/git-commit) [git diff](https://www.atlassian.com/git/tutorials/saving-changes/git-diff) [git stash](https://www.atlassian.com/git/tutorials/saving-changes/git-stash) [.gitignore](https://www.atlassian.com/git/tutorials/saving-changes/gitignore)

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.

* Git Stash
  + [Stashing your work](https://www.atlassian.com/git/tutorials/saving-changes/git-stash#stashing-your-work)
  + [Re-applying your stashed changes](https://www.atlassian.com/git/tutorials/saving-changes/git-stash#re-applying-your-stashed-changes)
  + [Stashing untracked or ignored files](https://www.atlassian.com/git/tutorials/saving-changes/git-stash#stashing-untracked-or-ignored)
  + [Managing multiple stashes](https://www.atlassian.com/git/tutorials/saving-changes/git-stash#managing-multiple-stashes)
  + [Viewing stash diffs](https://www.atlassian.com/git/tutorials/saving-changes/git-stash#viewing-stash-diffs)
  + [Partial stashes](https://www.atlassian.com/git/tutorials/saving-changes/git-stash#partial-stashes)
  + [Creating a branch from your stash](https://www.atlassian.com/git/tutorials/saving-changes/git-stash#creating-a-branch-from-your-stash)
  + [Cleaning up your stash](https://www.atlassian.com/git/tutorials/saving-changes/git-stash#cleaning-up-your-stash)
  + [How git stash works](https://www.atlassian.com/git/tutorials/saving-changes/git-stash#how-git-stash-works)

## 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. For example:

$ git status

On branch master

Changes to be committed:

new file: style.css

Changes not staged for commit:

modified: index.html

$ git stash

Saved working directory and index state WIP on master: 5002d47 our new homepage

HEAD is now at 5002d47 our new homepage

$ git status

On branch master

nothing to commit, working tree clean

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.

## Re-applying your stashed changes

You can reapply previously stashed changes with git stash pop:

$ git status

On branch master

nothing to commit, working tree clean

$ git stash pop

On branch master

Changes to be committed:

new file: style.css

Changes not staged for commit:

modified: index.html

Dropped refs/stash@{0} (32b3aa1d185dfe6d57b3c3cc3b32cbf3e380cc6a)

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:

$ git stash apply

On branch master

Changes to be committed:

new file: style.css

Changes not staged for commit:

modified: index.html

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.



## Managing multiple stashes

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

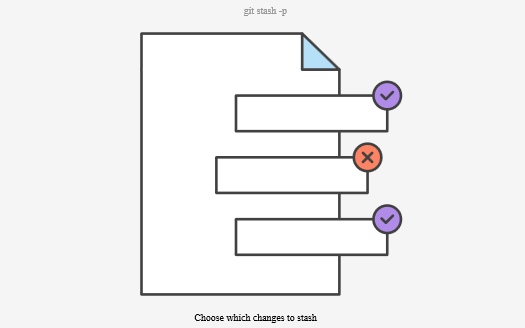
--- a/index.html

+++ b/index.html

@@ -1 +1,2 @@

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

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



## 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

|/

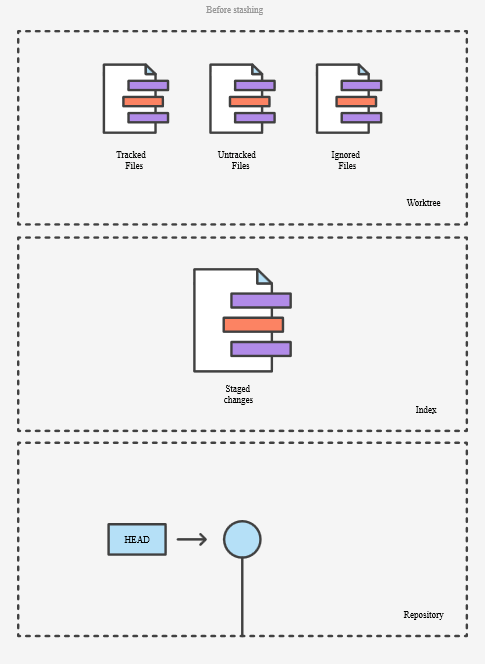
\* 5002d47 our new homepage

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

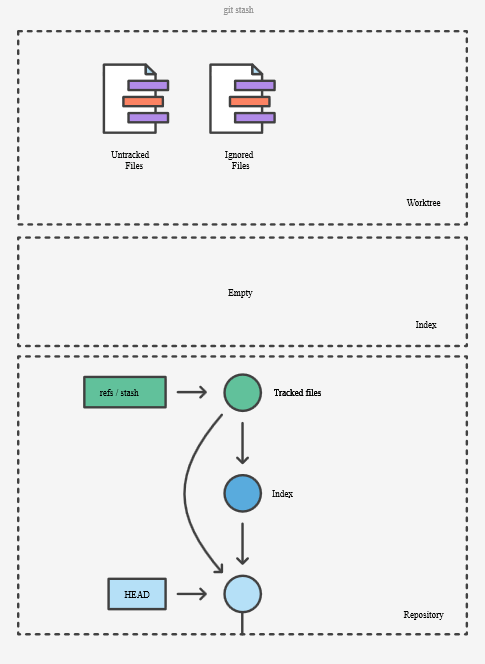
* stash@{0}, a new commit to store the tracked files that were in your working copy when you ran git stash
* stash@{0}'s first parent, the pre-existing commit that was at HEAD when you ran git stash
* stash@{0}'s second parent, a new commit representing the index when you ran git stash
* stash@{0}'s 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:

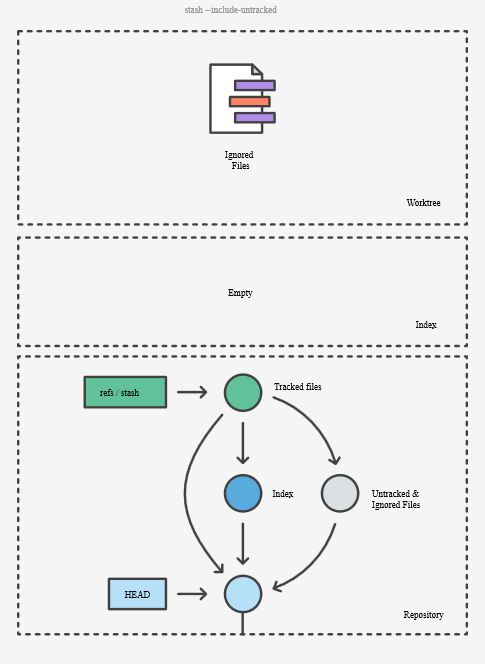
* Before stashing, your worktree may contain changes to tracked files, untracked files, and ignored files. Some of these changes may also be staged in the index.



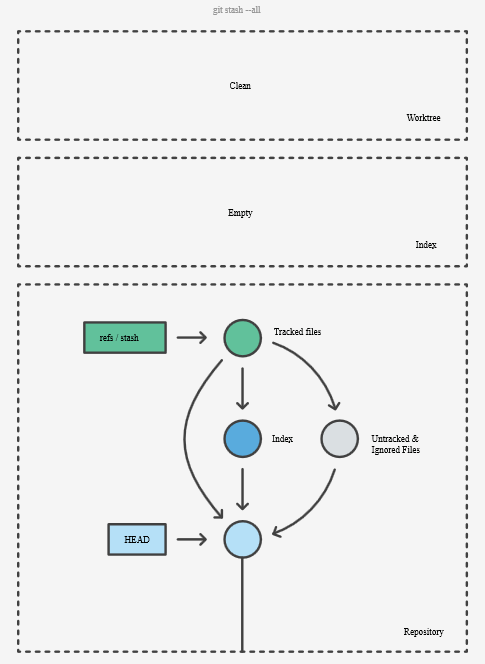
Invoking git stash encodes any changes to tracked files as two new commits in your DAG: one for unstaged changes, and one for changes staged in the index. The special refs/stash ref is updated to point to them.



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



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



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.

# .gitignore

[git add](https://www.atlassian.com/git/tutorials/saving-changes) [git commit](https://www.atlassian.com/git/tutorials/saving-changes/git-commit) [git diff](https://www.atlassian.com/git/tutorials/saving-changes/git-diff) [git stash](https://www.atlassian.com/git/tutorials/saving-changes/git-stash) [.gitignore](https://www.atlassian.com/git/tutorials/saving-changes/gitignore)

Git sees every file in your working copy as one of three things:

1. tracked - a file which has been previously staged or committed;
2. untracked - a file which has not been staged or committed; or
3. ignored - a file which Git has been explicitly told to ignore.

Ignored files are usually build artifacts and machine generated files that can be derived from your repository source or should otherwise not be committed. Some common examples are:

* dependency caches, such as the contents of /node\_modules or /packages
* compiled code, such as .o, .pyc, and .class files
* build output directories, such as /bin, /out, or /target
* files generated at runtime, such as .log, .lock, or .tmp
* hidden system files, such as .DS\_Store or Thumbs.db
* personal IDE config files, such as .idea/workspace.xml

Ignored files are tracked in a special file named .gitignore that is checked in at the root of your repository. There is no explicit git ignore command: instead the .gitignore file must be edited and committed by hand when you have new files that you wish to ignore. .gitignore files contain patterns that are matched against file names in your repository to determine whether or not they should be ignored.

* Ignoring files in Git
  + [Git ignore patterns](https://www.atlassian.com/git/tutorials/saving-changes/gitignore#git-ignore-patterns)
  + [Shared .gitignore files in your repository](https://www.atlassian.com/git/tutorials/saving-changes/gitignore#shared)
  + [Personal Git ignore rules](https://www.atlassian.com/git/tutorials/saving-changes/gitignore#personal-git-ignore-rules)
  + [Global Git ignore rules](https://www.atlassian.com/git/tutorials/saving-changes/gitignore#global-git-ignore-rules)
  + [Ignoring a previously committed file](https://www.atlassian.com/git/tutorials/saving-changes/gitignore#ignoring-a-previously-committed)
  + [Committing an ignored file](https://www.atlassian.com/git/tutorials/saving-changes/gitignore#committing-an-ignored-file)
  + [Stashing an ignored file](https://www.atlassian.com/git/tutorials/saving-changes/gitignore#stashing-an-ignored-file)
  + [Debugging .gitignore files](https://www.atlassian.com/git/tutorials/saving-changes/gitignore#debugging)

## Git ignore patterns

.gitignore uses [globbing patterns](http://linux.die.net/man/7/glob) to match against file names. You can construct your patterns using various symbols:

| **Pattern** | **Example matches** | **Explanation\*** |
| --- | --- | --- |
| \*\*/logs | logs/debug.log logs/monday/foo.bar build/logs/debug.log | You can prepend a pattern with a double asterisk to match directories anywhere in the repository. |
| \*\*/logs/debug.log | logs/debug.log build/logs/debug.log but not logs/build/debug.log | You can also use a double asterisk to match files based on their name and the name of their parent directory. |
| \*.log | debug.log foo.log .log logs/debug.log | An asterisk is a wildcard that matches zero or more characters. |
| \*.log  !important.log | debug.log trace.log but not important.log logs/important.log | Prepending an exclamation mark to a pattern negates it. If a file matches a pattern, but also matches a negating pattern defined later in the file, it will not be ignored. |
| \*.log  !important/\*.log trace.\* | debug.log important/trace.log but not important/debug.log | Patterns defined after a negating pattern will re-ignore any previously negated files. |
| /debug.log | debug.log but not logs/debug.log | Prepending a slash matches files only in the repository root. |
| debug.log | debug.log logs/debug.log | By default, patterns match files in any directory |
| debug?.log | debug0.log debugg.log but not debug10.log | A question mark matches exactly one character. |
| debug[0-9].log | debug0.log debug1.log but not debug10.log | Square brackets can also be used to match a single character from a specified range. |
| debug[01].log | debug0.log debug1.log but not  debug2.log debug01.log | Square brackets match a single character form the specified set. |
| debug[!01].log | debug2.log but not debug0.log debug1.log debug01.log | An exclamation mark can be used to match any character except one from the specified set. |
| debug[a-z].log | debuga.log debugb.log but not debug1.log | Ranges can be numeric or alphabetic. |
| logs | logs logs/debug.log logs/latest/foo.bar build/logs build/logs/debug.log | If you don't append a slash, the pattern will match both files and the contents of directories with that name. In the example matches on the left, both directories and files named logs are ignored |
| logs/ | logs/debug.log logs/latest/foo.bar build/logs/foo.bar build/logs/latest/debug.log | Appending a slash indicates the pattern is a directory. The entire contents of any directory in the repository matching that name – including all of its files and subdirectories – will be ignored |
| logs/  !logs/important.log | logs/debug.log logs/important.log | Wait a minute! Shouldn't logs/important.log be negated in the example on the left  Nope! Due to a performance-related quirk in Git, you can not negate a file that is ignored due to a pattern matching a directory |
| logs/\*\*/debug.log | logs/debug.log logs/monday/debug.log logs/monday/pm/debug.log | A double asterisk matches zero or more directories. |
| logs/\*day/debug.log | logs/monday/debug.log logs/tuesday/debug.log but not logs/latest/debug.log | Wildcards can be used in directory names as well. |
| logs/debug.log | logs/debug.log but not debug.log build/logs/debug.log | Patterns specifying a file in a particular directory are relative to the repository root. (You can prepend a slash if you like, but it doesn't do anything special.) |

\*\* these explanations assume your .gitignore file is in the top level directory of your repository, as is the convention. If your repository has multiple .gitignore files, simply mentally replace "repository root" with "directory containing the .gitignore file" (and consider unifying them, for the sanity of your team).\*

In addition to these characters, you can use # to include comments in your .gitignore file:

# ignore all logs

\*.log

You can use \ to escape .gitignore pattern characters if you have files or directories containing them:

# ignore the file literally named foo[01].txt

foo\[01\].txt

## Shared .gitignore files in your repository

Git ignore rules are usually defined in a .gitignore file at the root of your repository. However, you can choose to define multiple .gitignore files in different directories in your repository. Each pattern in a particular .gitignore file is tested relative to the directory containing that file. However the convention, and simplest approach, is to define a single .gitignore file in the root. As your .gitignore file is checked in, it is versioned like any other file in your repository and shared with your teammates when you push. Typically you should only include patterns in .gitignore that will benefit other users of the repository.

## Personal Git ignore rules

You can also define personal ignore patterns for a particular repository in a special file at .git/info/exclude. These are not versioned, and not distributed with your repository, so it's an appropriate place to include patterns that will likely only benefit you. For example if you have a custom logging setup, or special development tools that produce files in your repository's working directory, you could consider adding them to .git/info/exclude to prevent them from being accidentally committed to your repository.

## Global Git ignore rules

In addition, you can define global Git ignore patterns for all repositories on your local system by setting the Git core.excludesFile property. You'll have to create this file yourself. If you're unsure where to put your global .gitignore file, your home directory isn't a bad choice (and makes it easy to find later). Once you've created the file, you'll need to configure its location with git config:

$ touch ~/.gitignore

$ git config --global core.excludesFile ~/.gitignore

You should be careful what patterns you choose to globally ignore, as different file types are relevant for different projects. Special operating system files (e.g. .DS\_Store and thumbs.db) or temporary files created by some developer tools are typical candidates for ignoring globally.

## Ignoring a previously committed file

If you want to ignore a file that you've committed in the past, you'll need to delete the file from your repository and then add a .gitignore rule for it. Using the --cached option with git rm means that the file will be deleted from your repository, but will remain in your working directory as an ignored file.

$ echo debug.log >> .gitignore

$ git rm --cached debug.log

rm 'debug.log'

$ git commit -m "Start ignoring debug.log"

You can omit the --cached option if you want to delete the file from both the repository and your local file system.

## Committing an ignored file

It is possible to force an ignored file to be committed to the repository using the -f (or --force) option with git add:

$ cat .gitignore

\*.log

$ git add -f debug.log

$ git commit -m "Force adding debug.log"

You might consider doing this if you have a general pattern (like \*.log) defined, but you want to commit a specific file. However a better solution is to define an exception to the general rule:

$ echo !debug.log >> .gitignore

$ cat .gitignore

\*.log

!debug.log

$ git add debug.log

$ git commit -m "Adding debug.log"

This approach is more obvious, and less confusing, for your teammates.

## Stashing an ignored file

[git stash](https://www.atlassian.com/git/tutorials/git-stash) is a powerful Git feature for temporarily shelving and reverting local changes, allowing you to re-apply them later on. As you'd expect, by default git stash ignores ignored files and only stashes changes to files that are tracked by Git. However, you can invoke [git stash with the --all option](https://www.atlassian.com/git/tutorials/git-stash/#stashing-untracked-or-ignored) to stash changes to ignored and untracked files as well.

## Debugging .gitignore files

If you have complicated .gitignore patterns, or patterns spread over multiple .gitignore files, it can be difficult to track down why a particular file is being ignored. You can use the git check-ignore command with the -v (or --verbose) option to determine which pattern is causing a particular file to be ignored:

$ git check-ignore -v debug.log

.gitignore:3:\*.log debug.log

The output shows:

<file containing the pattern> : <line number of the pattern> : <pattern> <file name>

You can pass multiple file names to git check-ignore if you like, and the names themselves don't even have to correspond to files that exist in your repository.

# Git Status: Inspecting a repository

[git status](https://www.atlassian.com/git/tutorials/inspecting-a-repository) [git tag](https://www.atlassian.com/git/tutorials/inspecting-a-repository/git-tag) [git blame](https://www.atlassian.com/git/tutorials/inspecting-a-repository/git-blame)

## 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. Status output does not show you any information regarding the committed project history. For this, you need to use [git log](https://www.atlassian.com/git/tutorials/inspecting-a-repository/git-log).

### Related git commands

* [git tag](https://www.atlassian.com/git/tutorials/inspecting-a-repository/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).
* [**git blame**](https://www.atlassian.com/git/tutorials/inspecting-a-repository/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.
* [git log](https://www.atlassian.com/git/tutorials/inspecting-a-repository/git-tag)
  + The git log command displays committed snapshots. It lets you list the project history, filter it, and search for specific changes.

### Usage

git status

List which files are staged, unstaged, and untracked.

### Discussion

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:

#### 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

### Example

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. This example displays the repository status before and after staging and committing a snapshot:

# Edit hello.py

git status

# hello.py is listed under "Changes not staged for commit"

git add hello.py

git status

# hello.py is listed under "Changes to be committed"

git commit

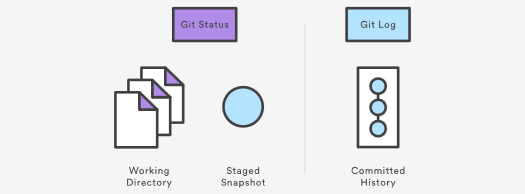
git status

# nothing to commit (working directory clean)

The first status output will show the file as unstaged. The git add action will be reflected in the second git status, and the final status output will tell you that there is nothing to commit—the working directory matches the most recent commit. Some Git commands (e.g., [git merge](https://www.atlassian.com/git/tutorials/using-branches/git-merge)) require the working directory to be clean so that you don't accidentally overwrite changes.

## 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.



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.

### 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.

it 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

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.

### 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 tag

[git status](https://www.atlassian.com/git/tutorials/inspecting-a-repository) [git tag](https://www.atlassian.com/git/tutorials/inspecting-a-repository/git-tag) [git blame](https://www.atlassian.com/git/tutorials/inspecting-a-repository/git-blame)

## 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.

# Git blame

[git status](https://www.atlassian.com/git/tutorials/inspecting-a-repository) [git tag](https://www.atlassian.com/git/tutorials/inspecting-a-repository/git-tag) [git blame](https://www.atlassian.com/git/tutorials/inspecting-a-repository/git-blame)

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.

# Undoing Commits & Changes

[git checkout](https://www.atlassian.com/git/tutorials/undoing-changes) [git clean](https://www.atlassian.com/git/tutorials/undoing-changes/git-clean) [git revert](https://www.atlassian.com/git/tutorials/undoing-changes/git-revert) [git reset](https://www.atlassian.com/git/tutorials/undoing-changes/git-reset) [git rm](https://www.atlassian.com/git/tutorials/undoing-changes/git-rm)

In this section, we will discuss the available 'undo' Git strategies and commands. It is first important to note that Git does not have a traditional 'undo' system like those found in a word processing application. It will be beneficial to refrain from mapping Git operations to any traditional 'undo' mental model. Additionally, Git has its own nomenclature for 'undo' operations that it is best to leverage in a discussion. This nomenclature includes terms like reset, revert, checkout, clean, and more.

A fun metaphor is to think of Git as a timeline management utility. Commits are snapshots of a point in time or points of interest along the timeline of a project's history. Additionally, multiple timelines can be managed through the use of branches. When 'undoing' in Git, you are usually moving back in time, or to another timeline where mistakes didn't happen.

This tutorial provides all of the necessary skills to work with previous revisions of a software project. First, it shows you how to explore old commits, then it explains the difference between reverting public commits in the project history vs. resetting unpublished changes on your local machine.

## Finding what is lost: Reviewing old commits

The whole idea behind any version control system is to store “safe” copies of a project so that you never have to worry about irreparably breaking your code base. Once you’ve built up a project history of commits, you can review and revisit any commit in the history. One of the best utilities for reviewing the history of a Git repository is the git log command. In the example below, we use [git log](https://www.atlassian.com/git/tutorials/git-log) to get a list of the latest commits to a popular open-source graphics library.

git log --oneline

e2f9a78fe Replaced FlyControls with OrbitControls

d35ce0178 Editor: Shortcuts panel Safari support.

9dbe8d0cf Editor: Sidebar.Controls to Sidebar.Settings.Shortcuts. Clean up.

05c5288fc Merge pull request #12612 from TyLindberg/editor-controls-panel

0d8b6e74b Merge pull request #12805 from harto/patch-1

23b20c22e Merge pull request #12801 from gam0022/improve-raymarching-example-v2

fe78029f1 Fix typo in documentation

7ce43c448 Merge pull request #12794 from WestLangley/dev-x

17452bb93 Merge pull request #12778 from OndrejSpanel/unitTestFixes

b5c1b5c70 Merge pull request #12799 from dhritzkiv/patch-21

1b48ff4d2 Updated builds.

88adbcdf6 WebVRManager: Clean up.

2720fbb08 Merge pull request #12803 from dmarcos/parentPoseObject

9ed629301 Check parent of poseObject instead of camera

219f3eb13 Update GLTFLoader.js

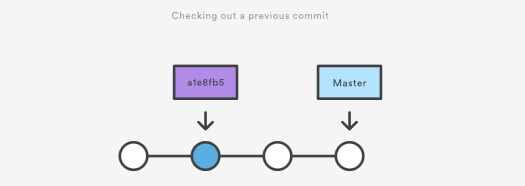
15f13bb3c Update GLTFLoader.js

6d9c22a3b Update uniforms only when onWindowResize

881b25b58 Update ProjectionMatrix on change aspect

Each commit has a unique SHA-1 identifying hash. These IDs are used to travel through the committed timeline and revisit commits. By default, git log will only show commits for the currently selected branch. It is entirely possible that the commit you're looking for is on another branch. You can view all commits across all branches by executing git log --branches=\*. The command [git branch](https://www.atlassian.com/git/tutorials/using-branches) is used to view and visit other branches. Invoking the command, git branch -a will return a list of all known branch names. One of these branch names can then be logged using git log <branch\_name>.

When you have found a commit reference to the point in history you want to visit, you can utilize the git checkout command to visit that commit. Git checkout is an easy way to “load” any of these saved snapshots onto your development machine. During the normal course of development, the HEAD usually points to master or some other local branch, but when you check out a previous commit, HEAD no longer points to a branch—it points directly to a commit. This is called a “detached HEAD” state, and it can be visualized as the following:



Checking out an old file does not move the HEAD pointer. It remains on the same branch and same commit, avoiding a 'detached head' state. You can then commit the old version of the file in a new snapshot as you would any other changes. So, in effect, this usage of git checkout on a file, serves as a way to revert back to an old version of an individual file. For more information on these two modes visit the [git checkout](https://www.atlassian.com/git/tutorials/using-branches/git-checkout) page

## Viewing an old revision

This example assumes that you’ve started developing a crazy experiment, but you’re not sure if you want to keep it or not. To help you decide, you want to take a look at the state of the project before you started your experiment. First, you’ll need to find the ID of the revision you want to see.

git log --oneline

Let’s say your project history looks something like the following:

b7119f2 Continue doing crazy things

872fa7e Try something crazy

a1e8fb5 Make some important changes to hello.txt

435b61d Create hello.txt

9773e52 Initial import

You can use git checkout to view the “Make some import changes to hello.txt” commit as follows:

git checkout a1e8fb5

This makes your working directory match the exact state of the a1e8fb5 commit. You can look at files, compile the project, run tests, and even edit files without worrying about losing the current state of the project. Nothing you do in here will be saved in your repository. To continue developing, you need to get back to the “current” state of your project:

git checkout master

This assumes that you're developing on the default master branch. Once you’re back in the master branch, you can use either [git revert](https://www.atlassian.com/git/tutorials/undoing-changes/git-revert)or [git reset](https://www.atlassian.com/git/tutorials/undoing-changes/git-reset) to undo any undesired changes.

## Undoing a committed snapshot

There are technically several different strategies to 'undo' a commit. The following examples will assume we have a commit history that looks like:

git log --oneline

872fa7e Try something crazy

a1e8fb5 Make some important changes to hello.txt

435b61d Create hello.txt

9773e52 Initial import

We will focus on undoing the 872fa7e Try something crazy commit. Maybe things got a little too crazy.

## How to undo a commit with git checkout

Using the git checkout command we can checkout the previous commit, a1e8fb5, putting the repository in a state before the crazy commit happened. Checking out a specific commit will put the repo in a "detached HEAD" state. This means you are no longer working on any branch. In a detached state, any new commits you make will be orphaned when you change branches back to an established branch. Orphaned commits are up for deletion by Git's garbage collector. The garbage collector runs on a configured interval and permanently destroys orphaned commits. To prevent orphaned commits from being garbage collected, we need to ensure we are on a branch.

From the detached HEAD state, we can execute git checkout -b new\_branch\_without\_crazy\_commit. This will create a new branch named new\_branch\_without\_crazy\_commit and switch to that state. The repo is now on a new history timeline in which the 872fa7e commit no longer exists. At this point, we can continue work on this new branch in which the 872fa7e commit no longer exists and consider it 'undone'. Unfortunately, if you need the previous branch, maybe it was your master branch, this undo strategy is not appropriate. Let's look at some other 'undo' strategies. For more information and examples review our in-depth [git checkout](https://www.atlassian.com/git/tutorials/using-branches/git-checkout) discussion.

## How to undo a public commit with git revert

Let's assume we are back to our original commit history example. The history that includes the 872fa7e commit. This time let's try a revert 'undo'. If we execute git revert HEAD, Git will create a new commit with the inverse of the last commit. This adds a new commit to the current branch history and now makes it look like:

git log --oneline

e2f9a78 Revert "Try something crazy"

872fa7e Try something crazy

a1e8fb5 Make some important changes to hello.txt

435b61d Create hello.txt

9773e52 Initial import

At this point, we have again technically 'undone' the 872fa7e commit. Although 872fa7e still exists in the history, the new e2f9a78 commit is an inverse of the changes in 872fa7e. Unlike our previous checkout strategy, we can continue using the same branch. This solution is a satisfactory undo. This is the ideal 'undo' method for working with public shared repositories. If you have requirements of keeping a curated and minimal Git history this strategy may not be satisfactory.

## How to undo a commit with git reset

For this undo strategy we will continue with our working example. [git reset](https://www.atlassian.com/git/tutorials/undoing-changes/git-reset) is an extensive command with multiple uses and functions. If we invoke git reset --hard a1e8fb5 the commit history is reset to that specified commit. Examining the commit history with git log will now look like:

git log --oneline

a1e8fb5 Make some important changes to hello.txt

435b61d Create hello.txt

9773e52 Initial import

The log output shows the e2f9a78 and 872fa7e commits no longer exist in the commit history. At this point, we can continue working and creating new commits as if the 'crazy' commits never happened. This method of undoing changes has the cleanest effect on history. Doing a reset is great for local changes however it adds complications when working with a shared remote repository. If we have a shared remote repository that has the 872fa7e commit pushed to it, and we try to git push a branch where we have reset the history, Git will catch this and throw an error. Git will assume that the branch being pushed is not up to date because of it's missing commits. In these scenarios, git revert should be the preferred undo method.

## Undoing the last commit

In the previous section, we discussed different strategies for undoing commits. These strategies are all applicable to the most recent commit as well. In some cases though, you might not need to remove or reset the last commit. Maybe it was just made prematurely. In this case you can amend the most recent commit. Once you have made more changes in the working directory and staged them for commit by using [git add](https://www.atlassian.com/git/tutorials/saving-changes), you can execute git commit --amend. This will have Git open the configured system editor and let you modify the last commit message. The new changes will be added to the amended commit.

## Undoing uncommitted changes

Before changes are committed to the repository history, they live in the staging index and the working directory. You may need to undo changes within these two areas. The staging index and working directory are internal Git state management mechanisms. For more detailed information on how these two mechanisms operate, visit the [git reset](https://www.atlassian.com/git/tutorials/resetting-checking-out-and-reverting) page which explores them in depth.

## The working directory

The working directory is generally in sync with the local file system. To undo changes in the working directory you can edit files like you normally would using your favorite editor. Git has a couple utilities that help manage the working directory. There is the [git clean](https://www.atlassian.com/git/tutorials/undoing-changes/git-clean) command which is a convenience utility for undoing changes to the working directory. Additionally, git reset can be invoked with the --mixed or --hard options and will apply a reset to the working directory.

## The staging index

The [git add](https://www.atlassian.com/git/tutorials/saving-changes) command is used to add changes to the staging index. Git reset is primarily used to undo the staging index changes. A --mixed reset will move any pending changes from the staging index back into the working directory.

## Undoing public changes

When working on a team with remote repositories, extra consideration needs to be made when undoing changes. Git reset should generally be considered a 'local' undo method. A reset should be used when undoing changes to a private branch. This safely isolates the removal of commits from other branches that may be in use by other developers. Problems arise when a reset is executed on a shared branch and that branch is then pushed remotely with git push. Git will block the push in this scenario complaining that the branch being pushed is out of date from the remote branch as it is missing commits.

The preferred method of undoing shared history is git revert. A revert is safer than a reset because it will not remove any commits from a shared history. A revert will retain the commits you want to undo and create a new commit that inverts the undesired commit. This method is safer for shared remote collaboration because a remote developer can then pull the branch and receive the new revert commit which undoes the undesired commit.

## Summary

We covered many high-level strategies for undoing things in Git. It's important to remember that there is more than one way to 'undo' in a Git project. Most of the discussion on this page touched on deeper topics that are more thoroughly explained on pages specific to the relevant Git commands. The most commonly used 'undo' tools are [git checkout,](https://www.atlassian.com/git/tutorials/using-branches/git-checkout) [git revert](https://www.atlassian.com/git/tutorials/undoing-changes/git-revert), and [git reset](https://www.atlassian.com/git/tutorials/undoing-changes/git-reset). Some key points to remember are:

* Once changes have been committed they are generally permanent
* Use git checkout to move around and review the commit history
* git revert is the best tool for undoing shared public changes
* git reset is best used for undoing local private changes

In addition to the primary undo commands, we took a look at other Git utilities: [git log](https://www.atlassian.com/git/tutorials/git-log) for finding lost commits [git clean](https://www.atlassian.com/git/tutorials/undoing-changes/git-clean) for undoing uncommitted changes [git add](https://www.atlassian.com/git/tutorials/saving-changes) for modifying the staging index.

Each of these commands has its own in-depth documentation. To learn more about a specific command mentioned here, visit the corresponding links.

# Git Clean

[git checkout](https://www.atlassian.com/git/tutorials/undoing-changes) [git clean](https://www.atlassian.com/git/tutorials/undoing-changes/git-clean) [git revert](https://www.atlassian.com/git/tutorials/undoing-changes/git-revert) [git reset](https://www.atlassian.com/git/tutorials/undoing-changes/git-reset) [git rm](https://www.atlassian.com/git/tutorials/undoing-changes/git-rm)

In this section, we will focus on a detailed discussion of the git clean command. Git clean is to some extent an 'undo' command. Git clean can be considered complementary to other commands like [git reset](https://www.atlassian.com/git/tutorials/undoing-changes/git-reset) and [git checkout](https://www.atlassian.com/git/tutorials/using-branches/git-checkout). Whereas these other commands operate on files previously added to the Git tracking index, the git clean command operates on untracked files. Untracked files are files that have been created within your repo's working directory but have not yet been added to the repository's tracking index using the [git add](https://www.atlassian.com/git/tutorials/saving-changes) command. To better demonstrate the difference between tracked and untracked files consider the following command line example:

$ mkdir git\_clean\_test

$ cd git\_clean\_test/

$ git init .

Initialized empty Git repository in /Users/kev/code/git\_clean\_test/.git/

$ echo "tracked" > ./tracked\_file

$ git add ./tracked\_file

$ echo "untracked" > ./untracked\_file

$ mkdir ./untracked\_dir && touch ./untracked\_dir/file

$ git status

On branch master

Initial commit

Changes to be committed: (use "git rm --cached <file>..." to unstage)

new file: tracked\_file

Untracked files: (use "git add <file>..." to include in what will be committed) untracked\_dir/ untracked\_file

The example creates a new Git repository in the git\_clean\_test directory. It then proceeds to create a tracked\_file which is added to the Git index, additionally, an untracked\_file is created, and an untracked\_dir. The example then invokes git status which displays output indicating Git's internal state of tracked and untracked changes. With the repository in this state, we can execute the git clean command to demonstrate its intended purpose.

$ git clean fatal: clean.requireForce defaults to true and neither -i, -n, nor -f given; refusing to clean

At this point, executing the default git clean command may produce a fatal error. The example above demonstrates what this may look like. By default, Git is globally configured to require that git clean be passed a "force" option to initiate. This is an important safety mechanism. When finally executed git clean is not undo-able. When fully executed, git clean will make a hard filesystem deletion, similar to executing the command line rm utility. Make sure you really want to delete the untracked files before you run it.

## Common options and usage

Given the previous explanation of the default git clean behaviors and caveats, the following content demonstrates various git clean use cases and the accompanying command line options required for their operation.

-n

The -n option will perform a “dry run” of git clean. This will show you which files are going to be removed without actually removing them. It is a best practice to always first perform a dry run of git clean. We can demonstrate this option in the demo repo we created earlier.

$ git clean -n

Would remove untracked\_file

The output tells us that untracked\_file will be removed when the git clean command is executed. Notice that the untracked\_dir is not reported in the output here. By default git clean will not operate recursively on directories. This is another safety mechanism to prevent accidental permanent deletion.

-f or --force

The force option initiates the actual deletion of untracked files from the current directory. Force is required unless the clean.requireForce configuration option is set to false. This will not remove untracked folders or files specified by .gitignore. Let us now execute a live git clean in our example repo.

$ git clean -f

Removing untracked\_file

The command will output the files that are removed. You can see here that untracked\_file has been removed. Executing git status at this point or doing a ls will show that untracked\_file has been deleted and is nowhere to be found. By default git clean -f will operate on all the current directory untracked files. Additionally, a <path> value can be passed with the -f option that will remove a specific file.

git clean -f <path>

-d include directories

The -d option tells git clean that you also want to remove any untracked directories, by default it will ignore directories. We can add the -d option to our previous examples:

$ git clean -dn

Would remove untracked\_dir/

$ git clean -df

Removing untracked\_dir/

Here we have executed a 'dry run' using the -dn combination which outputs untracked\_dir is up for removal. Then we execute a forced clean, and receive output that untracked\_dir is removed.

-x force removal of ignored files

A common software release pattern is to have a build or distribution directory that is not committed to the repositories tracking index. The build directory will contain ephemeral build artifacts that are generated from the committed source code. This build directory is usually added to the repositories .gitignore file. It can be convenient to also clean this directory with other untracked files. The -x option tells git clean to also include any ignored files. As with previous git clean invocations, it is a best practice to execute a 'dry run' first, before the final deletion. The -x option will act on all ignored files, not just project build specific ones. This could be unintended things like ./.idea IDE configuration files.

git clean -xf

Like the -d option -x can be passed and composed with other options. This example demonstrates a combination with -f that will remove untracked files from the current directory as well as any files that Git usually ignores.

## Interactive mode or git clean interactive

In addition to the ad-hoc command line execution we have demonstrated so far, git clean has an "interactive" mode that you can initiate by passing the -i option. Let us revisit the example repo from the introduction of this document. In that initial state, we will start an interactive clean session.

$ git clean -di

Would remove the following items:

untracked\_dir/ untracked\_file

\*\*\* Commands \*\*\*

1: clean 2: filter by pattern 3: select by numbers 4: ask each 5: quit 6: help

What now>

We have initiated the interactive session with the -d option so it will also act upon our untracked\_dir. The interactive mode will display a What now> prompt that requests a command to apply to the untracked files. The commands themselves are fairly self explanatory. We'll take a brief look at each in a random order starting with command 6: help. Selecting command 6 will further explain the other commands:

What now> 6

clean - start cleaning

filter by pattern - exclude items from deletion

select by numbers - select items to be deleted by numbers

ask each - confirm each deletion (like "rm -i")

quit - stop cleaning

help - this screen

? - help for prompt selection

5: quit

Is straight forward and will exit the interactive session.

1: clean

Will delete the indicated items. If we were to execute 1: clean at this point untracked\_dir/ untracked\_file would be removed.

4: ask each

will iterate over each untracked file and display a Y/N prompt for a deletion. It looks like the following:

\*\*\* Commands \*\*\*

1: clean 2: filter by pattern 3: select by numbers 4: ask each 5: quit 6: help

What now> 4

Remove untracked\_dir/ [y/N]? N

Remove untracked\_file [y/N]? N

2: filter by pattern

Will display an additional prompt that takes input used to filter the list of untracked files.

Would remove the following items:

untracked\_dir/ untracked\_file

\*\*\* Commands \*\*\*

1: clean 2: filter by pattern 3: select by numbers 4: ask each 5: quit 6: help

What now> 2

untracked\_dir/ untracked\_file

Input ignore patterns>> \*\_file

untracked\_dir/

Here we input the \*\_file wildcard pattern which then restricts the untracked file list to just untracked\_dir.

3: select by numbers

Similar to command 2, command 3 works to refine the list of untracked file names. The interactive session will prompt for numbers that correspond to an untracked file name.

Would remove the following items:

untracked\_dir/ untracked\_file

\*\*\* Commands \*\*\*

1: clean 2: filter by pattern 3: select by numbers 4: ask each 5: quit 6: help

What now> 3

1: untracked\_dir/ 2: untracked\_file

Select items to delete>> 2

1: untracked\_dir/ \* 2: untracked\_file

Select items to delete>>

Would remove the following item:

untracked\_file

\*\*\* Commands \*\*\*

1: clean 2: filter by pattern 3: select by numbers 4: ask each 5: quit 6: help

## Summary

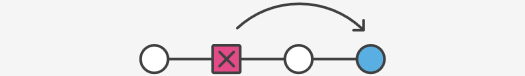
To recap, git clean is a convenience method for deleting untracked files in a repo's working directory. Untracked files are those that are in the repo's directory but have not yet been added to the repo's index with [git add](https://www.atlassian.com/git/tutorials/saving-changes). Overall the effect of git clean can be accomplished using [git status](https://www.atlassian.com/git/tutorials/inspecting-a-repository) and the operating systems native deletion tools. Git clean can be used alongside [git reset](https://www.atlassian.com/git/tutorials/undoing-changes/git-reset) to fully undo any additions and commits in a repository.

# Git Revert

[git checkout](https://www.atlassian.com/git/tutorials/undoing-changes) [git clean](https://www.atlassian.com/git/tutorials/undoing-changes/git-clean) [git revert](https://www.atlassian.com/git/tutorials/undoing-changes/git-revert) [git reset](https://www.atlassian.com/git/tutorials/undoing-changes/git-reset) [git rm](https://www.atlassian.com/git/tutorials/undoing-changes/git-rm)

The git revert command can be considered an 'undo' type command, however, it is not a traditional undo operation. Instead of removing the commit from the project history, it figures out how to invert the changes introduced by the commit and appends a new commit with the resulting inverse content. This prevents Git from losing history, which is important for the integrity of your revision history and for reliable collaboration.

Reverting should be used when you want to apply the inverse of a commit from your project history. This can be useful, for example, if you’re tracking down a bug and find that it was introduced by a single commit. Instead of manually going in, fixing it, and committing a new snapshot, you can use git revert to automatically do all of this for you.



## How it works

The git revert command is used for undoing changes to a repository's commit history. Other 'undo' commands like, [git checkout](https://www.atlassian.com/git/tutorials/using-branches/git-checkout) and [git reset](https://www.atlassian.com/git/tutorials/undoing-changes/git-reset), move the HEAD and branch ref pointers to a specified commit. Git revert also takes a specified commit, however, git revert does not move ref pointers to this commit. A revert operation will take the specified commit, inverse the changes from that commit, and create a new "revert commit". The ref pointers are then updated to point at the new revert commit making it the tip of the branch.  
  
To demonstrate let’s create an example repo using the command line examples below:

$ mkdir git\_revert\_test

$ cd git\_revert\_test/

$ git init .

Initialized empty Git repository in /git\_revert\_test/.git/

$ touch demo\_file

$ git add demo\_file

$ git commit -am"initial commit"

[master (root-commit) 299b15f] initial commit

1 file changed, 0 insertions(+), 0 deletions(-)

create mode 100644 demo\_file

$ echo "initial content" >> demo\_file

$ git commit -am"add new content to demo file"

[master 3602d88] add new content to demo file

n 1 file changed, 1 insertion(+)

$ echo "prepended line content" >> demo\_file

$ git commit -am"prepend content to demo file"

[master 86bb32e] prepend content to demo file

1 file changed, 1 insertion(+)

$ git log --oneline

86bb32e prepend content to demo file

3602d88 add new content to demo file

299b15f initial commit

Here we have initialized a repo in a newly created directory named git\_revert\_test. We have made 3 commits to the repo in which we have added a file demo\_file and modified its content twice. At the end of the repo setup procedure, we invoke git log to display the commit history, showing a total of 3 commits. With the repo in this state, we are ready to initiate a git revert.

$ git revert HEAD

[master b9cd081] Revert "prepend content to demo file"

1 file changed, 1 deletion(-)

Git revert expects a commit ref was passed in and will not execute without one. Here we have passed in the HEAD ref. This will revert the latest commit. This is the same behavior as if we reverted to commit 3602d8815dbfa78cd37cd4d189552764b5e96c58. Similar to a merge, a revert will create a new commit which will open up the configured system editor prompting for a new commit message. Once a commit message has been entered and saved Git will resume operation. We can now examine the state of the repo using git log and see that there is a new commit added to the previous log:

$ git log --oneline

1061e79 Revert "prepend content to demo file"

86bb32e prepend content to demo file

3602d88 add new content to demo file

299b15f initial commit

Note that the 3rd commit is still in the project history after the revert. Instead of deleting it, git revert added a new commit to undo its changes. As a result, the 2nd and 4th commits represent the exact same code base and the 3rd commit is still in our history just in case we want to go back to it down the road.

## Common options

-e

--edit

This is a default option and doesn't need to be specified. This option will open the configured system editor and prompts you to edit the commit message prior to committing the revert.

--no-edit

This is the inverse of the -e option. The revert will not open the editor.

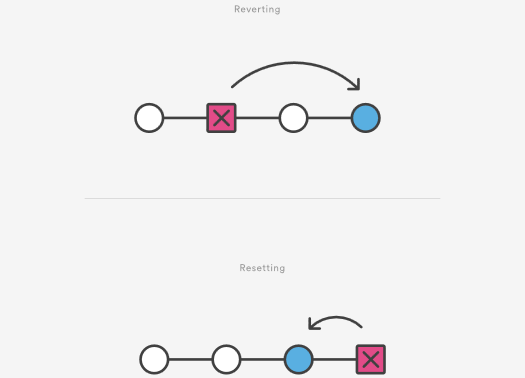
-n

--no-commit

Passing this option will prevent git revert from creating a new commit that inverses the target commit. Instead of creating the new commit this option will add the inverse changes to the Staging Index and Working Directory. These are the other trees Git uses to manage state the state of the repository. For more info visit the [git reset](https://www.atlassian.com/git/tutorials/undoing-changes/git-reset) page.

## Resetting vs. reverting

It's important to understand that git revert undoes a single commit—it does not "revert" back to the previous state of a project by removing all subsequent commits. In Git, this is actually called a reset, not a revert.



Reverting has two important advantages over resetting. First, it doesn’t change the project history, which makes it a “safe” operation for commits that have already been published to a shared repository. For details about why altering shared history is dangerous, please see the [git reset](https://www.atlassian.com/git/tutorials/undoing-changes/git-reset) page.  
  
Second, git revert is able to target an individual commit at an arbitrary point in the history, whereas git reset can only work backward from the current commit. For example, if you wanted to undo an old commit with git reset, you would have to remove all of the commits that occurred after the target commit, remove it, then re-commit all of the subsequent commits. Needless to say, this is not an elegant undo solution. For a more detailed discussion on the differences between git revert and other 'undo' commands see [Resetting, Checking Out and Reverting.](https://www.atlassian.com/git/tutorials/resetting-checking-out-and-reverting)

## Summary

The git revert command is a forward-moving undo operation that offers a safe method of undoing changes. Instead of deleting or orphaning commits in the commit history, a revert will create a new commit that inverses the changes specified. Git revert is a safer alternative to git reset in regards to losing work. To demonstrate the effects of git revert we leveraged other commands that have more in-depth documentation on their individual pages: [git log](https://www.atlassian.com/git/tutorials/git-log), [git commit](https://www.atlassian.com/git/tutorials/saving-changes#git-commit), and [git reset](https://www.atlassian.com/git/tutorials/undoing-changes/git-reset).

# Git Reset

[git checkout](https://www.atlassian.com/git/tutorials/undoing-changes) [git clean](https://www.atlassian.com/git/tutorials/undoing-changes/git-clean) [git revert](https://www.atlassian.com/git/tutorials/undoing-changes/git-revert) [git reset](https://www.atlassian.com/git/tutorials/undoing-changes/git-reset) [git rm](https://www.atlassian.com/git/tutorials/undoing-changes/git-reset)

The git reset command is a complex and versatile tool for undoing changes. It has three primary forms of invocation. These forms correspond to command line arguments --soft, --mixed, --hard. The three arguments each correspond to Git's three internal state management mechanism's, The Commit Tree (HEAD), The Staging Index, and The Working Directory.

## Git Reset & Three Trees of Git

To properly understand git reset usage, we must first understand Git's internal state management systems. Sometimes these mechanisms are called Git's "three trees". Trees may be a misnomer, as they are not strictly traditional tree data-structures. They are, however, node and pointer-based data structures that Git uses to track a timeline of edits. The best way to demonstrate these mechanisms is to create a changeset in a repository and follow it through the three trees.

To get started we will create a new repository with the commands below:

$ mkdir git\_reset\_test

$ cd git\_reset\_test/

$ git init .

Initialized empty Git repository in /git\_reset\_test/.git/

$ touch reset\_lifecycle\_file

$ git add reset\_lifecycle\_file

$ git commit -m"initial commit"

[master (root-commit) d386d86] initial commit

1 file changed, 0 insertions(+), 0 deletions(-)

create mode 100644 reset\_lifecycle\_file

The above example code creates a new git repository with a single empty file, reset\_lifecycle\_file. At this point, the example repository has a single commit (d386d86) from adding reset\_lifecycle\_file.

## The working directory

The first tree we will examine is "The Working Directory". This tree is in sync with the local filesystem and is representative of the immediate changes made to content in files and directories.

$ echo 'hello git reset' > reset\_lifecycle\_file

$ git status 

On branch master 

Changes not staged for commit: 

(use "git add ..." to update what will be committed) 

(use "git checkout -- ..." to discard changes in working directory) 

modified: reset\_lifecycle\_file

In our demo repository, we modify and add some content to the reset\_lifecycle\_file. Invoking git status shows that Git is aware of the changes to the file. These changes are currently a part of the first tree, "The Working Directory". Git status can be used to show changes to the Working Directory. They will be displayed in the red with a 'modified' prefix.

## Staging index

Next up is the 'Staging Index' tree. This tree is tracking Working Directory changes, that have been promoted with git add, to be stored in the next commit. This tree is a complex internal caching mechanism. Git generally tries to hide the implementation details of the Staging Index from the user.

To accurately view the state of the Staging Index we must utilize a lesser known Git command git ls-files. The git ls-files command is essentially a debug utility for inspecting the state of the Staging Index tree.

git ls-files -s

100644 e69de29bb2d1d6434b8b29ae775ad8c2e48c5391 0 reset\_lifecycle\_file

Here we have executed git ls-files with the -s or --stage option. Without the -s option the git ls-files output is simply a list of file names and paths that are currently part of the index. The -s option displays additional metadata for the files in the Staging Index. This metadata is the staged contents' mode bits, object name, and stage number. Here we are interested in the object name, the second value (d7d77c1b04b5edd5acfc85de0b592449e5303770). This is a standard Git object SHA-1 hash. It is a hash of the content of the files. The Commit History stores its own object SHA's for identifying pointers to commits and refs and the Staging Index has its own object SHA's for tracking versions of files in the index.

Next, we will promote the modified reset\_lifecycle\_file into the Staging Index.

$ git add reset\_lifecycle\_file

$ git status

On branch master Changes to be committed:

(use "git reset HEAD ..." to unstage)

modified: reset\_lifecycle\_file

Here we have invoked git add reset\_lifecycle\_file which adds the file to the Staging Index. Invoking git status now shows reset\_lifecycle\_file in green under "Changes to be committed". It is important to note that git status is not a true representation of the Staging Index. The git status command output displays changes between the Commit History and the Staging Index. Let us examine the Staging Index content at this point.

$ git ls-files -s

100644 d7d77c1b04b5edd5acfc85de0b592449e5303770 0 reset\_lifecycle\_file

We can see that the object SHA for reset\_lifecycle\_file has been updated from e69de29bb2d1d6434b8b29ae775ad8c2e48c5391 to d7d77c1b04b5edd5acfc85de0b592449e5303770.

## Commit history

The final tree is the Commit History. The git commit command adds changes to a permanent snapshot that lives in the Commit History. This snapshot also includes the state of the Staging Index at the time of commit.

$ git commit -am"update content of reset\_lifecycle\_file"

[master dc67808] update content of reset\_lifecycle\_file

1 file changed, 1 insertion(+)

$ git status

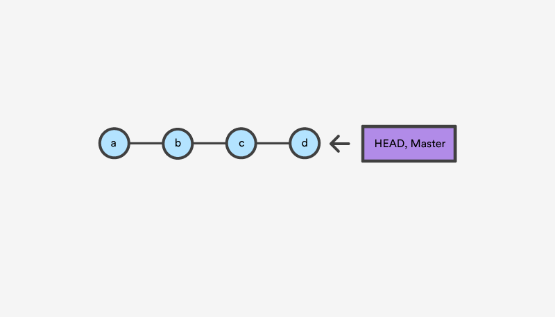
On branch master

nothing to commit, working tree clean

Here we have created a new commit with a message of "update content of resetlifecyclefile". The changeset has been added to the Commit History. Invoking git status at this point shows that there are no pending changes to any of the trees. Executing git log will display the Commit History. Now that we have followed this changeset through the three trees we can begin to utilize git reset.

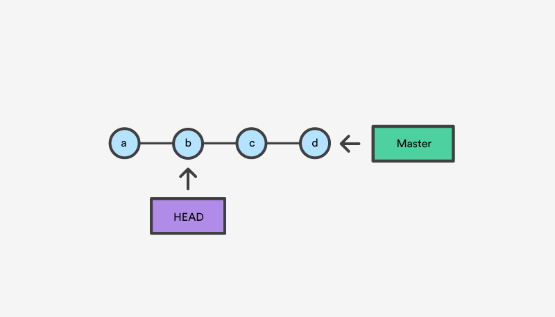
## How it works

At a surface level, git reset is similar in behavior to git checkout. Where git checkout solely operates on the HEAD ref pointer, git reset will move the HEAD ref pointer and the current branch ref pointer. To better demonstrate this behavior consider the following example:



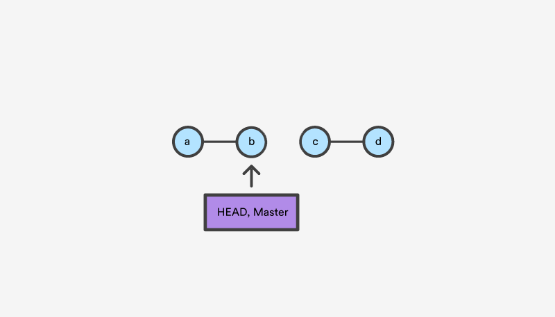
This example demonstrates a sequence of commits on the master branch. The HEAD ref and master branch ref currently point to commit d. Now let us execute and compare, both git checkout b and git reset b.

### git checkout b



With git checkout, the master ref is still pointing to d. The HEAD ref has been moved, and now points at commit b. The repo is now in a 'detached HEAD' state.

### git reset b

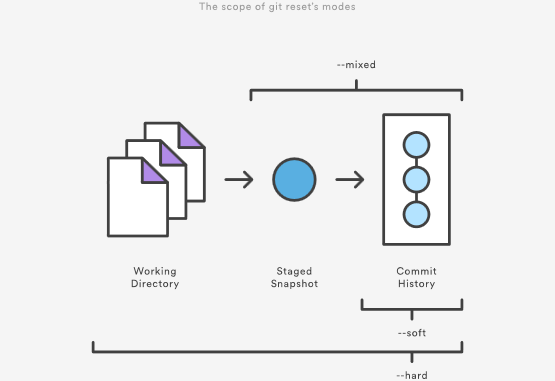


Comparatively, git reset, moves both the HEAD and branch refs to the specified commit.

In addition to updating the commit ref pointers, git reset will modify the state of the three trees. The ref pointer modification always happens and is an update to the third tree, the Commit tree. The command line arguments --soft, --mixed, and --hard direct how to modify the Staging Index, and Working Directory trees.

## Main Options

The default invocation of git reset has implicit arguments of --mixed and HEAD. This means executing git reset is equivalent to executing git reset --mixed HEAD. In this form HEAD is the specified commit. Instead of HEAD any Git SHA-1 commit hash can be used.



## --hard

This is the most direct, DANGEROUS, and frequently used option. When passed --hard The Commit History ref pointers are updated to the specified commit. Then, the Staging Index and Working Directory are reset to match that of the specified commit. Any previously pending changes to the Staging Index and the Working Directory gets reset to match the state of the Commit Tree. This means any pending work that was hanging out in the Staging Index and Working Directory will be lost.

To demonstrate this, let's continue with the three tree example repo we established earlier. First let's make some modifications to the repo. Execute the following commands in the example repo:

$ echo 'new file content' > new\_file

$ git add new\_file

$ echo 'changed content' >> reset\_lifecycle\_file

These commands have created a new file named new\_file and added it to the repo. Additionally, the content of reset\_lifecycle\_file will be modified. With these changes in place let us now examine the state of the repo using git status.

$ git status

On branch master

Changes to be committed:

(use "git reset HEAD ..." to unstage)

new file: new\_file

Changes not staged for commit:

(use "git add ..." to update what will be committed)

(use "git checkout -- ..." to discard changes in working directory)

modified: reset\_lifecycle\_file

We can see that there are now pending changes to the repo. The Staging Index tree has a pending change for the addition of new\_file and the Working Directory has a pending change for the modifications to reset\_lifecycle\_file.

Before moving forward let us also examine the state of the Staging Index:

$ git ls-files -s

100644 8e66654a5477b1bf4765946147c49509a431f963 0 new\_file

100644 d7d77c1b04b5edd5acfc85de0b592449e5303770 0 reset\_lifecycle\_file

We can see that new\_file has been added to the index. We have made updates to reset\_lifecycle\_file but the Staging Index SHA (d7d77c1b04b5edd5acfc85de0b592449e5303770) remains the same. This is expected behavior because have not used git add to promote these changes to the Staging Index. These changes exist in the Working Directory.

Let us now execute a git reset --hard and examine the new state of the repository.

$ git reset --hard

HEAD is now at dc67808 update content of reset\_lifecycle\_file

$ git status

On branch master

nothing to commit, working tree clean

$ git ls-files -s

100644 d7d77c1b04b5edd5acfc85de0b592449e5303770 0 reset\_lifecycle\_file

Here we have executed a "hard reset" using the --hard option. Git displays output indicating that HEAD is pointing to the latest commit dc67808. Next, we check the state of the repo with git status. Git indicates there are no pending changes. We also examine the state of the Staging Index and see that it has been reset to a point before new\_file was added. Our modifications to reset\_lifecycle\_file and the addition of new\_file have been destroyed. This data loss cannot be undone, this is critical to take note of.

## --mixed

This is the default operating mode. The ref pointers are updated. The Staging Index is reset to the state of the specified commit. Any changes that have been undone from the Staging Index are moved to the Working Directory. Let us continue.

$ echo 'new file content' > new\_file

$ git add new\_file

$ echo 'append content' >> reset\_lifecycle\_file

$ git add reset\_lifecycle\_file

$ git status

On branch master

Changes to be committed:

(use "git reset HEAD ..." to unstage)

new file: new\_file

modified: reset\_lifecycle\_file

$ git ls-files -s

100644 8e66654a5477b1bf4765946147c49509a431f963 0 new\_file

100644 7ab362db063f9e9426901092c00a3394b4bec53d 0 reset\_lifecycle\_file

In the example above we have made some modifications to the repository. Again, we have added a new\_file and modified the contents of reset\_lifecycle\_file. These changes are then applied to the Staging Index with git add. With the repo in this state, we will now execute the reset.

$ git reset --mixed

$ git status

On branch master

Changes not staged for commit:

(use "git add ..." to update what will be committed)

(use "git checkout -- ..." to discard changes in working directory)

modified: reset\_lifecycle\_file

Untracked files:

(use "git add ..." to include in what will be committed)

new\_file

no changes added to commit (use "git add" and/or "git commit -a")

$ git ls-files -s

100644 d7d77c1b04b5edd5acfc85de0b592449e5303770 0 reset\_lifecycle\_file

Here we have executed a "mixed reset". To reiterate, --mixed is the default mode and the same effect as executing git reset. Examining the output from git status and git ls-files, shows that the Staging Index has been reset to a state where reset\_lifecycle\_file is the only file in the index. The object SHA for reset\_lifecycle\_file has been reset to the previous version.

The important things to take note of here is that git status shows us that there are modifications to reset\_lifecycle\_file and there is an untracked file: new\_file. This is the explicit --mixed behavior. The Staging Index has been reset and the pending changes have been moved into the Working Directory. Compare this to the --hard reset case where the Staging Index was reset and the Working Directory was reset as well, losing these updates.

## --soft

When the --soft argument is passed, the ref pointers are updated and the reset stops there. The Staging Index and the Working Directory are left untouched. This behavior can be hard to clearly demonstrate. Let's continue with our demo repo and prepare it for a soft reset.

$ git add reset\_lifecycle\_file

$ git ls-files -s

100644 67cc52710639e5da6b515416fd779d0741e3762e 0 reset\_lifecycle\_file

$ git status

On branch master

Changes to be committed:

(use "git reset HEAD ..." to unstage)

modified: reset\_lifecycle\_file

Untracked files:

(use "git add ..." to include in what will be committed)

new\_file

Here we have again used git add to promote the modified reset\_lifecycle\_file into the Staging Index. We confirm that the index has been updated with the git ls-files output. The output from git status now displays the "Changes to be committed" in green. The new\_file from our previous examples is floating around in the Working Directory as an untracked file. Lets quickly execute rm new\_file to delete the file as we will not need it for the upcoming examples.

With the repository in this state we now execute a soft reset.

$ git reset --soft

$ git status

On branch master

Changes to be committed:

(use "git reset HEAD ..." to unstage)

modified: reset\_lifecycle\_file

$ git ls-files -s

100644 67cc52710639e5da6b515416fd779d0741e3762e 0 reset\_lifecycle\_file

We have executed a 'soft reset'. Examining the repo state with git status and git ls-files shows that nothing has changed. This is expected behavior. A soft reset will only reset the Commit History. By default, git reset is invoked with HEAD as the target commit. Since our Commit History was already sitting on HEAD and we implicitly reset to HEAD nothing really happened.

To better understand and utilize --soft we need a target commit that is not HEAD. We have reset\_lifecycle\_file waiting in the Staging Index. Let's create a new commit.

$ git commit -m"prepend content to reset\_lifecycle\_file"

At this point, our repo should have three commits. We will be going back in time to the first commit. To do this we will need the first commit's ID. This can be found by viewing output from git log.

$ git log

commit 62e793f6941c7e0d4ad9a1345a175fe8f45cb9df

Author: bitbucket

Date: Fri Dec 1 15:03:07 2017 -0800

prepend content to reset\_lifecycle\_file

commit dc67808a6da9f0dec51ed16d3d8823f28e1a72a

Author: bitbucket

Date: Fri Dec 1 10:21:57 2017 -0800

update content of reset\_lifecycle\_file

commit 780411da3b47117270c0e3a8d5dcfd11d28d04a4

Author: bitbucket

Date: Thu Nov 30 16:50:39 2017 -0800

initial commit

Keep in mind that Commit History ID's will be unique to each system. This means the commit ID's in this example will be different from what you see on your personal machine. The commit ID we are interested in for this example is 780411da3b47117270c0e3a8d5dcfd11d28d04a4. This is the ID that corresponds to the "initial commit". Once we have located this ID we will use it as the target for our soft reset.

Before we travel back in time lets first check the current state of the repo.

$ git status && git ls-files -s

On branch master

nothing to commit, working tree clean

100644 67cc52710639e5da6b515416fd779d0741e3762e 0 reset\_lifecycle\_file

Here we execute a combo command of git status and git ls-files -s this shows us there are pending changes to the repo and reset\_lifecycle\_file in the Staging Index is at a version of 67cc52710639e5da6b515416fd779d0741e3762e. With this in mind lets execute a soft reset back to our first commit.

$git reset --soft 780411da3b47117270c0e3a8d5dcfd11d28d04a4

$ git status && git ls-files -s

On branch master

Changes to be committed:

(use "git reset HEAD ..." to unstage)

modified: reset\_lifecycle\_file

100644 67cc52710639e5da6b515416fd779d0741e3762e 0 reset\_lifecycle\_file

The code above executes a "soft reset" and also invokes the git status and git ls-files combo command, which outputs the state of the repository. We can examine the repo state output and note some interesting observations. First, git status indicates there are modifications to reset\_lifecycle\_file and highlights them indicating they are changes staged for the next commit. Second, the git ls-files input indicates that the Staging Index has not changed and retains the SHA 67cc52710639e5da6b515416fd779d0741e3762e we had earlier.

To further clarify what has happened in this reset let us examine the git log:

$ git log

commit 780411da3b47117270c0e3a8d5dcfd11d28d04a4

Author: bitbucket

Date: Thu Nov 30 16:50:39 2017 -0800

initial commit

The log output now shows that there is a single commit in the Commit History. This helps to clearly illustrate what --soft has done. As with all git reset invocations, the first action reset takes is to reset the commit tree. Our previous examples with --hard and --mixed have both been against the HEAD and have not moved the Commit Tree back in time. During a soft reset, this is all that happens.

This may then be confusing as to why git status indicates there are modified files. --soft does not touch the Staging Index, so the updates to our Staging Index followed us back in time through the commit history. This can be confirmed by the output of git ls-files -s showing that the SHA for reset\_lifecycle\_file is unchanged. As a reminder, git status does not show the state of 'the three trees', it essentially shows a diff between them. In this case, it is displaying that the Staging Index is ahead of the changes in the Commit History as if we have already staged them.

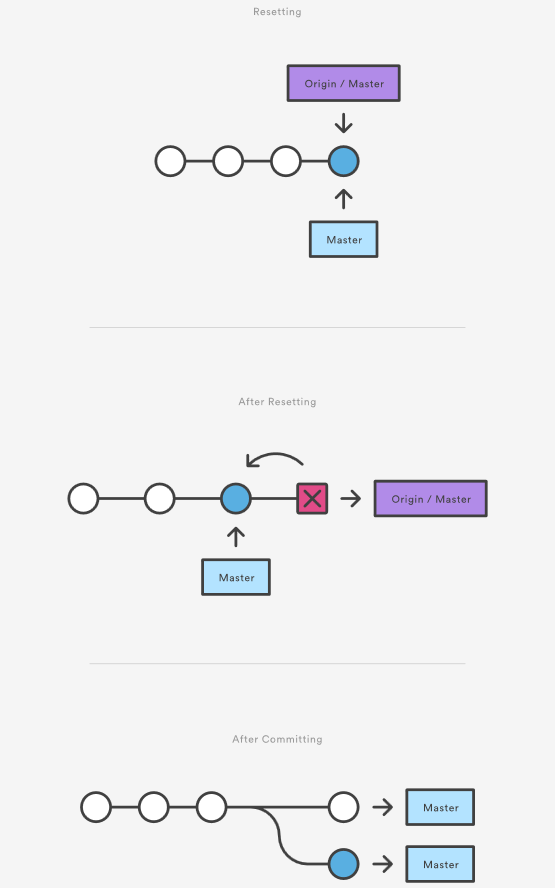
## Resetting vs Reverting

If [git revert](https://www.atlassian.com/git/tutorials/undoing-changes/git-revert) is a “safe” way to undo changes, you can think of git reset as the dangerous method. There is a real risk of losing work with git reset. Git reset will never delete a commit, however, commits can become 'orphaned' which means there is no direct path from a ref to access them. These orphaned commits can usually be found and restored using [git reflog](https://www.atlassian.com/git/tutorials/rewriting-history/git-reflog). Git will permanently delete any orphaned commits after it runs the internal garbage collector. By default, Git is configured to run the garbage collector every 30 days. Commit History is one of the 'three git trees' the other two, Staging Index and Working Directory are not as permanent as Commits. Care must be taken when using this tool, as it’s one of the only Git commands that have the potential to lose your work.  
  
Whereas reverting is designed to safely undo a public commit, git reset is designed to undo local changes to the Staging Index and Working Directory. Because of their distinct goals, the two commands are implemented differently: resetting completely removes a changeset, whereas reverting maintains the original changeset and uses a new commit to apply the undo.

## Don't Reset Public History

You should never use git reset <commit> when any snapshots after <commit> have been pushed to a public repository. After publishing a commit, you have to assume that other developers are reliant upon it.

Removing a commit that other team members have continued developing poses serious problems for collaboration. When they try to sync up with your repository, it will look like a chunk of the project history abruptly disappeared. The sequence below demonstrates what happens when you try to reset a public commit. The origin/master branch is the central repository’s version of your local master branch.



As soon as you add new commits after the reset, Git will think that your local history has diverged from origin/master, and the merge commit required to synchronize your repositories is likely to confuse and frustrate your team.  
  
The point is, make sure that you’re using git reset <commit> on a local experiment that went wrong—not on published changes. If you need to fix a public commit, the git revert command was designed specifically for this purpose.

## Examples

git reset <file>

Remove the specified file from the staging area, but leave the working directory unchanged. This unstages a file without overwriting any changes.

git reset

Reset the staging area to match the most recent commit, but leave the working directory unchanged. This unstages all files without overwriting any changes, giving you the opportunity to re-build the staged snapshot from scratch.

git reset --hard

Reset the staging area and the working directory to match the most recent commit. In addition to unstaging changes, the --hard flag tells Git to overwrite all changes in the working directory, too. Put another way: this obliterates all uncommitted changes, so make sure you really want to throw away your local developments before using it.

git reset <commit>

Move the current branch tip backward to commit, reset the staging area to match, but leave the working directory alone. All changes made since <commit> will reside in the working directory, which lets you re-commit the project history using cleaner, more atomic snapshots.

git reset --hard <commit>

Move the current branch tip backward to <commit>  and reset both the staging area and the working directory to match. This obliterates not only the uncommitted changes, but all commits after, as well.

## Unstaging a file

The git reset command is frequently encountered while preparing the staged snapshot. The next example assumes you have two files called hello.py and main.py that you’ve already added to the repository.

# Edit both hello.py and main.py

# Stage everything in the current directory

git add .

# Realize that the changes in hello.py and main.py

# should be committed in different snapshots

# Unstage main.py

git reset main.py

# Commit only hello.py

git commit -m "Make some changes to hello.py"

# Commit main.py in a separate snapshot

git add main.py

git commit -m "Edit main.py"

As you can see, git reset helps you keep your commits highly-focused by letting you unstage changes that aren’t related to the next commit.

## Removing Local Commits

The next example shows a more advanced use case. It demonstrates what happens when you’ve been working on a new experiment for a while, but decide to completely throw it away after committing a few snapshots.

# Create a new file called `foo.py` and add some code to it

# Commit it to the project history

git add foo.py

git commit -m "Start developing a crazy feature"

# Edit `foo.py` again and change some other tracked files, too

# Commit another snapshot

git commit -a -m "Continue my crazy feature"

# Decide to scrap the feature and remove the associated commits

git reset --hard HEAD~2

The git reset HEAD~2 command moves the current branch backward by two commits, effectively removing the two snapshots we just created from the project history. Remember that this kind of reset should only be used on unpublished commits. Never perform the above operation if you’ve already pushed your commits to a shared repository.

## Summary

To review, git reset is a powerful command that is used to undo local changes to the state of a Git repo. Git reset operates on "The Three Trees of Git". These trees are the Commit History (HEAD), the Staging Index, and the Working Directory. There are three command line options that correspond to the three trees. The options --soft, --mixed, and --hard can be passed to git reset.  
  
In this article we leveraged several other Git commands to help demonstrate the reset processes. Learn more about those commands on their individual pages at: [git status](https://www.atlassian.com/git/tutorials/inspecting-a-repository), [git log](https://www.atlassian.com/git/tutorials/git-log), [git add](https://www.atlassian.com/git/tutorials/saving-changes), [git checkout](https://www.atlassian.com/git/tutorials/using-branches/git-checkout), [git reflog](https://www.atlassian.com/git/tutorials/rewriting-history/git-reflog), and [git revert](https://www.atlassian.com/git/tutorials/undoing-changes/git-revert).

# Git RM

[git checkout](https://www.atlassian.com/git/tutorials/undoing-changes) [git clean](https://www.atlassian.com/git/tutorials/undoing-changes/git-clean) [git revert](https://www.atlassian.com/git/tutorials/undoing-changes/git-revert) [git reset](https://www.atlassian.com/git/tutorials/undoing-changes/git-reset) [git rm](https://www.atlassian.com/git/tutorials/undoing-changes/git-rm)

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)

# Rewriting history

###### Git commit --amend and other methods of rewriting history

## Intro

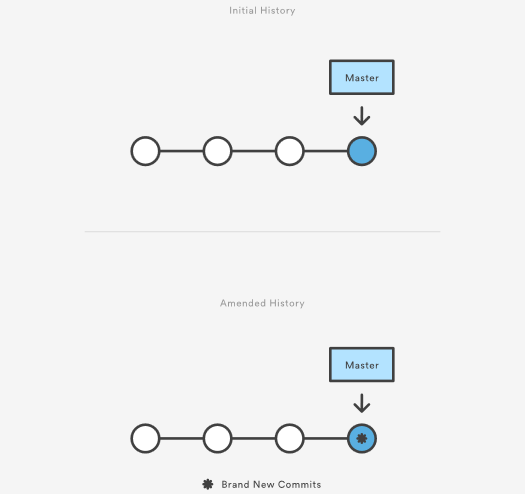
This tutorial will cover various methods of rewriting and altering Git history. Git uses a few different methods to record changes. We will discuss the strengths and weaknesses of the different methods and give examples of how to work with them. This tutorial discusses some of the most common reasons for overwriting committed snapshots and shows you how to avoid the pitfalls of doing so.

Git's main job is to make sure you never lose a committed change. But it's also designed to give you total control over your development workflow. This includes letting you define exactly what your project history looks like; however, it also creates the potential of losing commits. Git provides its history-rewriting commands under the disclaimer that using them may result in lost content.

Git has several mechanisms for storing history and saving changes. These mechanisms include: Commit --amend, git rebase and git reflog. These options give you powerful work flow customization options. By the end of this tutorial, you'll be familiar with commands that will let you restructure your Git commits, and be able to avoid pitfalls that are commonly encountered when rewriting history.

## Changing the Last Commit: git commit --amend

The git commit --amend command is a convenient way to modify the most recent commit. It lets you combine staged changes with the previous commit instead of creating an entirely new commit. It can also be used to simply edit the previous commit message without changing its snapshot. But, amending does not just alter the most recent commit, it replaces it entirely, meaning the amended commit will be a new entity with its own ref. To Git, it will look like a brand new commit, which is visualized with an asterisk (\*) in the diagram below. There are a few common scenarios for using git commit --amend. We'll cover usage examples in the following sections.



### Change most recent Git commit message

git commit --amend

Let's say you just committed and you made a mistake in your commit log message. Running this command when there is nothing staged lets you edit the previous commit’s message without altering its snapshot.

Premature commits happen all the time in the course of your everyday development. It’s easy to forget to stage a file or to format your commit message the wrong way. The --amend flag is a convenient way to fix these minor mistakes.

git commit --amend -m "an updated commit message"

Adding the -m option allows you to pass in a new message from the command line without being prompted to open an editor.

### Changing committed files

The following example demonstrates a common scenario in Git-based development. Let's say we've edited a few files that we would like to commit in a single snapshot, but then we forget to add one of the files the first time around. Fixing the error is simply a matter of staging the other file and committing with the --amend flag:

# Edit hello.py and main.py git add hello.py git commit

# Realize you forgot to add the changes from main.py git add main.py

git commit --amend --no-edit

The --no-edit flag will allow you to make the amendment to your commit without changing its commit message. The resulting commit will replace the incomplete one, and it will look like we committed the changes to hello.py and main.py in a single snapshot.

### Don’t amend public commits

Amended commits are actually entirely new commits and the previous commit will no longer be on your current branch. This has the same consequences as resetting a public snapshot. Avoid amending a commit that other developers have based their work on. This is a confusing situation for developers to be in and it’s complicated to recover from.

### Recap

To review, git commit --amend lets you take the most recent commit and add new staged changes to it. You can add or remove changes from the Git staging area to apply with a --amend commit. If there are no changes staged, a --amend will still prompt you to modify the last commit message log. Be cautious when using --amend on commits shared with other team members. Amending a commit that is shared with another user will potentially require confusing and lengthy merge conflict resolutions.

## Changing older or multiple commits

To modify older or multiple commits, you can use git rebase to combine a sequence of commits into a new base commit. In standard mode, git rebase allows you to literally rewrite history — automatically applying commits in your current working branch to the passed branch head. Since your new commits will be replacing the old, it's important to not use git rebase on commits that have been pushed public, or it will appear that your project history disappeared.

In these or similar instances where it's important to preserve a clean project history, adding the -i option to git rebase allows you to run rebase interactive. This gives you the opportunity to alter individual commits in the process, rather than moving all commits. You can learn more about interactive rebasing and additional rebase commands on the [git rebase page](https://www.atlassian.com/git/tutorials/rewriting-history/git-rebase).

#### Changing committed files

During a rebase, the edit or e command will pause the rebase playback on that commit and allow you to make additional changes with git commit --amend Git will interrupt the playback and present a message:

Stopped at 5d025d1... formatting

You can amend the commit now, with

git commit --amend

Once you are satisfied with your changes, run

git rebase --continue

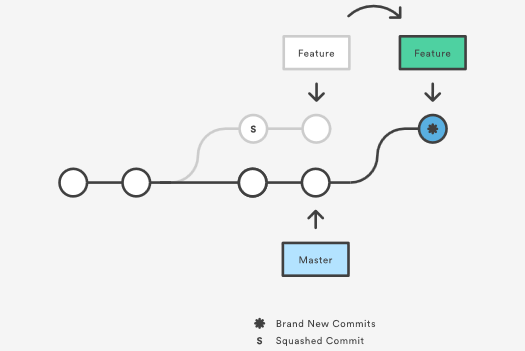
#### Multiple messages

Each regular Git commit will have a log message explaining what happened in the commit. These messages provide valuable insight into the project history. During a rebase, you can run a few commands on commits to modify commit messages.

* Reword or 'r' will stop rebase playback and let you rewrite the individual commit message during.
* Squash or 's' during rebase playback, any commits marked s will be paused on and you will be prompted to edit the separate commit messages into a combined message. More on this in the squash commits section below.
* Fixup or 'f' has the same combining effect as squash. Unlike squash, fixup commits will not interrupt rebase playback to open an editor to combine commit messages. The commits marked 'f' will have their messages discarded in-favor of the previous commit's message.

#### Squash commits for a clean history

The s "squash" command is where we see the true utility of rebase. Squash allows you to specify which commits you want to merge into the previous commits. This is what enables a "clean history." During rebase playback, Git will execute the specified rebase command for each commit. In the case of squash commits, Git will open your configured text editor and prompt to combine the specified commit messages. This entire process can be visualized as follows:



Note that the commits modified with a rebase command have a different ID than either of the original commits. Commits marked with pick will have a new ID if the previous commits have been rewritten.

Modern Git hosting solutions like Bitbucket now offer "auto squashing" features upon merge. These features will automatically rebase and squash a branch's commits for you when utilizing the hosted solutions UI. For more info see "[Squash commits when merging a Git branch with Bitbucket](https://blog.bitbucket.org/2017/01/31/git-squash-commits-merging-bitbucket/)."

### Recap

Git rebase gives you the power to modify your history, and interactive rebasing allows you to do so without leaving a “messy” trail. This creates the freedom to make and correct errors and refine your work, while still maintaining a clean, linear project history.

## The safety net: git reflog

Reference logs, or "reflogs" are a mechanism Git uses to record updates applied to tips of branches and other commit references. Reflog allows you to go back to commits even though they are not referenced by any branch or tag. After rewriting history, the reflog contains information about the old state of branches and allows you to go back to that state if necessary. Every time your branch tip is updated for any reason (by switching branches, pulling in new changes, rewriting history or simply by adding new commits), a new entry will be added to the reflog. In this section we will take a high level look at the git reflog command and explore some common uses.

### Usage

git reflog

This displays the reflog for the local repository.

git reflog --relative-date

This shows the reflog with relative date information (e.g. 2 weeks ago).

### Example

To understand git reflog, let's run through an example.

0a2e358 HEAD@{0}: reset: moving to HEAD~2

0254ea7 HEAD@{1}: checkout: moving from 2.2 to master

c10f740 HEAD@{2}: checkout: moving from master to 2.2

The reflog above shows a checkout from master to the 2.2 branch and back. From there, there's a hard reset to an older commit. The latest activity is represented at the top labeled HEAD@{0}.

If it turns out that you accidentally moved back, the reflog will contain the commit master pointed to (0254ea7) before you accidentally dropped 2 commits.

git reset --hard 0254ea7

Using Git reset, it is now possible to change master back to the commit it was before. This provides a safety net in case the history was accidentally changed.

It's important to note that the reflog only provides a safety net if changes have been committed to your local repository and that it only tracks movements of the repositories branch tip. Additionally reflog entries have an expiration date. The default expiration time for reflog entries is 90 days.

For additional information, see our [git reflog](https://www.atlassian.com/git/tutorials/rewriting-history/git-reflog) page.

## Summary

In this article we discussed several methods of changing git history, and undoing git changes. We took a high level look at the git rebase process. Some Key takeaways are:

* There are many ways to rewrite history with git.
* Use git commit --amend to change your latest log message.
* Use git commit --amend to make modifications to the most recent commit.
* Use git rebase to combine commits and modify history of a branch.
* git rebase -i gives much more fine grained control over history modifications than a standard git rebase.

Learn more about the commands we covered at their individual pages:

* [git rebase](https://www.atlassian.com/git/tutorials/rewriting-history/git-rebase)
* [git reflog](https://www.atlassian.com/git/tutorials/rewriting-history/git-reflog)

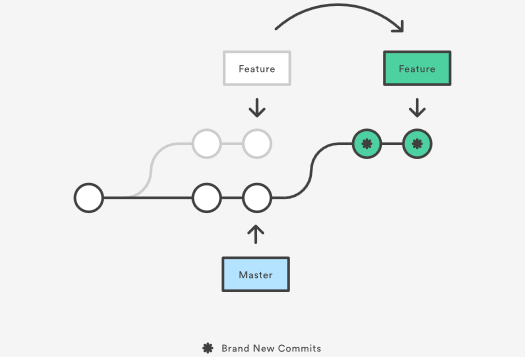
# git rebase

This document will serve as an in-depth discussion of the git rebase command. The Rebase command has also been looked at on the [setting up a repository](https://www.atlassian.com/git/tutorials/setting-up-a-repository) and [rewriting history](https://www.atlassian.com/git/tutorials/rewriting-history) pages. This page will take a more detailed look at git rebase configuration and execution. Common Rebase use cases and pitfalls will be covered here.

Rebase is one of two Git utilities that specializes in integrating changes from one branch onto another. The other change integration utility is git merge. Merge is always a forward moving change record. Alternatively, rebase has powerful history rewriting features. For a detailed look at Merge vs. Rebase, visit our [Merging vs Rebasing guide](https://www.atlassian.com/git/tutorials/merging-vs-rebasing). Rebase itself has 2 main modes: "manual" and "interactive" mode. We will cover the different Rebase modes in more detail below.

## What is git rebase?

Rebasing is the process of moving or combining a sequence of commits to a new base commit. Rebasing is most useful and easily visualized in the context of a feature branching workflow. The general process can be visualized as the following:



From a content perspective, rebasing is changing the base of your branch from one commit to another making it appear as if you'd created your branch from a different commit. Internally, Git accomplishes this by creating new commits and applying them to the specified base. It's very important to understand that even though the branch looks the same, it's composed of entirely new commits.

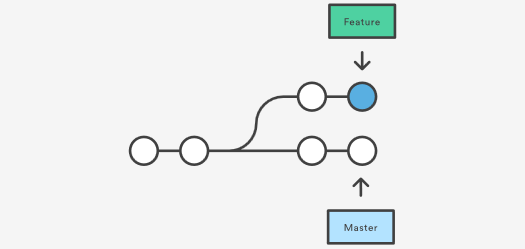
## Usage

The primary reason for rebasing is to maintain a linear project history. For example, consider a situation where the master branch has progressed since you started working on a feature branch. You want to get the latest updates to the master branch in your feature branch, but you want to keep your branch's history clean so it appears as if you've been working off the latest master branch. This gives the later benefit of a clean merge of your feature branch back into the master branch. Why do we want to maintain a "clean history"? The benefits of having a clean history become tangible when performing Git operations to investigate the introduction of a regression. A more real-world scenario would be:

1. A bug is identified in the master branch. A feature that was working successfully is now broken.
2. A developer examines the history of the master branch using git log because of the "clean history" the developer is quickly able to reason about the history of the project.
3. The developer can not identify when the bug was introduced using git log so the developer executes a git bisect.
4. Because the git history is clean, git bisect has a refined set of commits to compare when looking for the regression. The developer quickly finds the commit that introduced the bug and is able to act accordingly.

Learn more about [git log](https://www.atlassian.com/git/tutorials/git-log) and [git bisect](https://git-scm.com/docs/git-bisect) on their individual usage pages.

You have two options for integrating your feature into the master branch: merging directly or rebasing and then merging. The former option results in a 3-way merge and a merge commit, while the latter results in a fast-forward merge and a perfectly linear history. The following diagram demonstrates how rebasing onto the master branch facilitates a fast-forward merge.



Rebasing is a common way to integrate upstream changes into your local repository. Pulling in upstream changes with Git merge results in a superfluous merge commit every time you want to see how the project has progressed. On the other hand, rebasing is like saying, “I want to base my changes on what everybody has already done.”

### Don't rebase public history

As we've discussed previously in [rewriting history](https://www.atlassian.com/git/tutorials/rewriting-history), you should never rebase commits once they've been pushed to a public repository. The rebase would replace the old commits with new ones and it would look like that part of your project history abruptly vanished.

### Git Rebase Standard vs Git Rebase Interactive

Git rebase interactive is when git rebase accepts an -- i argument. This stands for "Interactive." Without any arguments, the command runs in standard mode. In both cases, let's assume we have created a separate feature branch.

# Create a feature branch based off of master

git checkout -b feature\_branch master

# Edit files

git commit -a -m "Adds new feature"

Git rebase in standard mode will automatically take the commits in your current working branch and apply them to the head of the passed branch.

git rebase

This automatically rebases the current branch onto <base>, which can be any kind of commit reference (for example an ID, a branch name, a tag, or a relative reference to HEAD).

Running git rebase with the -i flag begins an interactive rebasing session. Instead of blindly moving all of the commits to the new base, interactive rebasing gives you the opportunity to alter individual commits in the process. This lets you clean up history by removing, splitting, and altering an existing series of commits. It's like Git commit --amend on steroids.

git rebase --interactive

This rebases the current branch onto <base> but uses an interactive rebasing session. This opens an editor where you can enter commands (described below) for each commit to be rebased. These commands determine how individual commits will be transferred to the new base. You can also reorder the commit listing to change the order of the commits themselves. Once you've specified commands for each commit in the rebase, Git will begin playing back commits applying the rebase commands. The rebasing edit commands are as follows:

pick 2231360 some old commit

pick ee2adc2 Adds new feature

# Rebase 2cf755d..ee2adc2 onto 2cf755d (9 commands)

#

# Commands:

# p, pick = use commit

# r, reword = use commit, but edit the commit message

# e, edit = use commit, but stop for amending

# s, squash = use commit, but meld into previous commit

# f, fixup = like "squash", but discard this commit's log message

# x, exec = run command (the rest of the line) using shell

# d, drop = remove commit

#### Additional rebase commands

As detailed in the [rewriting history page](https://www.atlassian.com/git/tutorials/rewriting-history), rebasing can be used to change older and multiple commits, committed files, and multiple messages. While these are the most common applications, git rebase also has additional command options that can be useful in more complex applications.

* git rebase -- d means during playback the commit will be discarded from the final combined commit block.
* git rebase -- p leaves the commit as is. It will not modify the commit's message or content and will still be an individual commit in the branches history.
* git rebase -- x during playback executes a command line shell script on each marked commit. A useful example would be to run your codebase's test suite on specific commits, which may help identify regressions during a rebase.

### Recap

Interactive rebasing gives you complete control over what your project history looks like. This affords a lot of freedom to developers, as it lets them commit a "messy" history while they're focused on writing code, then go back and clean it up after the fact.

Most developers like to use an interactive rebase to polish a feature branch before merging it into the main code base. This gives them the opportunity to squash insignificant commits, delete obsolete ones, and make sure everything else is in order before committing to the “official” project history. To everybody else, it will look like the entire feature was developed in a single series of well-planned commits.

The real power of interactive rebasing can be seen in the history of the resulting master branch. To everybody else, it looks like you're a brilliant developer who implemented the new feature with the perfect amount of commits the first time around. This is how interactive rebasing can keep a project's history clean and meaningful.

### Configuration options

There are a few rebase properties that can be set using git config. These options will alter the git rebase output look and feel.

* **rebase.stat**: A boolean that is set to false by default. The option toggles display of visual diffstat content that shows what changed since the last debase.
* **rebase.autoSquash:** A boolean value that toggles the --autosquash behavior.
* **rebase.missingCommitsCheck:** Can be set to multiple values which change rebase behavior around missing commits.

|  |  |
| --- | --- |
| warn | Prints warning output in interactive mode which warns of removed commits |
| error | Stops the rebase and prints removed commit warning messages |
| ignore | Set by default this ignores any missing commit warnings |

* **rebase.instructionFormat:** A git log format string that will be used for formatting interactive rebase display

### Advanced rebase application

The command line argument --onto can be passed to git rebase. When in git rebase --onto mode the command expands to:

git rebase --onto

The --onto command enables a more powerful form or rebase that allows passing specific refs to be the tips of a rebase.  
Let’s say we have an example repo with branches like:

o---o---o---o---o master

\

o---o---o---o---o featureA

\

o---o---o featureB

featureB is based on featureA, however, we realize featureB is not dependent on any of the changes in featureA and could just be branched off master.

git rebase --onto master featureA featureB

featureA is the <oldbase>. master becomes the <newbase> and featureB is reference for what HEAD of the <newbase> will point to. The results are then:

o---o---o featureB

/

o---o---o---o---o master

\

o---o---o---o---o featureA

## Understanding the dangers of rebase

One caveat to consider when working with Git Rebase is merge conflicts may become more frequent during a rebase workflow. This occurs if you have a long-lived branch that has strayed from master. Eventually you will want to rebase against master and at that time it may contain many new commits that your branch changes may conflict with. This is easily remedied by rebasing your branch frequently against master, and making more frequent commits. The --continue and --abort command line arguments can be passed to git rebase to advance or reset the the rebase when dealing with conflicts.

A more serious rebase caveat is lost commits from interactive history rewriting. Running rebase in interactive mode and executing subcommands like squash or drop will remove commits from your branche's immediate log. At first glance this can appear as though the commits are permanently gone. Using git reflog these commits can be restored and the entire rebase can be undone. For more info on using git reflog to find lost commits, visit our [Git reflog documentation page](https://www.atlassian.com/git/tutorials/rewriting-history/git-reflog).

Git Rebase itself is not seriously dangerous. The real danger cases arise when executing history rewriting interactive rebases and force pushing the results to a remote branch that's shared by other users. This is a pattern that should be avoided as it has the capability to overwrite other remote users' work when they pull.

## Recovering from upstream rebase

If another user has rebased and force pushed to the branch that you’re committing to, a git pull will then overwrite any commits you have based off that previous branch with the tip that was force pushed. Luckily, using git reflog you can get the reflog of the remote branch. On the remote branch's reflog you can find a ref before it was rebased. You can then rebase your branch against that remote ref using the --onto option as discussed above in the Advanced Rebase Application section.

## Summary

In this article we covered git rebase usage. We discussed basic and advanced use cases and more advanced examples. Some key discussion points are:

* git rebase standard vs interactive modes
* git rebase configuration options
* git rebase --onto
* git rebase lost commits

We looked at git rebase usage with other tools like [git reflog](https://www.atlassian.com/git/tutorials/rewriting-history/git-reflog), [git fetch](https://www.atlassian.com/git/tutorials/syncing#git-fetch), and [git push](https://www.atlassian.com/git/tutorials/syncing#git-push). Visit their corresponding pages for further information.

# git reflog

This page provides a detailed discussion of the git reflog command. Git keeps track of updates to the tip of branches using a mechanism called reference logs, or "reflogs." Many Git commands accept a parameter for specifying a reference or "ref", which is a pointer to a commit. Common examples include:

* git checkout
* git reset
* git merge

Reflogs track when Git refs were updated in the local repository. In addition to branch tip reflogs, a special reflog is maintained for the Git stash. Reflogs are stored in directories under the local repository's .git directory. git reflog directories can be found at .git/logs/refs/heads/., .git/logs/HEAD, and also .git/logs/refs/stash if the git stash has been used on the repo.

We discussed git reflog at a high level on the [Rewriting History Page](https://www.atlassian.com/git/tutorials/rewriting-history). This document will cover: extended configuration options of git reflog, common use-cases and pitfalls of git reflog, how to undo changes with git reflog, and more.

## Basic usage

The most basic Reflog use case is invoking:

git reflog

This is essentially a short cut that's equivalent to:

git reflog show HEAD

This will output the HEAD reflog. You should see output similar to:

eff544f HEAD@{0}: commit: migrate existing content

bf871fd HEAD@{1}: commit: Add Git Reflog outline

9a4491f HEAD@{2}: checkout: moving from master to git\_reflog

9a4491f HEAD@{3}: checkout: moving from Git\_Config to master

39b159a HEAD@{4}: commit: expand on git context

9b3aa71 HEAD@{5}: commit: more color clarification

f34388b HEAD@{6}: commit: expand on color support

9962aed HEAD@{7}: commit: a git editor -> the Git editor

Visit the [Rewriting History page](https://www.atlassian.com/git/tutorials/rewriting-history) for another example of common reflog access.

### Reflog references

By default, git reflog will output the reflog of the HEAD ref. HEAD is a symbolic reference to the currently active branch. Reflogs are available for other refs as well. The syntax to access a git ref is name@{qualifier}. In addition to HEAD refs, other branches, tags, remotes, and the Git stash can be referenced as well.

You can get a complete reflog of all refs by executing:

git reflog show --all

To see the reflog for a specific branch pass that branch name to git reflog show

git reflog show otherbranch

9a4491f otherbranch@{0}: commit: seperate articles into branch PRs

35aee4a otherbranch{1}: commit (initial): initial commit add git-init and setting-up-a-repo docs

Executing this example will show a reflog for the otherbranch branch. The following example assumes you have previously stashed some changes using the git stash command.

git reflog stash

0d44de3 stash@{0}: WIP on git\_reflog: c492574 flesh out intro

This will output a reflog for the Git stash. The returned ref pointers can be passed to other Git commands:

git diff stash@{0} otherbranch@{0}

When executed, this example code will display Git diff output comparing the stash@{0} changes against the otherbranch@{0} ref.

### Timed reflogs

Every reflog entry has a timestamp attached to it. These timestamps can be leveraged as the qualifier token of Git ref pointer syntax. This enables filtering Git reflogs by time. The following are some examples of available time qualifiers:

* 1.minute.ago
* 1.hour.ago
* 1.day.ago
* yesterday
* 1.week.ago
* 1.month.ago
* 1.year.ago
* 2011-05-17.09:00:00

Time qualifiers can be combined (e.g. 1.day.2.hours.ago), Additionally plural forms are accepted (e.g. 5.minutes.ago).

Time qualifier refs can be passed to other git commands.

git diff master@{0} master@{1.day.ago}

This example will diff the current master branch against master 1 day ago. This example is very useful if you want to know changes that have occurred within a time frame.

## Subcommands & configuration options

git reflog accepts few addition arguments which are considered subcommands.

### Show - git reflog show

show is implicitly passed by default. For example, the command:

git reflog master@{0}

is equivalent to the command:

git reflog show master@{0}

In addition, git reflog show is an alias for git log -g --abbrev-commit --pretty=oneline. Executing git reflog show will display the log for the passed <refid>.

### Expire - git reflog expire

The expire subcommand cleans up old or unreachable reflog entries. The expire subcommand has potential for data loss. This subcommand is not typically used by end users, but used by git internally. Passing a -n or --dry-run option to git reflog expire Will perform a "dry run" which will output which reflog entries are marked to be pruned, but will not actually prune them.

By default, the reflog expiration date is set to 90 days. An expire time can be specified by passing a command line argument --expire=time to git reflog expire or by setting a git configuration name of gc.reflogExpire.

### Delete - git reflog delete

The delete subcommand is self explanatory and will delete a passed in reflog entry. As with expire, delete has potential to lose data and is not commonly invoked by end users.

## Recovering lost commits

Git never really loses anything, even when performing history rewriting operations like rebasing or commit amending. For the next example let's assume that we have made some new changes to our repo. Our git log --pretty=oneline looks like the following:

338fbcb41de10f7f2e54095f5649426cb4bf2458 extended content

1e63ceab309da94256db8fb1f35b1678fb74abd4 bunch of content

c49257493a95185997c87e0bc3a9481715270086 flesh out intro

eff544f986d270d7f97c77618314a06f024c7916 migrate existing content

bf871fd762d8ef2e146d7f0226e81a92f91975ad Add Git Reflog outline

35aee4a4404c42128bee8468a9517418ed0eb3dc initial commit add git-init and setting-up-a-repo docs

We then commit those changes and execute the following:

#make changes to HEAD

git commit -am "some WIP changes"

With the addition of the new commit. The log now looks like:

37656e19d4e4f1a9b419f57850c8f1974f871b07 some WIP changes

338fbcb41de10f7f2e54095f5649426cb4bf2458 extended content

1e63ceab309da94256db8fb1f35b1678fb74abd4 bunch of content

c49257493a95185997c87e0bc3a9481715270086 flesh out intro

eff544f986d270d7f97c77618314a06f024c7916 migrate existing content

bf871fd762d8ef2e146d7f0226e81a92f91975ad Add Git Reflog outline

35aee4a4404c42128bee8468a9517418ed0eb3dc initial commit add git-init and setting-up-a-repo docs

At this point we perform an interactive rebase against the master branch by executing...

git rebase -i origin/master

During the rebase we mark commits for squash with the s rebase subcommand. During the rebase, we squash a few commits into the most recent "some WIP changes" commit.

Because we squashed commits the git log output now looks like:

40dhsoi37656e19d4e4f1a9b419f57850ch87dah987698hs some WIP changes

35aee4a4404c42128bee8468a9517418ed0eb3dc initial commit add git-init and setting-up-a-repo docs

If we examine git log at this point it appears that we no longer have the commits that were marked for squashing. What if we want to operate on one of the squashed commits? Maybe to remove its changes from history? This is an opportunity to leverage the reflog.

git reflog

37656e1 HEAD@{0}: rebase -i (finish): returning to refs/heads/git\_reflog

37656e1 HEAD@{1}: rebase -i (start): checkout origin/master

37656e1 HEAD@{2}: commit: some WIP changes

We can see there are reflog entries for the start and finish of the rebase and prior to those is our "some WIP changes" commit. We can pass the reflog ref to git reset and reset to a commit that was before the rebase.

git reset HEAD@{2}

Executing this reset command will move HEAD to the commit where "some WIP changes" was added, essentially restoring the other squashed commits.

## Summary

In this tutorial we discussed the git reflog command. Some key points covered were:

* How to view reflog for specific branches
* How to undo a git rebase using the reflog
* How specify and view time based reflog entries

We briefly mentioned that git reflog can be used with other git commands like [git checkout](https://www.atlassian.com/git/tutorials/using-branches#git-checkout), [git reset](https://www.atlassian.com/git/tutorials/resetting-checking-out-and-reverting), and [git merge](https://www.atlassian.com/git/tutorials/git-merge). Learn more at their respective pages. For additional discussion on refs and the reflog, [learn more here](https://www.atlassian.com/git/tutorials/refs-and-the-reflog).

# git syncing

[git remote](https://www.atlassian.com/git/tutorials/syncing) [git fetch](https://www.atlassian.com/git/tutorials/syncing/git-fetch) [git push](https://www.atlassian.com/git/tutorials/syncing/git-push) [git pull](https://www.atlassian.com/git/tutorials/syncing/git-pull)

SVN uses a single centralized repository to serve as the communication hub for developers, and collaboration takes place by passing changesets between the developers’ working copies and the central repository. This is different from Git's distributed collaboration model, which gives every developer their own copy of the repository, complete with its own local history and branch structure. Users typically need to share a series of commits rather than a single changeset. Instead of committing a changeset from a working copy to the central repository, Git lets you share entire branches between repositories.

The git remote command is one piece of the broader system which is responsible for syncing changes. Records registered through the git remote command are used in conjunction with the [git fetch](https://www.atlassian.com/git/tutorials/syncing/git-fetch), [git push](https://www.atlassian.com/git/tutorials/syncing/git-push), and [git pull](https://www.atlassian.com/git/tutorials/syncing/git-pull) commands. These commands all have their own syncing responsibilities which can be explored on the corresponding links.

## Git remote

The git remote command lets you create, view, and delete connections to other repositories. Remote connections are more like bookmarks rather than direct links into other repositories. Instead of providing real-time access to another repository, they serve as convenient names that can be used to reference a not-so-convenient URL.

For example, the following diagram shows two remote connections from your repo into the central repo and another developer’s repo. Instead of referencing them by their full URLs, you can pass the origin and john shortcuts to other Git commands.

## Git remote usage overview

The git remote command is essentially an interface for managing a list of remote entries that are stored in the repository's ./.git/config file. The following commands are used to view the current state of the remote list.

## Viewing git remote configurations

git remote

List the remote connections you have to other repositories.

git remote -v

Same as the above command, but include the URL of each connection.

## Creating and modifying git remote configurations

The git remote command is also a convenience or 'helper' method for modifying a repo's ./.git/config file. The commands presented below let you manage connections with other repositories. The following commands will modify the repo's /.git/config file. The result of the following commands can also be achieved by directly editing the ./.git/config file with a text editor.

git remote add <name> <url>

Create a new connection to a remote repository. After adding a remote, you’ll be able to use <name> as a convenient shortcut for <url> in other Git commands.

git remote rm <name>

Remove the connection to the remote repository called <name>.

git remote rename <old-name> <new-name>

Rename a remote connection from <old-name> to <new-name>.

## Git remote discussion

Git is designed to give each developer an entirely isolated development environment. This means that information is not automatically passed back and forth between repositories. Instead, developers need to manually pull upstream commits into their local repository or manually push their local commits back up to the central repository. The git remote command is really just an easier way to pass URLs to these "sharing" commands.

## The origin Remote

When you clone a repository with git clone, it automatically creates a remote connection called origin pointing back to the cloned repository. This is useful for developers creating a local copy of a central repository, since it provides an easy way to pull upstream changes or publish local commits. This behavior is also why most Git-based projects call their central repository origin.

## Repository URLs

Git supports many ways to reference a remote repository. Two of the easiest ways to access a remote repo are via the HTTP and the SSH protocols. HTTP is an easy way to allow anonymous, read-only access to a repository. For example:

http://host/path/to/repo.git

But, it’s generally not possible to push commits to an HTTP address (you wouldn’t want to allow anonymous pushes anyways). For read-write access, you should use SSH instead:

ssh://user@host/path/to/repo.git

You’ll need a valid SSH account on the host machine, but other than that, Git supports authenticated access via SSH out of the box. Modern secure 3rd party hosting solutions like Bitbucket.com will provide these URLs for you.

## Git remote commands

The git remote command is one of many Git commands that takes additional appended 'subcommands'. Below is an examination of the commonly used git remote subcommands.

ADD <NAME> <URL>

Adds a record to ./.git/config for remote named <name> at the repository url <url>.

Accepts a -f option, that will git fetch <name> immediately after the remote record is created.

Accepts a --tags option, that will git fetch <name> immediately and import every tag from the remote repository.

RENAME <OLD> <NEW>

Updates ./.git/config to rename the record <OLD> to <NEW>. All remote-tracking branches and configuration settings for the remote are updated.

REMOVE or RM <NAME>

Modifies ./.git/config and removes the remote named <NAME>. All remote-tracking branches and configuration settings for the remote are removed.

GET-URL <NAME>

Outputs the URLs for a remote record.

Accepts --push, push URLs are queried rather than fetch URLs.

With --all, all URLs for the remote will be listed.

SHOW <NAME>

Outputs high-level information about the remote <NAME>.

PRUNE <NAME>

Deletes any local branches for <NAME> that are not present on the remote repository.

Accepts a --dry-run option which will list what branches are set to be pruned, but will not actually prune them.

## Git remote examples

In addition to origin, it’s often convenient to have a connection to your teammates’ repositories. For example, if your co-worker, John, maintained a publicly accessible repository on dev.example.com/john.git, you could add a connection as follows:

git remote add john http://dev.example.com/john.git

Having this kind of access to individual developers’ repositories makes it possible to collaborate outside of the central repository. This can be very useful for small teams working on a large project.

## Showing your remotes

By default, the git remote command will list previously stored remote connections to other repositories. This will produce single line output that lists the names of "bookmark" name of remote repos.

$ git remote  
origin  
upstream  
other\_users\_repo

Invoking git remote with the -v option will print the list of bookmarked repository names and additionally, the corresponding repository URL. The -v option stands for "verbose". Below is example output of verbose git remote output.

git remote -v  
origin  git@bitbucket.com:origin\_user/reponame.git (fetch)  
origin  git@bitbucket.com:origin\_user/reponame.git (push)  
upstream    https://bitbucket.com/upstream\_user/reponame.git (fetch)  
upstream    https://bitbucket.com/upstream\_user/reponame.git (push)  
other\_users\_repo    https://bitbucket.com/other\_users\_repo/reponame (fetch)  
other\_users\_repo    https://bitbucket.com/other\_users\_repo/reponame (push)

## Adding Remote Repositories

The git remote add command will create a new connection record to a remote repository. After adding a remote, you’ll be able to use <name> as a convenient shortcut for <url> in other Git commands. For more information on the accepted URL syntax, view the "Repository URLs" section below. This command will create a new record within the repository's ./.git/config. An example of this config file update follows:

$ git remote add fake\_test https://bitbucket.com/upstream\_user/reponame.git; [remote "remote\_test"]   
   url = https://bitbucket.com/upstream\_user/reponame.git   
   fetch = +refs/heads/\*:refs/remotes/remote\_test/\*

## Inspecting a Remote

The show subcommand can be appended to git remote to give detailed output on the configuration of a remote. This output will contain a list of branches associated with the remote and also the endpoints attached for fetching and pushing.

git remote show upstream  
\* remote upstream  
   Fetch URL: https://bitbucket.com/upstream\_user/reponame.git  
   Push URL: https://bitbucket.com/upstream\_user/reponame.git  
   HEAD branch: master  
   Remote branches:  
      master tracked  
      simd-deprecated tracked  
      tutorial tracked  
   Local ref configured for 'git push':  
      master pushes to master (fast-forwardable)

## Fetching and pulling from Git remotes

Once a remote record has been configured through the use of the git remote command, the remote name can be passed as an argument to other Git commands to communicate with the remote repo. Both [git fetch](https://www.atlassian.com/git/tutorials/syncing/git-fetch), and [git pull](https://www.atlassian.com/git/tutorials/syncing/git-pull) can be used to read from a remote repository. Both commands have different operations that are explained in further depth on their respective links.

## Pushing to Git remotes

The git push command is used to write to a remote repository.

git push <remote-name> <branch-name>

This example will upload the local state of <branch-name> to the remote repository specified by <remote-name>.

## Renaming and Removing Remotes

git remote rename <old-name> <new-name>

The command git remote rename is self-explanatory. When executed, this command will rename a remote connection from <old-name> to <new-name>. Additionally, this will modify the contents of ./.git/config to rename the record for the remote there as well.

git remote rm <name>

The command git remote rm will remove the connection to the remote repository specified by the <name> parameter. To demonstrate let us 'undo' the remote addition from our last example. If we execute git remote rm remote\_test, and then examine the contents of ./.git/config we can see that the [remote "remote\_test"] record is no longer there.

# git fetch

[git remote](https://www.atlassian.com/git/tutorials/syncing) [git fetch](https://www.atlassian.com/git/tutorials/syncing/git-fetch) [git push](https://www.atlassian.com/git/tutorials/syncing/git-push) [git pull](https://www.atlassian.com/git/tutorials/syncing/git-pull)

The git fetch command downloads commits, files, and refs from a remote repository into your local repo. Fetching is what you do when you want to see what everybody else has been working on. It’s similar to svn update in that it lets you see how the central history has progressed, but it doesn’t force you to actually merge the changes into your repository. Git isolates fetched content as a from existing local content, it has absolutely no effect on your local development work. Fetched content has to be explicitly checked out using the [git checkout](https://www.atlassian.com/git/tutorials/using-branches/git-checkout) command. This makes fetching a safe way to review commits before integrating them with your local repository.

When downloading content from a remote repo, git pull and git fetch commands are available to accomplish the task. You can consider git fetch the 'safe' version of the two commands. It will download the remote content but not update your local repo's working state, leaving your current work intact. git pull is the more aggressive alternative, it will download the remote content for the active local branch and immediately execute git merge to create a merge commit for the new remote content. If you have pending changes in progress this will cause conflicts and kickoff the merge conflict resolution flow.

## How git fetch works with remote branches

To better understand how git fetch works let us discuss how Git organizes and stores commits. Behind the scenes, in the repository's ./.git/objects directory, Git stores all commits, local and remote. Git keeps remote and local branch commits distinctly separate through the use of branch refs. The refs for local branches are stored in the ./.git/refs/heads/. Executing the [git branch](https://www.atlassian.com/git/tutorials/using-branches) command will output a list of the local branch refs. The following is an example of git branch output with some demo branch names.

git branch  
master  
feature1  
debug2

Examining the contents of the /.git/refs/heads/ directory would reveal similar output.

ls ./.git/refs/heads/  
master  
feature1  
debug2

Remote branches are just like local branches, except they map to commits from somebody else’s repository. Remote branches are prefixed by the remote they belong to so that you don’t mix them up with local branches. Like local branches, Git also has refs for remote branches. Remote branch refs live in the ./.git/refs/remotes/ directory. The next example code snippet shows the branches you might see after fetching a remote repo named conveniently named remote-repo:

git branch -r  
# origin/master  
# origin/feature1  
# origin/debug2  
# remote-repo/master  
# remote-repo/other-feature

This output displays the local branches we had previously examined but now displays them prefixed with origin/. Additionally, we now see the remote branches prefixed with remote-repo. You can check out a remote branch just like a local one, but this puts you in a detached HEAD state (just like checking out an old commit). You can think of them as read-only branches. To view your remote branches, simply pass the -r flag to the git branch command.

You can inspect remote branches with the usual git checkout and git log commands. If you approve the changes a remote branch contains, you can merge it into a local branch with a normal git merge. So, unlike SVN, synchronizing your local repository with a remote repository is actually a two-step process: fetch, then merge. The git pull command is a convenient shortcut for this process.

## Git fetch commands and options

git fetch <remote>

Fetch all of the branches from the repository. This also downloads all of the required commits and files from the other repository.

git fetch <remote> <branch>

Same as the above command, but only fetch the specified branch.

git fetch --all

A power move which fetches all registered remotes and their branches:

git fetch --dry-run

The --dry-run option will perform a demo run of the command. I will output examples of actions it will take during the fetch but not apply them.

## Git fetch examples

## git fetch a remote branch

The following example will demonstrate how to fetch a remote branch and update your local working state to the remote contents. In this example, lets assume there is a central repo origin which the local repository has been cloned from using the git clone command. Let us also assume an additional remote repository named coworkers\_repo that contains a feature\_branch which we will configure and fetch. With these assumptions set let us continue the example.

Firstly we will need to configure the remote repo using the [git remote](https://www.atlassian.com/git/tutorials/syncing) command.

git remote coworkers\_repo git@bitbucket.org:coworker/coworkers\_repo.git

Here we have created a reference to the coworker's repo using the repo URL. We will now pass that remote name to git fetch to download the contents.

git fetch coworkers feature\_branch  
fetching coworkers/feature\_branch

We now locally have the contents of coworkers/feature\_branch we will need the integrate this into our local working copy. We begin this process by using the [git checkout](https://www.atlassian.com/git/tutorials/using-branches/git-checkout) command to checkout the newly downloaded remote branch.

git checkout coworkers/feature\_branch  
Note: checking out coworkers/feature\_branch'.  
  
You are in 'detached HEAD' state. You can look around, make experimental  
changes and commit them, and you can discard any commits you make in this  
state without impacting any branches by performing another checkout.  
  
If you want to create a new branch to retain commits you create, you may  
do so (now or later) by using -b with the checkout command again. Example:  
  
git checkout -b <new-branch-name>

The output from this checkout operation indicates that we are in a detached HEAD state. This is expected and means that our HEAD ref is pointing to a ref that is not in sequence with our local history. Being that HEAD is pointed at the coworkers/feature\_branch ref, we can create a new local branch from that ref. The 'detached HEAD' output shows us how to do this using the git checkout command:

git checkout -b local\_feature\_branch

Here we have created a new local branch named local\_feature\_branch this puts updates HEAD to point at the latest remote content and we can continue development on it from this point.

## Synchronize origin with git fetch

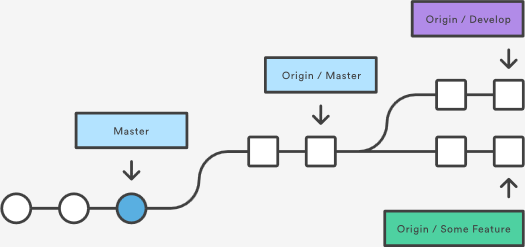
The following example walks through the typical workflow for synchronizing your local repository with the central repository's master branch.

git fetch origin

This will display the branches that were downloaded:

a1e8fb5..45e66a4 master -> origin/master  
a1e8fb5..9e8ab1c develop -> origin/develop  
\* [new branch] some-feature -> origin/some-feature

The commits from these new remote branches are shown as squares instead of circles in the diagram below. As you can see, git fetch gives you access to the entire branch structure of another repository.



To see what commits have been added to the upstream master, you can run a git log using origin/master as a filter:

git log --oneline master..origin/master

To approve the changes and merge them into your local master branch with the following commands:

git checkout master  
git log origin/master

Then we can use git merge origin/master:

git merge origin/master

The origin/master and master branches now point to the same commit, and you are synchronized with the upstream developments.

## Git fetch summary

In review, git fetch is a primary command used to download contents from a remote repository. git fetch is used in conjunction with git remote, git branch, git checkout, and [git reset](https://www.atlassian.com/git/tutorials/undoing-changes/git-reset) to update a local repository to the state of a remote. The git fetch command is a critical piece of collaborative git work flows. git fetch has similar behavior to git pull however git fetch can be considered a safer, nondestructive version.

# git push

[git remote](https://www.atlassian.com/git/tutorials/syncing) [git fetch](https://www.atlassian.com/git/tutorials/syncing/git-fetch) [git push](https://www.atlassian.com/git/tutorials/syncing/git-push) [git pull](https://www.atlassian.com/git/tutorials/syncing/git-pull)

The git push command is used to upload local repository content to a remote repository. Pushing is how you transfer commits from your local repository to a remote repo. It's the counterpart to [git fetch](https://www.atlassian.com/git/tutorials/syncing/git-fetch), but whereas fetching imports commits to local branches, pushing exports commits to remote branches. Remote branches are configured using the [git remote](https://www.atlassian.com/git/tutorials/syncing) command. Pushing has the potential to overwrite changes, caution should be taken when pushing. These issues are discussed below.

## Git push usage

git push <remote> <branch>

Push the specified branch to <remote>, along with all of the necessary commits and internal objects. This creates a local branch in the destination repository. To prevent you from overwriting commits, Git won’t let you push when it results in a non-fast-forward merge in the destination repository.

git push <remote> --force

Same as the above command, but force the push even if it results in a non-fast-forward merge. Do not use the --force flag unless you’re absolutely sure you know what you’re doing.

git push <remote> --all

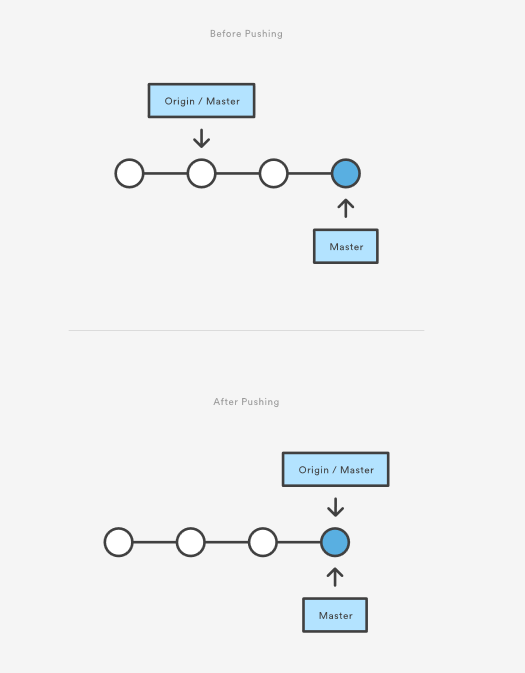
Push all of your local branches to the specified remote.

git push <remote> --tags

Tags are not automatically pushed when you push a branch or use the --all option. The --tags flag sends all of your local tags to the remote repository.

## Git push discussion

git push is most commonly used to publish an upload local changes to a central repository. After a local repository has been modified a push is executed to share the modifications with remote team members.



The above diagram shows what happens when your local master has progressed past the central repository’s master and you publish changes by running git push origin master. Notice how git push is essentially the same as running git merge master from inside the remote repository.

## Git push and syncing

git push is one component of many used in the overall Git "syncing" process. The syncing commands operate on remote branches which are configured using the [git remote](https://www.atlassian.com/git/tutorials/syncing) command. git push can be considered and 'upload' command whereas, [git fetch](https://www.atlassian.com/git/tutorials/syncing/git-fetch) and [git pull](https://www.atlassian.com/git/tutorials/syncing/git-pull) can be thought of as 'download' commands. Once changesets have been moved via a download or upload a [git merge](https://www.atlassian.com/git/tutorials/using-branches/git-merge) may be performed at the destination to integrate the changes.

## Pushing to bare repositories

A frequently used, modern Git practice is to have a remotely hosted --bare repository act as a central origin repository. This origin repository is often hosted off-site with a trusted 3rd party like Bitbucket. Since pushing messes with the remote branch structure, It is safest and most common to push to repositories that have been created with the --bare flag. Bare repos don’t have a working directory so a push will not alter any in progress working directory content. For more information on bare repository creation, read about [git init](https://www.atlassian.com/git/tutorials/setting-up-a-repository/git-init).

## Force Pushing

Git prevents you from overwriting the central repository’s history by refusing push requests when they result in a non-fast-forward merge. So, if the remote history has diverged from your history, you need to pull the remote branch and merge it into your local one, then try pushing again. This is similar to how SVN makes you synchronize with the central repository via svn update before committing a changeset.

The --force flag overrides this behavior and makes the remote repository’s branch match your local one, deleting any upstream changes that may have occurred since you last pulled. The only time you should ever need to force push is when you realize that the commits you just shared were not quite right and you fixed them with a git commit --amend or an interactive rebase. However, you must be absolutely certain that none of your teammates have pulled those commits before using the --force option.

## Examples

## Default git push

The following example describes one of the standard methods for publishing local contributions to the central repository. First, it makes sure your local master is up-to-date by fetching the central repository’s copy and rebasing your changes on top of them. The interactive rebase is also a good opportunity to clean up your commits before sharing them. Then, the git push command sends all of the commits on your local master to the central repository.

git checkout master  
git fetch origin master  
git rebase -i origin/master  
# Squash commits, fix up commit messages etc.  
git push origin master

Since we already made sure the local master was up-to-date, this should result in a fast-forward merge, and git push should not complain about any of the non-fast-forward issues discussed above.

## Amended force push

The [git commit](https://www.atlassian.com/git/tutorials/saving-changes/git-commit) command accepts a --amend option which will update the previous commit. A commit is often amended to update the commit message or add new changes. Once a commit is amended a git push will fail because Git will see the amended commit and the remote commit as diverged content. The --force option must be used to push an amended commit.

# make changes to a repo and git add  
git commit --amend  
# update the existing commit message  
git push --force origin master

The above example assumes it is being executed on an existing repository with a commit history. git commit --amend is used to update the previous commit. The amended commit is then force pushed using the --force option.

## Deleting a remote branch or tag

Sometimes branches need to be cleaned up for book keeping or organizational purposes. The fully delete a branch, it must be deleted locally and also remotely.

git branch -D branch\_name  
git push origin :branch\_name

The above will delete the remote branch named branch\_name passing a branch name prefixed with a colon to git push will delete the remote branch.

# git pull

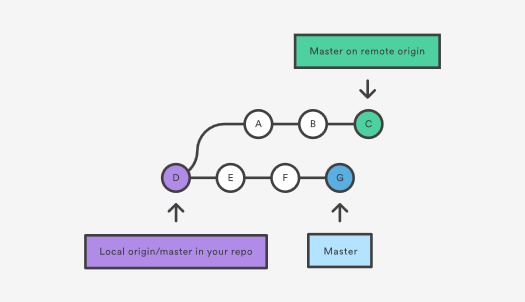
[git remote](https://www.atlassian.com/git/tutorials/syncing) [git fetch](https://www.atlassian.com/git/tutorials/syncing/git-fetch) [git push](https://www.atlassian.com/git/tutorials/syncing/git-push) [git pull](https://www.atlassian.com/git/tutorials/syncing/git-pull)

The git pull command is used to fetch and download content from a remote repository and immediately update the local repository to match that content. Merging remote upstream changes into your local repository is a common task in Git-based collaboration work flows. The git pull command is actually a combination of two other commands, [git fetch](https://www.atlassian.com/git/tutorials/syncing/git-fetch) followed by [git merge](https://www.atlassian.com/git/tutorials/using-branches/git-merge). In the first stage of operation git pull will execute a git fetch scoped to the local branch that HEAD is pointed at. Once the content is downloaded, git pull will enter a merge workflow. A new merge commit will be-created and HEAD updated to point at the new commit.

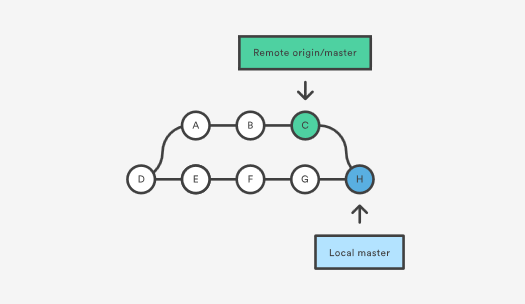
## Git pull usage

## How it works

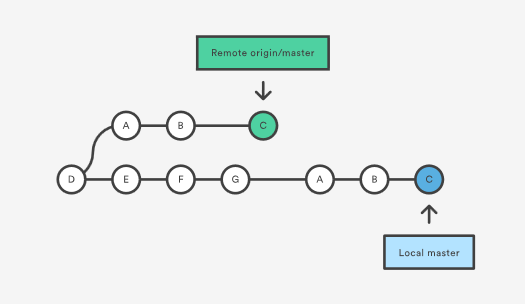
The git pull command first runs git fetch which downloads content from the specified remote repository. Then a git merge is executed to merge the remote content refs and heads into a new local merge commit. To better demonstrate the pull and merging process let us consider the following example. Assume we have a repository with a master branch and a remote origin.



In this scenario, git pull will download all the changes from the point where the local and master diverged. In this example, that point is E. git pull will fetch the diverged remote commits which are A-B-C. The pull process will then create a new local merge commit containing the content of the new diverged remote commits.



In the above diagram, we can see the new commit H. This commit is a new merge commit that contains the contents of remote A-B-C commits and has a combined log message. This example is one of a few git pull merging strategies. A --rebase option can be passed to git pull to use a rebase merging strategy instead of a merge commit. The next example will demonstrate how a rebase pull works. Assume that we are at a starting point of our first diagram, and we have executed git pull --rebase.



n this diagram, we can now see that a rebase pull does not create the new H commit. Instead, the rebase has copied the remote commits A--B--C and appended them to the local origin/master commit history.

## Common Options

git pull <remote>

Fetch the specified remote’s copy of the current branch and immediately merge it into the local copy. This is the same as git fetch <remote> followed by git merge origin/<current-branch>.

git pull --no-commit <remote>

Similar to the default invocation, fetches the remote content but does not create a new merge commit.

git pull --rebase <remote>

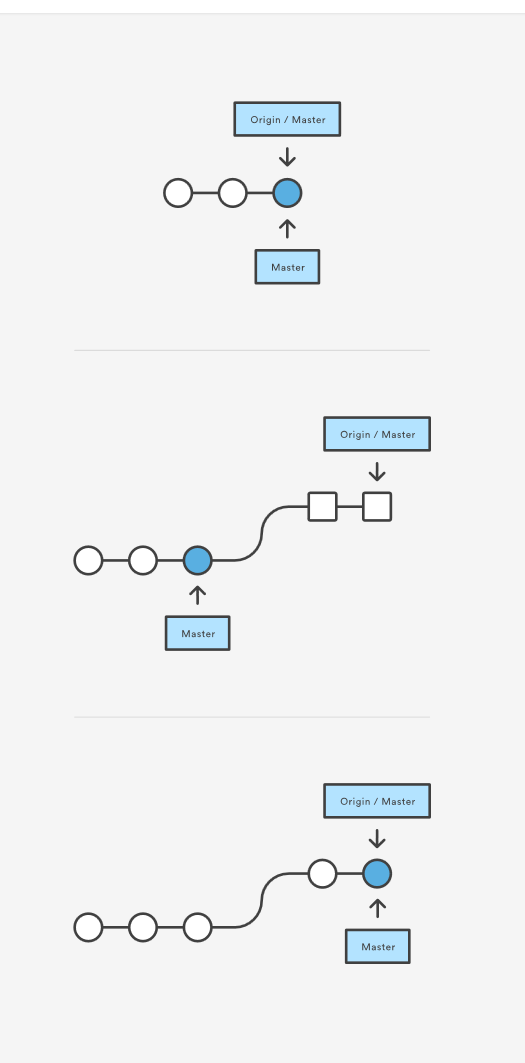
Same as the previous pull Instead of using git merge to integrate the remote branch with the local one, use [git rebase](https://www.atlassian.com/git/tutorials/rewriting-history/git-rebase).

git pull --verbose

Gives verbose output during a pull which displays the content being downloaded and the merge details.

## Git pull discussion

You can think of git pull as Git's version of svn update. It’s an easy way to synchronize your local repository with upstream changes. The following diagram explains each step of the pulling process.



You start out thinking your repository is synchronized, but then git fetch reveals that origin's version of master has progressed since you last checked it. Then git merge immediately integrates the remote master into the local one.

## Git pull and syncing

git pull is one of many commands that claim the responsibility of 'syncing' remote content. The [git remote](https://www.atlassian.com/git/tutorials/syncing) command is used to specify what remote endpoints the syncing commands will operate on. The [git push](https://www.atlassian.com/git/tutorials/syncing/git-push) command is used to upload content to a remote repository.

The git fetch command can be confused with git pull. They are both used to download remote content. An important safety distinction can me made between git pull and get fetch. git fetch can be considered the "safe" option whereas, git pull can be considered unsafe. git fetch will download the remote content and not alter the state of the local repository. Alternatively, git pull will download remote content and immediately attempt to change the local state to match that content. This may unintentionally cause the local repository to get in a conflicted state.

## Pulling via Rebase

The --rebase option can be used to ensure a linear history by preventing unnecessary merge commits. Many developers prefer rebasing over merging, since it’s like saying, "I want to put my changes on top of what everybody else has done." In this sense, using git pull with the --rebase flag is even more like svn update than a plain git pull.

In fact, pulling with --rebase is such a common workflow that there is a dedicated configuration option for it:

git config --global branch.autosetuprebase always

After running that command, all git pull commands will integrate via git rebase instead of git merge.

## Git Pull Examples

The following examples demonstrate how to use git pull in common scenarios:

## Default Behavior

git pull

Executing the default invocation of git pull will is equivalent to git fetch origin HEAD and git merge HEAD where HEAD is ref pointing to the current branch.

## Git pull on remotes

git checkout new\_feature  
git pull <remote repo>

This example first performs a checkout and switches to the <newfeature> branch. Following that, the git pull is executed with <remote repo> being passed. This will implicitly pull down the newfeature branch from <remote repo>. Once the download is complete it will initiate a git merge.

## Git pull rebase instead of merge

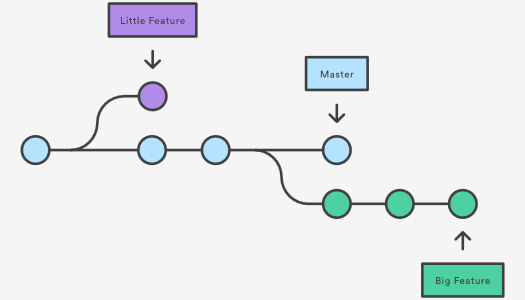
The following example demonstrates how to synchronize with the central repository's master branch using a rebase:

git checkout master  
git pull --rebase origin

This simply moves your local changes onto the top of what everybody else has already contributed.

# Git Branch

This document is an in-depth review of the git branch command and a discussion of the overall Git branching model. Branching is a feature available in most modern version control systems. Branching in other VCS's can be an expensive operation in both time and disk space. In Git, branches are a part of your everyday development process. Git branches are effectively a pointer to a snapshot of your changes. When you want to add a new feature or fix a bug—no matter how big or how small—you spawn a new branch to encapsulate your changes. This makes it harder for unstable code to get merged into the main code base, and it gives you the chance to clean up your future's history before merging it into the main branch.



The diagram above visualizes a repository with two isolated lines of development, one for a little feature, and one for a longer-running feature. By developing them in branches, it’s not only possible to work on both of them in parallel, but it also keeps the main master branch free from questionable code.

The implementation behind Git branches is much more lightweight than other version control system models. Instead of copying files from directory to directory, Git stores a branch as a reference to a commit. In this sense, a branch represents the tip of a series of commits—it's not a container for commits. The history for a branch is extrapolated through the commit relationships.

As you read, remember that Git branches aren't like SVN branches. Whereas SVN branches are only used to capture the occasional large-scale development effort, Git branches are an integral part of your everyday workflow. The following content will expand on the internal Git branching architecture.

## How it works

A branch represents an independent line of development. Branches serve as an abstraction for the edit/stage/commit process. You can think of them as a way to request a brand new working directory, staging area, and project history. New commits are recorded in the history for the current branch, which results in a fork in the history of the project.

The git branch command lets you create, list, rename, and delete branches. It doesn’t let you switch between branches or put a forked history back together again. For this reason, git branch is tightly integrated with the [git checkout](https://www.atlassian.com/git/tutorials/using-branches/git-checkout) and [git merge](https://www.atlassian.com/git/tutorials/using-branches/git-merge) commands.

## Common Options

git branch

List all of the branches in your repository. This is synonymous with git branch --list.

git branch <branch>

Create a new branch called <branch>. This does not check out the new branch.

git branch -d <branch>

Delete the specified branch. This is a “safe” operation in that Git prevents you from deleting the branch if it has unmerged changes.

git branch -D <branch>

Force delete the specified branch, even if it has unmerged changes. This is the command to use if you want to permanently throw away all of the commits associated with a particular line of development.

git branch -m <branch>

Rename the current branch to <branch>.

git branch -a

List all remote branches.

## Creating Branches

It's important to understand that branches are just pointers to commits. When you create a branch, all Git needs to do is create a new pointer, it doesn’t change the repository in any other way. If you start with a repository that looks like this:

Then, you create a branch using the following command:

git branch crazy-experiment

The repository history remains unchanged. All you get is a new pointer to the current commit:

Note that this only creates the new branch. To start adding commits to it, you need to select it with git checkout, and then use the standard git add and git commit commands.

## Creating remote branches

So far these examples have all demonstrated local branch operations. The git branch command also works on remote branches. In order to operate on remote branches, a remote repo must first be configured and added to the local repo config.

$ git remote add new-remote-repo https://bitbucket.com/user/repo.git

# Add remote repo to local repo config

$ git push <new-remote-repo> crazy-experiment~

# pushes the crazy-experiment branch to new-remote-repo

This command will push a copy of the local branch crazy-experiment to the remote repo <remote>.

## Deleting Branches

Once you’ve finished working on a branch and have merged it into the main code base, you’re free to delete the branch without losing any history:

git branch -d crazy-experiment

However, if the branch hasn’t been merged, the above command will output an error message:

error: The branch 'crazy-experiment' is not fully merged.

If you are sure you want to delete it, run 'git branch -D crazy-experiment'.

This protects you from losing access to that entire line of development. If you really want to delete the branch (e.g., it’s a failed experiment), you can use the capital -D flag:

git branch -D crazy-experiment

This deletes the branch regardless of its status and without warnings, so use it judiciously.

The previous commands will delete a local copy of a branch. The branch may still exist in remote repos. To delete a remote branch execute the following.

git push origin --delete crazy-experiment

Or

git push origin :crazy-experiment

This will push a delete signal to the remote origin repository that triggers a delete of the remote crazy-experiment branch.

## Summary

In this document we discussed Git's branching behavior and the git branch command. The git branch commands primary functions are to create, list, rename and delete branches. To operate further on the resulting branches the command is commonly used with other commands like git checkout. Learn more about git checkout branch operations; such as switching branches and merging branches, on the [git checkout](https://www.atlassian.com/git/tutorials/using-branches/git-checkout) page.

Compared to other VCSs, Git's branch operations are inexpensive and frequently used. This flexibility enables powerful [Git workflow](https://www.atlassian.com/git/tutorials/comparing-workflows) customization. For more info on Git workflows visit our extended workflow discussion pages: [The  
Feature Branch Workflow](https://www.atlassian.com/git/tutorials/comparing-workflows/feature-branch-workflow), [GitFlow Workflow](https://www.atlassian.com/git/tutorials/comparing-workflows/gitflow-workflow), and [Forking Workflow](https://www.atlassian.com/git/tutorials/comparing-workflows/forking-workflow).

# Git Checkout

This page is an examination of the git checkout command. It will cover usage examples and edge cases. In Git terms, a "checkout" is the act of switching between different versions of a target entity. The git checkout command operates upon three distinct entities: files, commits, and branches. In addition to the definition of "checkout" the phrase "checking out" is commonly used to imply the act of executing the git checkout command. In the [Undoing Changes](https://www.atlassian.com/git/tutorials/undoing-changes) topic, we saw how git checkout can be used to view old commits. The focus for the majority of this document will be checkout operations on branches.

Checking out branches is similar to checking out old commits and files in that the working directory is updated to match the selected branch/revision; however, new changes are saved in the project history—that is, it’s not a read-only operation.

## Checking out branches

The git checkout command lets you navigate between the branches created by git branch. Checking out a branch updates the files in the working directory to match the version stored in that branch, and it tells Git to record all new commits on that branch. Think of it as a way to select which line of development you’re working on.

Having a dedicated branch for each new feature is a dramatic shift from a traditional SVN workflow. It makes it ridiculously easy to try new experiments without the fear of destroying existing functionality, and it makes it possible to work on many unrelated features at the same time. In addition, branches also facilitate several collaborative workflows.

The git checkout command may occasionally be confused with git clone. The difference between the two commands is that clone works to fetch code from a remote repository, alternatively checkout works to switch between versions of code already on the local system.

## Usage: Existing branches

Assuming the repo you're working in contains pre-existing branches, you can switch between these branches using git checkout. To find out what branches are available and what the current branch name is, execute git branch.

$> git branch

master

another\_branch

feature\_inprogress\_branch

$> git checkout feature\_inprogress\_branch

The above example demonstrates how to view a list of available branches by executing the git branch command, and switch to a specified branch, in this case, the feature\_inprogress\_branch.

## New Branches

Git checkout works hand-in-hand with [git branch](https://www.atlassian.com/git/tutorials/using-branches). The git branch command can be used to create a new branch. When you want to start a new feature, you create a new branch off master using git branch new\_branch. Once created you can then use git checkout new\_branch to switch to that branch. Additionally, The git checkout command accepts a -b argument that acts as a convenience method which will create the new branch and immediately switch to it. You can work on multiple features in a single repository by switching between them with git checkout.

git checkout -b <new-branch>

The above example simultaneously creates and checks out <new-branch>. The -b option is a convenience flag that tells Git to run git branch <new-branch> before running git checkout <new-branch>.

git checkout -b <new-branch> <existing-branch>

By default git checkout -b will base the new-branch off the current HEAD. An optional additional branch parameter can be passed to git checkout. In the above example, <existing-branch> is passed which then bases new-branch off of existing-branch instead of the current HEAD.

## Switching Branches

Switching branches is a straightforward operation. Executing the following will point HEAD to the tip of <branchname>.

git checkout <branchname>

Git tracks a history of checkout operations in the reflog. You can execute git reflog to view the history.

## Git Checkout a Remote Branch

When collaborating with a team it is common to utilize remote repositories. These repositories may be hosted and shared or they may be another colleague's local copy. Each remote repository will contain its own set of branches. In order to checkout a remote branch you have to first fetch the contents of the branch.

git fetch --all

In modern versions of Git, you can then checkout the remote branch like a local branch.

git checkout <remotebranch>

Older versions of Git require the creation of a new branch based on the remote.

git checkout <remotebranch> origin/<remotebranch>

Additionally you can checkout a new local branch and reset it to the remote branches last commit.

git checkout -b <branchname>

git reset --hard origin/<branchname>

## Detached HEADS

Now that we’ve seen the three main uses of git checkout on branches, it's important to discuss the “detached HEAD” state. Remember that the HEAD is Git’s way of referring to the current snapshot. Internally, the git checkout command simply updates the HEAD to point to either the specified branch or commit. When it points to a branch, Git doesn't complain, but when you check out a commit, it switches into a “detached HEAD” state.

This is a warning telling you that everything you’re doing is “detached” from the rest of your project’s development. If you were to start developing a feature while in a detached HEAD state, there would be no branch allowing you to get back to it. When you inevitably check out another branch (e.g., to merge your feature in), there would be no way to reference your feature:

The point is, your development should always take place on a branch—never on a detached HEAD. This makes sure you always have a reference to your new commits. However, if you’re just looking at an old commit, it doesn’t really matter if you’re in a detached HEAD state or not.

## Summary

This page focused on usage of the git checkout command when changing branches. In summation, git checkout, when used on branches, alters the target of the HEAD ref. It can be used to create branches, switch branches, and checkout remote branches. The git checkout command is an essential tool for standard Git operation. It is a counterpart to [git merge](https://www.atlassian.com/git/tutorials/using-branches/git-merge). The git checkout and git merge commands are critical tools to enabling [git workflows.](https://www.atlassian.com/git/tutorials/comparing-workflows)

# Git Merge

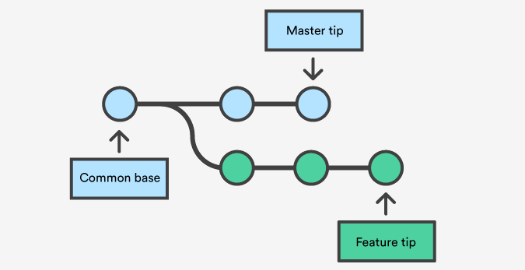
Merging is Git's way of putting a forked history back together again. The git merge command lets you take the independent lines of development created by git branch and integrate them into a single branch.

Note that all of the commands presented below merge into the current branch. The current branch will be updated to reflect the merge, but the target branch will be completely unaffected. Again, this means that git merge is often used in conjunction with git checkout for selecting the current branch and git branch -d for deleting the obsolete target branch.

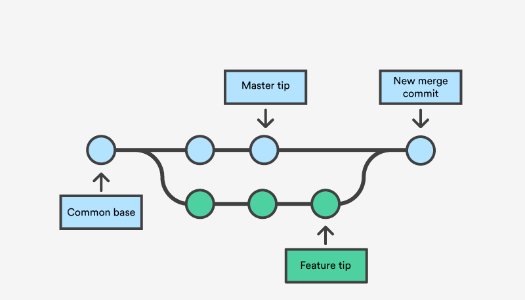
## How it works

Git merge will combine multiple sequences of commits into one unified history. In the most frequent use cases, git merge is used to combine two branches. The following examples in this document will focus on this branch merging pattern. In these scenarios, git merge takes two commit pointers, usually the branch tips, and will find a common base commit between them. Once Git finds a common base commit it will create a new "merge commit" that combines the changes of each queued merge commit sequence.

Say we have a new branch feature that is based off the master branch. We now want to merge this feature branch into master.



Invoking this command will merge the specified branch feature into the current branch, we'll assume master. Git will determine the merge algorithm automatically (discussed below).



Merge commits are unique against other commits in the fact that they have two parent commits. When creating a merge commit Git will attempt to auto magically merge the separate histories for you. If Git encounters a piece of data that is changed in both histories it will be unable to automatically combine them. This scenario is a version control conflict and Git will need user intervention to continue.

## Preparing to merge

Before performing a merge there are a couple of preparation steps to take to ensure the merge goes smoothly.

## Confirm the receiving branch

Execute git status to ensure that HEAD is pointing to the correct merge-receiving branch. If needed, execute git checkout <receiving> to switch to the receiving branch. In our case we will execute git checkout master.

## Fetch latest remote commits

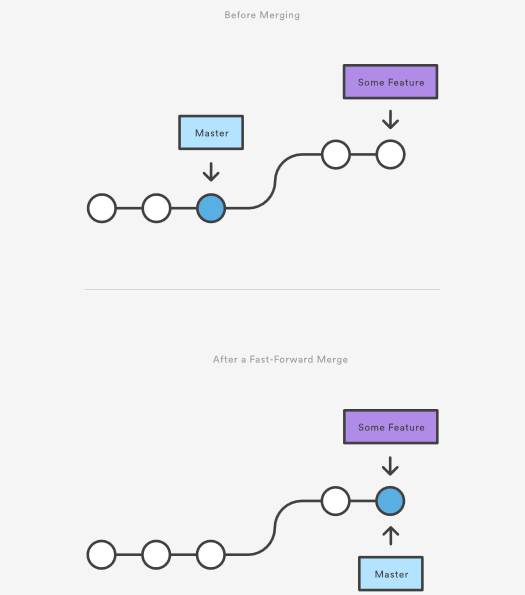
Make sure the receiving branch and the merging branch are up-to-date with the latest remote changes. Execute git fetch to pull the latest remote commits. Once the fetch is completed ensure the master branch has the latest updates by executing git pull.

## Merging

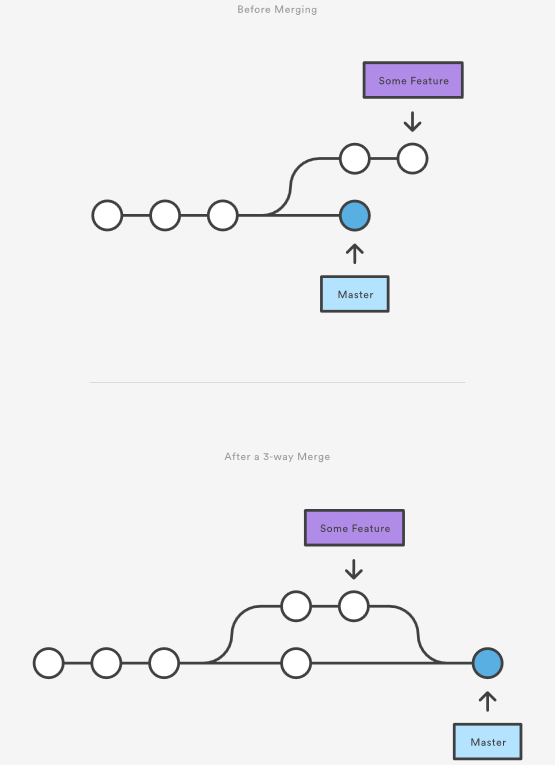
Once the previously discussed "preparing to merge" steps have been taken a merge can be initiated by executing git merge <branch name> where <branch name> is the name of the branch that will be merged into the receiving branch.

## Fast Forward Merge

A fast-forward merge can occur when there is a linear path from the current branch tip to the target branch. Instead of “actually” merging the branches, all Git has to do to integrate the histories is move (i.e., “fast forward”) the current branch tip up to the target branch tip. This effectively combines the histories, since all of the commits reachable from the target branch are now available through the current one. For example, a fast forward merge of some-feature into master would look something like the following:



However, a fast-forward merge is not possible if the branches have diverged. When there is not a linear path to the target branch, Git has no choice but to combine them via a 3-way merge. 3-way merges use a dedicated commit to tie together the two histories. The nomenclature comes from the fact that Git uses three commits to generate the merge commit: the two branch tips and their common ancestor.



While you can use either of these merge strategies, many developers like to use fast-forward merges (facilitated through [rebasing](https://www.atlassian.com/git/tutorials/rewriting-history/git-rebase)) for small features or bug fixes, while reserving 3-way merges for the integration of longer-running features. In the latter case, the resulting merge commit serves as a symbolic joining of the two branches.

Our first example demonstrates a fast-forward merge. The code below creates a new branch, adds two commits to it, then integrates it into the main line with a fast-forward merge.

# Start a new feature

git checkout -b new-feature master

# Edit some files

git add <file>

git commit -m "Start a feature"

# Edit some files

git add <file>

git commit -m "Finish a feature"

# Merge in the new-feature branch

git checkout master

git merge new-feature

git branch -d new-feature

This is a common workflow for short-lived topic branches that are used more as an isolated development than an organizational tool for longer-running features.

Also note that Git should not complain about the git branch -d, since new-feature is now accessible from the master branch.

In the event that you require a merge commit during a fast forward merge for record keeping purposes you can execute git merge with the --no-ff option.

git merge --no-ff <branch>

This command merges the specified branch into the current branch, but always generates a merge commit (even if it was a fast-forward merge). This is useful for documenting all merges that occur in your repository.

## 3-way merge

The next example is very similar, but requires a 3-way merge because master progresses while the feature is in-progress. This is a common scenario for large features or when several developers are working on a project simultaneously.

Start a new feature

git checkout -b new-feature master

# Edit some files

git add <file>

git commit -m "Start a feature"

# Edit some files

git add <file>

git commit -m "Finish a feature"

# Develop the master branch

git checkout master

# Edit some files

git add <file>

git commit -m "Make some super-stable changes to master"

# Merge in the new-feature branch

git merge new-feature

git branch -d new-feature

Note that it’s impossible for Git to perform a fast-forward merge, as there is no way to move master up to new-feature without backtracking.

For most workflows, new-feature would be a much larger feature that took a long time to develop, which would be why new commits would appear on master in the meantime. If your feature branch was actually as small as the one in the above example, you would probably be better off rebasing it onto master and doing a fast-forward merge. This prevents superfluous merge commits from cluttering up the project history.

## Resolving conflict

If the two branches you're trying to merge both changed the same part of the same file, Git won't be able to figure out which version to use. When such a situation occurs, it stops right before the merge commit so that you can resolve the conflicts manually.

The great part of Git's merging process is that it uses the familiar edit/stage/commit workflow to resolve merge conflicts. When you encounter a merge conflict, running the git status command shows you which files need to be resolved. For example, if both branches modified the same section of hello.py, you would see something like the following:

On branch master

Unmerged paths:

(use "git add/rm ..." as appropriate to mark resolution)

both modified: hello.py

## How conflicts are presented

When Git encounters a conflict during a merge, It will edit the content of the affected files with visual indicators that mark both sides of the conflicted content. These visual markers are: <<<<<<<, =======, and >>>>>>>. Its helpful to search a project for these indicators during a merge to find where conflicts need to be resolved.

here is some content not affected by the conflict

<<<<<<< master

this is conflicted text from master

=======

this is conflicted text from feature branch

>>>>>>> feature branch;

Generally the content before the ======= marker is the receiving branch and the part after is the merging branch.

Once you've identified conflicting sections, you can go in and fix up the merge to your liking. When you're ready to finish the merge, all you have to do is run git add on the conflicted file(s) to tell Git they're resolved. Then, you run a normal git commit to generate the merge commit. It’s the exact same process as committing an ordinary snapshot, which means it’s easy for normal developers to manage their own merges.

Note that merge conflicts will only occur in the event of a 3-way merge. It’s not possible to have conflicting changes in a fast-forward merge.

## Summary

This document is an overview of the git merge command. Merging is an essential process when working with Git. We discussed the internal mechanics behind a merge and the differences between a fast forward merge and a three way, true merge. Some key take-aways are:

1. Git merging combines sequences of commits into one unified history of commits.
2. There are two main ways Git will merge: Fast Forward and Three way
3. Git can automatically merge commits unless there are changes that conflict in both commit sequences.

This document integrated and referenced other Git commands like: [git branch](https://www.atlassian.com/git/tutorials/using-branches), [git pull](https://www.atlassian.com/git/tutorials/syncing#git-pull), and [git fetch](https://www.atlassian.com/git/tutorials/syncing#git-fetch). Visit their corresponding stand-alone pages for more information.

# Git merge conflicts

Version control systems are all about managing contributions between multiple distributed authors ( usually developers ). Sometimes multiple developers may try to edit the same content. If Developer A tries to edit code that Developer B is editing a conflict may occur. To alleviate the occurrence of conflicts developers will work in separate [isolated branches](https://www.atlassian.com/git/tutorials/using-branches). The git merge command's primary responsibility is to combine separate branches and resolve any conflicting edits.

## Understanding merge conflicts

Merging and conflicts are a common part of the Git experience. Conflicts in other version control tools like SVN can be costly and time-consuming. Git makes merging super easy. Most of the time, Git will figure out how to automatically integrate new changes.

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

## Types of merge conflicts

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

### Git fails to start the merge

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

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

### Git fails during the merge

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

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

## Creating a merge conflict

In order to get real familiar with merge conflicts, the next section will simulate a conflict to later examine and resolve. The example will be using a Unix-like command-line Git interface to execute the example simulation.

$ mkdir git-merge-test  
$ cd git-merge-test  
$ git init .  
$ echo "this is some content to mess with" > merge.txt  
$ git add merge.txt  
$ git commit -am"we are commiting the inital content"  
[master (root-commit) d48e74c] we are commiting the inital content  
1 file changed, 1 insertion(+)  
create mode 100644 merge.txt

This code example executes a sequence of commands that accomplish the following.

* Create a new directory named git-merge-test, change to that directory, and initialize it as a new Git repo.
* Create a new text file merge.txt with some content in it.
* Add merge.txt to the repo and commit it.

Now we have a new repo with one branch master and a file merge.txt with content in it. Next, we will create a new branch to use as the conflicting merge.

$ git checkout -b new\_branch\_to\_merge\_later  
$ echo "totally different content to merge later" > merge.txt  
$ git commit -am"edited the content of merge.txt to cause a conflict"  
[new\_branch\_to\_merge\_later 6282319] edited the content of merge.txt to cause a conflict  
1 file changed, 1 insertion(+), 1 deletion(-)

The proceeding command sequence achieves the following:

* create and check out a new branch named new\_branch\_to\_merge\_later
* overwrite the content in merge.txt
* commit the new content

With this new branch: new\_branch\_to\_merge\_later we have created a commit that overrides the content of merge.txt

git checkout master  
Switched to branch 'master'  
echo "content to append" >> merge.txt  
git commit -am"appended content to merge.txt"  
[master 24fbe3c] appended content to merge.tx  
1 file changed, 1 insertion(+)

This chain of commands checks out the master branch, appends content to merge.txt, and commits it. This now puts our example repo in a state where we have 2 new commits. One in the master branch and one in the new\_branch\_to\_merge\_later branch. At this time lets git merge new\_branch\_to\_merge\_later and see what happen!

$ git merge new\_branch\_to\_merge\_later  
Auto-merging merge.txt  
CONFLICT (content): Merge conflict in merge.txt  
Automatic merge failed; fix conflicts and then commit the result.

BOOM 💥. A conflict appears. Thanks, Git for letting us know about this!

## How to identify merge conflicts

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

$ git status  
On branch master  
You have unmerged paths.  
(fix conflicts and run "git commit")  
(use "git merge --abort" to abort the merge)  
  
Unmerged paths:  
(use "git add <file>..." to mark resolution)  
  
both modified:   merge.txt

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

$ cat merge.txt  
<<<<<<< HEAD  
this is some content to mess with  
content to append  
=======  
totally different content to merge later  
>>>>>>> new\_branch\_to\_merge\_later

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

* <<<<<<< HEAD
* =======
* >>>>>>> new\_branch\_to\_merge\_later

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

## How to resolve merge conflicts using the command line

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

this is some content to mess with  
content to append  
totally different content to merge later

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

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

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

## Git commands that can help resolve merge conflicts

### General tools

git status

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

git log --merge

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

git diff

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

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

git checkout

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

git reset --mixed

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

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

git merge --abort

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

git reset

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

## Summary

Merge conflicts can be an intimidating experience. Luckily, Git offers powerful tools to help navigate and resolve conflicts. Git can handle most merges on its own with automatic merging features. A conflict arises when two separate branches have made edits to the same line in a file, or when a file has been deleted in one branch but edited in the other. Conflicts will most likely happen when working in a team environment.

There are many tools to help resolve merge conflicts. Git has plenty of command line tools we discussed here. For more detailed information on these tools visit stand-alone pages for [git log](https://www.atlassian.com/git/tutorials/git-log), [git reset](https://www.atlassian.com/git/tutorials/undoing-changes/git-reset), [git status](https://www.atlassian.com/git/tutorials/inspecting-a-repository), [git checkout](https://www.atlassian.com/git/tutorials/using-branches/git-checkout), and [git reset](https://www.atlassian.com/git/tutorials/undoing-changes/git-reset). In addition to the Git, many third-party tools offer streamlined merge conflict support features.

# Git Merge Strategy Options and Examples

When a piece of work is complete, tested and ready to be merged back into the main line of development, your team has some policy choices to make. What are your merge strategy options? In this article we'll examine the possibilities and then provide some notes on how Atlassian operates. Hopefully at the end you'll have the tools to decide what works best for your team.

## Git Merge Strategies

A merge happens when combining two branches. Git will take two (or more) commit pointers and attempt to find a common base commit between them. Git has several different methods to find a base commit, these methods are called "merge strategies". Once Git finds a common base commit it will create a new "merge commit" that combines the changes of the specified merge commits. Technically, a merge commit is a regular commit which just happens to have two parent commits.

git merge will automatically select a merge strategy unless explicitly specified. The git merge and git pull commands can be passed an -s (strategy) option. The -s option can be appended with the name of the desired merge strategy. If not explicitly specified, Git will select the most appropriate merge strategy based on the provided branches. The following is a list of the available merge strategies.

### Recursive

git merge -s recursive branch1 branch2

This operates on two heads. Recursive is the default merge strategy when pulling or merging one branch. Additionally this can detect and handle merges involving renames, but currently cannot make use of detected copies. This is the default merge strategy when pulling or merging one branch.

### Resolve

git merge -s resolve branch1 branch2

This can only resolve two heads using a 3-way merge algorithm. It tries to carefully detect cris-cross merge ambiguities and is considered generally safe and fast.

### Octopus

git merge -s octopus branch1 branch2 branch3 branchN

The default merge strategy for more than two heads. When more than one branch is passed octopus is automatically engaged. If a merge has conflicts that need manual resolution octopus will refuse the merge attempt. It is primarily used for bundling similar feature branch heads together.

### Ours

git merge -s ours branch1 branch2 branchN

The Ours strategy operates on multiple N number of branches. The output merge result is always that of the current branch HEAD. The "ours" term implies the preference effectively ignoring all changes from all other branches. It is intended to be used to combine history of similar feature branches.

### Subtree

git merge -s subtree branchA branchB

This is an extension of the recursive strategy. When merging A and B, if B is a child subtree of A, B is first updated to reflect the tree structure of A, This update is also done to the common ancestor tree that is shared between A and B.

## Types of Git Merge Strategies

### Explicit Merges

Explicit merges are the default merge type. The 'explicit' part is that they create a new merge commit. This alters the commit history and explicitly shows where a merge was executed. The merge commit content is also explicit in the fact that it shows which commits were the parents of the merge commit. Some teams avoid explicit merges because arguably the merge commits add "noise" to the history of the project.

### implicit merge via rebase or fast-forward merge

Whereas explicit merges create a merge commit, implicit merges do not. An implicit merge takes a series of commits from a specified branch head and applies them to the top of a target branch. Implicit merges are triggered by [rebase events](https://www.atlassian.com/git/tutorials/rewriting-history/git-rebase), or [fast forward merges](https://www.atlassian.com/git/tutorials/using-branches/git-merge). An implicit merge is an ad-hoc selection of commits from a specified branch.

### Squash on merge, generally without explicit merge

Another type of implicit merge is a squash. A squash can be performed during an [interactive rebase](https://www.atlassian.com/git/tutorials/rewriting-history/git-rebase). A squash merge will take the commits from a target branch and combine or squash them in to one commit. This commit is then appended to the HEAD of the merge base branch. A squash is commonly used to keep a 'clean history' during a merge. The target merge branch can have a verbose history of frequent commits. When squashed and merged the target branches commit history then becomes a singular squashed 'branch commit'. This technique is useful with git workflows that utilize feature branches.

## Recursive Git Merge Strategy Options

The 'recursive' strategy introduced above, has its own subset of additional operation options.

ours

Not to be confused with the Ours merge strategy. This option conflicts to be auto-resolved cleanly by favoring the 'our' version. Changes from the 'theirs' side are automatically incorporated if they do not conflict.

theirs

The opposite of the 'ours' strategy. the "theirs" option favors the foreign merging tree in conflict resolution.

patience

This option spends extra time to avoid mis-merges on unimportant matching lines. This options is best used when branches to be merged have extremely diverged.

diff-algorithim

This option allows specification of an explicit diff-algorithim. The diff-algorithims are shared with the [git diff](https://www.atlassian.com/git/tutorials/saving-changes/git-diff) command.

ignore-\*  
  
    ignore-space-change  
    ignore-all-space  
    ignore-space-at-eol  
    ignore-cr-at-eol

A set of options that target whitespace characters. Any line that matches the subset of the passed option will be ignored.

renormalize

This option runs a check-out and check-in on all of the tree git trees while resolving a three-way merge. This option is intended to be used with merging branches with differing checkin/checkout states.

no-normalize

Disables the renormalize option. This overrides the merge.renormalize configuration variable.

no-renames

This option will ignore renamed files during the merge.

find-renames=n

This is the default behavior. The recursive merge will honor file renames. The n parameter can be used to pass a threshold for rename similarity. The default n value is 100%.

subtree

This option borrows from the `subtree` strategy. Where the strategy operates on two trees and modifies how to make them match on a shared ancestor, this option instead operates on the path metadata of the tree to make them match.

## Our Git Merge Policy

Atlassian strongly prefers using explicit merges. The reason is very simple: explicit merges provide great traceability and context on the features being merged. A local history clean-up rebase before sharing a feature branch for review is absolutely encouraged, but this does not change the policy at all. It augments it.

# Comparing Workflows

A Git Workflow is a recipe or recommendation for how to use Git to accomplish work in a consistent and productive manner. Git workflows encourage users to leverage Git effectively and consistently. Git offers a lot of flexibility in how users manage changes. Given Git's focus on flexibility, there is no standardized process on how to interact with Git. When working with a team on a Git managed project, it’s important to make sure the team is all in agreement on how the flow of changes will be applied. To ensure the team is on the same page, an agreed upon Git workflow should be developed or selected. There are several publicized Git workflows that may be a good fit for your team. Here, we’ll be discussing some of these workflow options.

The array of possible workflows can make it hard to know where to begin when implementing Git in the workplace. This page provides a starting point by surveying the most common Git workflows for software teams.

As you read through, remember that these workflows are designed to be guidelines rather than concrete rules. We want to show you what’s possible, so you can mix and match aspects from different workflows to suit your individual needs.

## What is a successful Git workflow?

When evaluating a workflow for your team, it's most important that you consider your team’s culture. You want the workflow to enhance the effectiveness of your team and not be a burden that limits productivity. Some things to consider when evaluating a Git workflow are:

* Does this workflow scale with team size?
* Is it easy to undo mistakes and errors with this workflow?
* Does this workflow impose any new unnecessary cognitive overhead to the team?

## Centralized Workflow

The Centralized Workflow is a great Git workflow for teams transitioning from SVN. Like Subversion, the Centralized Workflow uses a central repository to serve as the single point-of-entry for all changes to the project. Instead of trunk, the default development branch is called master and all changes are committed into this branch. This workflow doesn’t require any other branches besides master.

Transitioning to a distributed version control system may seem like a daunting task, but you don’t have to change your existing workflow to take advantage of Git. Your team can develop projects in the exact same way as they do with Subversion.

However, using Git to power your development workflow presents a few advantages over SVN. First, it gives every developer their own local copy of the entire project. This isolated environment lets each developer work independently of all other changes to a project - they can add commits to their local repository and completely forget about upstream developments until it's convenient for them.

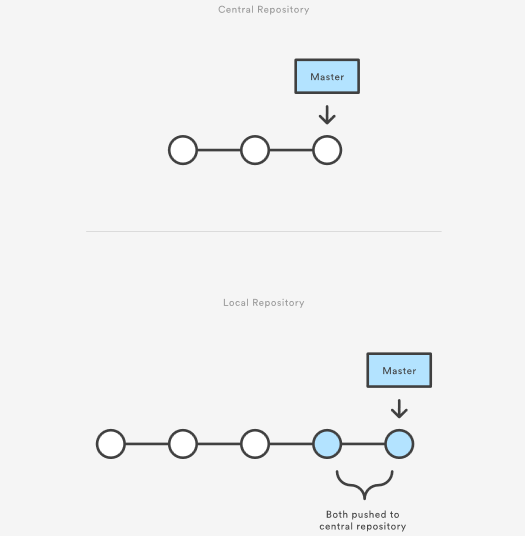
Second, it gives you access to Git’s robust branching and merging model. Unlike SVN, Git branches are designed to be a fail-safe mechanism for integrating code and sharing changes between repositories. The Centralized Workflow is similar to other workflows in its utilization of a remote server-side hosted repository that developers push and pull form. Compared to other workflows, the Centralized Workflow has no defined pull request or forking patterns. A Centralized Workflow is generally better suited for teams migrating from SVN to Git and smaller size teams.

## How it works

Developers start by cloning the central repository. In their own local copies of the project, they edit files and commit changes as they would with SVN; however, these new commits are stored locally - they’re completely isolated from the central repository. This lets developers defer synchronizing upstream until they’re at a convenient break point.

To publish changes to the official project, developers "push" their local master branch to the central repository. This is the equivalent of svn commit, except that it adds all of the local commits that aren’t already in the central master branch.

### Initialize the central repository



First, someone needs to create the central repository on a server. If it’s a new project, you can initialize an empty repository. Otherwise, you’ll need to import an existing Git or SVN repository.

Central repositories should always be bare repositories (they shouldn’t have a working directory), which can be created as follows:

ssh user@host git init --bare /path/to/repo.git

Be sure to use a valid SSH username for user, the domain or IP address of your server for host, and the location where you'd like to store your repo for /path/to/repo.git. Note that the .git extension is conventionally appended to the repository name to indicate that it’s a bare repository.

### Hosted central repositories

Central repositories are often created through 3rd party Git hosting services like [Bitbucket Cloud](https://bitbucket.org/product) or [Bitbucket Server](https://bitbucket.org/product/enterprise). The process of initializing a bare repository discussed above is handled for you by the hosting service. The hosting service will then provide an address for the central repository to access from your local repository.

### Clone the central repository

Next, each developer creates a local copy of the entire project. This is accomplished via the [git clone](https://www.atlassian.com/git/tutorials/setting-up-a-repository/git-clone) command:

git clone ssh://user@host/path/to/repo.git

When you clone a repository, Git automatically adds a shortcut called origin that points back to the “parent” repository, under the assumption that you'll want to interact with it further on down the road.

### Make changes and commit

Once the repository is cloned locally, a developer can make changes using the standard Git commit process: edit, stage, and commit. If you’re not familiar with the staging area, it’s a way to prepare a commit without having to include every change in the working directory. This lets you create highly focused commits, even if you’ve made a lot of local changes.

git status # View the state of the repo

git add <some-file> # Stage a file

git commit # Commit a file</some-file>

Remember that since these commands create local commits, John can repeat this process as many times as he wants without worrying about what’s going on in the central repository. This can be very useful for large features that need to be broken down into simpler, more atomic chunks.

### Push new commits to central repository

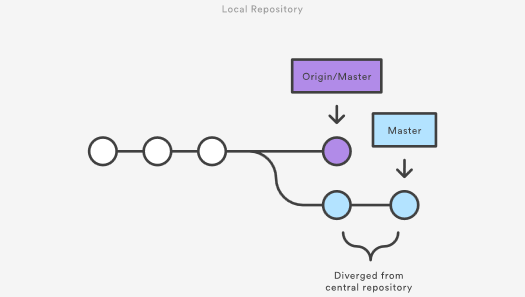
Once the local repository has new changes committed. These change will need to be pushed to share with other developers on the project.

git push origin master

This command will push the new committed changes to the central repository. When pushing changes to the central repository, it is possible that updates from another developer have been previously pushed that contain code which conflict with the intended push updates. Git will output a message indicating this conflict. In this situation, git pull will first need to be executed. This conflict scenario will be expanded on in the following section.

### Managing conflicts

The central repository represents the official project, so its commit history should be treated as sacred and immutable. If a developer’s local commits diverge from the central repository, Git will refuse to push their changes because this would overwrite official commits.



Before the developer can publish their feature, they need to fetch the updated central commits and rebase their changes on top of them. This is like saying, “I want to add my changes to what everyone else has already done.” The result is a perfectly linear history, just like in traditional SVN workflows.

If local changes directly conflict with upstream commits, Git will pause the rebasing process and give you a chance to manually resolve the conflicts. The nice thing about Git is that it uses the same git status and git add commands for both generating commits and resolving merge conflicts. This makes it easy for new developers to manage their own merges. Plus, if they get themselves into trouble, Git makes it very easy to abort the entire rebase and try again (or go find help).

## Example

Let’s take a general example at how a typical small team would collaborate using this workflow. We’ll see how two developers, John and Mary, can work on separate features and share their contributions via a centralized repository.

### John works on his feature

In his local repository, John can develop features using the standard Git commit process: edit, stage, and commit.

Remember that since these commands create local commits, John can repeat this process as many times as he wants without worrying about what’s going on in the central repository.

### Mary works on her feature

Meanwhile, Mary is working on her own feature in her own local repository using the same edit/stage/commit process. Like John, she doesn’t care what’s going on in the central repository, and she really doesn’t care what John is doing in his local repository, since all local repositories are private.

### John publishes his feature

Once John finishes his feature, he should publish his local commits to the central repository so other team members can access it. He can do this with the [git push](https://www.atlassian.com/git/tutorials/syncing/git-push) command, like so:

git push origin master

Remember that origin is the remote connection to the central repository that Git created when John cloned it. The master argument tells Git to try to make the origin’s master branch look like his local master branch. Since the central repository hasn’t been updated since John cloned it, this won’t result in any conflicts and the push will work as expected.

### Mary tries to publish her feature

Let’s see what happens if Mary tries to push her feature after John has successfully published his changes to the central repository. She can use the exact same push command:

git push origin master

But, since her local history has diverged from the central repository, Git will refuse the request with a rather verbose error message:

error: failed to push some refs to '/path/to/repo.git'

hint: Updates were rejected because the tip of your current branch is behind

hint: its remote counterpart. Merge the remote changes (e.g. 'git pull')

hint: before pushing again.

hint: See the 'Note about fast-forwards' in 'git push --help' for details.

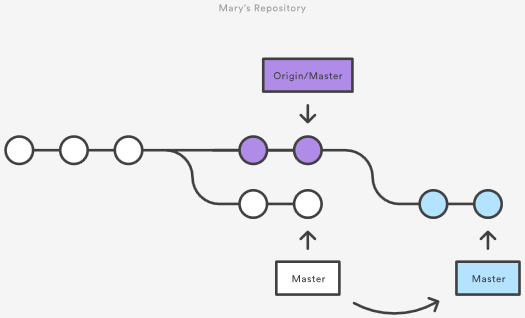
This prevents Mary from overwriting official commits. She needs to pull John’s updates into her repository, integrate them with her local changes, and then try again.

### Mary rebases on top of John’s commit(s)

Mary can use [git pull](https://www.atlassian.com/git/tutorials/syncing/git-pull) to incorporate upstream changes into her repository. This command is sort of like svn update—it pulls the entire upstream commit history into Mary’s local repository and tries to integrate it with her local commits:

git pull --rebase origin master

The --rebase option tells Git to move all of Mary’s commits to the tip of the master branch after synchronising it with the changes from the central repository, as shown below:



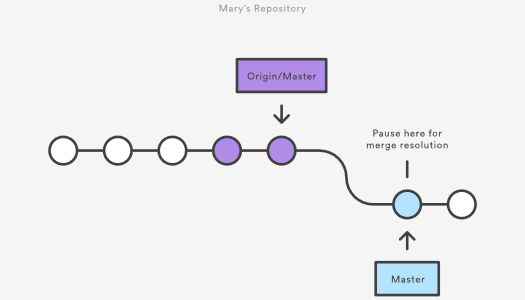
The pull would still work if you forgot this option, but you would wind up with a superfluous “merge commit” every time someone needed to synchronize with the central repository. For this workflow, it’s always better to rebase instead of generating a merge commit.

### Mary resolves a merge conflict

Rebasing works by transferring each local commit to the updated master branch one at a time. This means that you catch merge conflicts on a commit-by-commit basis rather than resolving all of them in one massive merge commit. This keeps your commits as focused as possible and makes for a clean project history. In turn, this makes it much easier to figure out where bugs were introduced and, if necessary, to roll back changes with minimal impact on the project.

If Mary and John are working on unrelated features, it’s unlikely that the rebasing process will generate conflicts. But if it does, Git will pause the rebase at the current commit and output the following message, along with some relevant instructions:

CONFLICT (content): Merge conflict in <some-file>



The great thing about Git is that anyone can resolve their own merge conflicts. In our example, Mary would simply run a [git status](https://www.atlassian.com/git/tutorials/inspecting-a-repository/git-status) to see where the problem is. Conflicted files will appear in the Unmerged paths section:

# Unmerged paths:

# (use "git reset HEAD <some-file>..." to unstage)

# (use "git add/rm <some-file>..." as appropriate to mark resolution)

#

# both modified: <some-file>

Then, she’ll edit the file(s) to her liking. Once she’s happy with the result, she can stage the file(s) in the usual fashion and let [git rebase](https://www.atlassian.com/git/tutorials/rewriting-history/git-rebase) do the rest:

git add <some-file>

git rebase --continue

And that’s all there is to it. Git will move on to the next commit and repeat the process for any other commits that generate conflicts.

If you get to this point and realize and you have no idea what’s going on, don’t panic. Just execute the following command and you’ll be right back to where you started:

git rebase --abort

### Mary successfully publishes her feature

After she’s done synchronizing with the central repository, Mary will be able to publish her changes successfully:

git push origin master

### Where to go from here

As you can see, it’s possible to replicate a traditional Subversion development environment using only a handful of Git commands. This is great for transitioning teams off of SVN, but it doesn’t leverage the distributed nature of Git.

The Centralized Workflow is great for small teams. The conflict resolution process detailed above can form a bottleneck as your team scales in size. If your team is comfortable with the Centralized Workflow but wants to streamline its collaboration efforts, it's definitely worth exploring the benefits of the [Feature Branch Workflow](https://www.atlassian.com/git/tutorials/comparing-workflows/feature-branch-workflow). By dedicating an isolated branch to each feature, it’s possible to initiate in-depth discussions around new additions before integrating them into the official project.

## Other common workflows

The Centralized Workflow is essentially a building block for other Git workflows. Most popular Git workflows will have some sort of centralized repo that individual developers will push and pull from. Below we will briefly discuss some other popular Git workflows. These extended workflows offer more specialized patterns in regard to managing branches for feature development, hot fixes, and eventual release.

## Feature branching

Feature Branching is a logical extension of Centralized Workflow. The core idea behind the [Feature Branch Workflow](http://www.atlassian.com/git/tutorials/comparing-workflows/feature-branch-workflow) is that all feature development should take place in a dedicated branch instead of the master branch. This encapsulation makes it easy for multiple developers to work on a particular feature without disturbing the main codebase. It also means the master branch should never contain broken code, which is a huge advantage for continuous integration environments.

## Gitflow Workflow

The [Gitflow Workflow](http://www.atlassian.com/git/tutorials/comparing-workflows/gitflow-workflow) was first published in a highly regarded 2010 blog post from [Vincent Driessen at nvie](http://nvie.com/posts/a-successful-git-branching-model/). The Gitflow Workflow defines a strict branching model designed around the project release. This workflow doesn’t add any new concepts or commands beyond what’s required for the Feature Branch Workflow. Instead, it assigns very specific roles to different branches and defines how and when they should interact.

## Forking Workflow

The [Forking Workflow](http://www.atlassian.com/git/tutorials/comparing-workflows/forking-workflow) is fundamentally different than the other workflows discussed in this tutorial. Instead of using a single server-side repository to act as the “central” codebase, it gives every developer a server-side repository. This means that each contributor has not one, but two Git repositories: a private local one and a public server-side one.

## Guidelines

There is no one size fits all Git workflow. As previously stated, it’s important to develop a Git workflow that is a productivity enhancement for your team. In addition to team culture, a workflow should also complement business culture. Git features like branches and tags should complement your business’s release schedule. If your team is using [task tracking project management software](https://www.atlassian.com/software/jira) you may want to use branches that correspond with tasks in progress. In addition, some guidelines to consider when deciding on a workflow are:

### Short-lived branches

The longer a branch lives separate from the production branch, the higher the risk for merge conflicts and deployment challenges. Short-lived branches promote cleaner merges and deploys.

### Minimize and simplify reverts

It’s important to have a workflow that helps proactively prevent merges that will have to be reverted. A workflow that tests a branch before allowing it to be merged into the master branch is an example. However, accidents do happen. That being said, it’s beneficial to have a workflow that allows for easy reverts that will not disrupt the flow for other team members.

### Match a release schedule

A workflow should complement your business’s software development release cycle. If you plan to release multiple times a day, you will want to keep your master branch stable. If your release schedule is less frequent, you may want to consider using Git tags to tag a branch to a version.

## Summary

In this document we discussed Git workflows. We took an in-depth look at a Centralized Workflow with practical examples. Expanding on the Centralized Workflow we discussed additional specialized workflows. Some key takeaways from this document are:

* There is no one-size-fits-all Git workflow
* A workflow should be simple and enhance the productivity of your team
* Your business requirements should help shape your Git workflow

To read about the next Git workflow check out our comprehensive breakdown of the [Feature Branch Workflow](https://www.atlassian.com/git/tutorials/comparing-workflows/feature-branch-workflow).

# Git Feature Branch Workflow

The core idea behind the Feature Branch Workflow is that all feature development should take place in a dedicated branch instead of the master branch. This encapsulation makes it easy for multiple developers to work on a particular feature without disturbing the main codebase. It also means the master branch will never contain broken code, which is a huge advantage for continuous integration environments.

Encapsulating feature development also makes it possible to leverage pull requests, which are a way to initiate discussions around a branch. They give other developers the opportunity to sign off on a feature before it gets integrated into the official project. Or, if you get stuck in the middle of a feature, you can open a pull request asking for suggestions from your colleagues. The point is, pull requests make it incredibly easy for your team to comment on each other’s work.

The Git Feature Branch Workflow is a composable workflow that can be leveraged by other high-level Git workflows. We discussed other Git workflows on [the Git workflow overview page](http://www.atlassian.com/git/tutorials/comparing-workflows/). Git Feature Branch Workflow is branching model focused, meaning that it is a guiding framework for managing and creating branches. Other workflows are more repo focused. The Git Feature Branch Workflow can be incorporated into other workflows. The [Gitflow](http://www.atlassian.com/git/tutorials/comparing-workflows/gitflow-workflow), and [Git Forking Workflows](http://www.atlassian.com/git/tutorials/comparing-workflows/forking-workflow) traditionally use a Git Feature Branch Workflow in regards to their branching models.

## How it works

The Feature Branch Workflow assumes a central repository, and master represents the official project history. Instead of committing directly on their local master branch, developers create a new branch every time they start work on a new feature. Feature branches should have descriptive names, like animated-menu-items or issue-#1061. The idea is to give a clear, highly-focused purpose to each branch. Git makes no technical distinction between the master branch and feature branches, so developers can edit, stage, and commit changes to a feature branch.

In addition, feature branches can (and should) be pushed to the central repository. This makes it possible to share a feature with other developers without touching any official code. Since master is the only “special” branch, storing several feature branches on the central repository doesn’t pose any problems. Of course, this is also a convenient way to back up everybody’s local commits. The following is a walk-through of the life-cycle of a feature branch.

### Start with the master branch

All feature branches are created off the latest code state of a project. This guide assumes this is maintained and updated in the master branch.

git checkout master

git fetch origin

git reset --hard origin/master

This switches the repo to the master branch, pulls the latest commits and resets the repo's local copy of master to match the latest version.

### Create a new-branch

Use a separate branch for each feature or issue you work on. After creating a branch, check it out locally so that any changes you make will be on that branch.

git checkout -b new-feature

This checks out a branch called new-feature based on master, and the -b flag tells Git to create the branch if it doesn’t already exist.

### Update, add, commit, and push changes

On this branch, edit, stage, and commit changes in the usual fashion, building up the feature with as many commits as necessary. Work on the feature and make commits like you would any time you use Git. When ready, push your commits, updating the feature branch on Bitbucket.

git status

git add <some-file>

git commit

### Push feature branch to remote

It’s a good idea to push the feature branch up to the central repository. This serves as a convenient backup, when collaborating with other developers, this would give them access to view commits to the new branch.

git push -u origin new-feature

This command pushes new-feature to the central repository (origin), and the -u flag adds it as a remote tracking branch. After setting up the tracking branch, git push can be invoked without any parameters to automatically push the new-feature branch to the central repository. To get feedback on the new feature branch, create a pull request in a repository management solution like [Bitbucket Cloud](https://bitbucket.org/product) or [Bitbucket Server](https://www.atlassian.com/software/bitbucket/server). From there, you can add reviewers and make sure everything is good to go before merging.

### Resolve feedback

Now teammates comment and approve the pushed commits. Resolve their comments locally, commit, and push the suggested changes to Bitbucket. Your updates appear in the pull request.

### Merge your pull request

Before you merge, you may have to resolve merge conflicts if others have made changes to the repo. When your pull request is approved and conflict-free, you can add your code to the master branch. Merge from the pull request in Bitbucket.

## Pull requests

Aside from isolating feature development, branches make it possible to discuss changes via pull requests. Once someone completes a feature, they don’t immediately merge it into master. Instead, they push the feature branch to the central server and file a pull request asking to merge their additions into master. This gives other developers an opportunity to review the changes before they become a part of the main codebase.

Code review is a major benefit of pull requests, but they’re actually designed to be a generic way to talk about code. You can think of pull requests as a discussion dedicated to a particular branch. This means that they can also be used much earlier in the development process. For example, if a developer needs help with a particular feature, all they have to do is file a pull request. Interested parties will be notified automatically, and they’ll be able to see the question right next to the relevant commits.

Once a pull request is accepted, the actual act of publishing a feature is much the same as in the [Centralized Workflow](https://www.atlassian.com/git/tutorials/comparing-workflows). First, you need to make sure your local master is synchronized with the upstream master. Then, you merge the feature branch into master and push the updated master back to the central repository.

Pull requests can be facilitated by product repository management solutions like Bitbucket Cloud or Bitbucket Server. View the Bitbucket Server pull requests documentation for an example.

## Example

The following is an example of the type of scenario in which a feature branching workflow is used. The scenario is that of a team doing code review around on a new feature pull request. This is one example of the many purposes this model can be used for.

### Mary begins a new feature

Before she starts developing a feature, Mary needs an isolated branch to work on. She can request a new branch with the following command:

git checkout -b marys-feature master

This checks out a branch called marys-feature based on master, and the -b flag tells Git to create the branch if it doesn’t already exist. On this branch, Mary edits, stages, and commits changes in the usual fashion, building up her feature with as many commits as necessary:

git status

git add <some-file>

git commit

### Mary goes to lunch

Mary adds a few commits to her feature over the course of the morning. Before she leaves for lunch, it’s a good idea to push her feature branch up to the central repository. This serves as a convenient backup, but if Mary was collaborating with other developers, this would also give them access to her initial commits.

git push -u origin marys-feature

This command pushes marys-feature to the central repository (origin), and the -u flag adds it as a remote tracking branch. After setting up the tracking branch, Mary can call git push without any parameters to push her feature.

### Mary finishes her feature

When Mary gets back from lunch, she completes her feature. Before merging it into master, she needs to file a pull request letting the rest of the team know she's done. But first, she should make sure the central repository has her most recent commits:

git push

Then, she files the pull request in her Git GUI asking to merge marys-feature into master, and team members will be notified automatically. The great thing about pull requests is that they show comments right next to their related commits, so it's easy to ask questions about specific changesets.

### Bill receives the pull request

Bill gets the pull request and takes a look at marys-feature. He decides he wants to make a few changes before integrating it into the official project, and he and Mary have some back-and-forth via the pull request.

### Mary makes the changes

To make the changes, Mary uses the exact same process as she did to create the first iteration of her feature. She edits, stages, commits, and pushes updates to the central repository. All her activity shows up in the pull request, and Bill can still make comments along the way.

If he wanted, Bill could pull marys-feature into his local repository and work on it on his own. Any commits he added would also show up in the pull request.

### Mary publishes her feature

Once Bill is ready to accept the pull request, someone needs to merge the feature into the stable project (this can be done by either Bill or Mary):

git checkout master

git pull

git pull origin marys-feature

git push

This process often results in a merge commit. Some developers like this because it’s like a symbolic joining of the feature with the rest of the code base. But, if you’re partial to a linear history, it’s possible to rebase the feature onto the tip of master before executing the merge, resulting in a fast-forward merge.

Some GUI’s will automate the pull request acceptance process by running all of these commands just by clicking an “Accept” button. If yours doesn’t, it should at least be able to automatically close the pull request when the feature branch gets merged into master.

Meanwhile, John is doing the exact same thing

While Mary and Bill are working on marys-feature and discussing it in her pull request, John is doing the exact same thing with his own feature branch. By isolating features into separate branches, everybody can work independently, yet it’s still trivial to share changes with other developers when necessary.

## Summary

In this document, we discussed the Git Feature Branch Workflow. This workflow helps organize and track branches that are focused on business domain feature sets. Other Git workflows like the Git Forking Workflow and the Gitflow Workflow are repo focused and can leverage the Git Feature Branch Workflow to manage their branching models. This document demonstrated a high-level code example and fictional example for implementing the Git Feature Branch Workflow. Some key associations to make with the Feature Branch Workflow are:

* focused on branching patterns
* can be leveraged by other repo oriented workflows
* promotes collaboration with team members through pull requests and merge reviews

Utilizing [git rebase](https://www.atlassian.com/git/tutorials/rewriting-history/git-rebase) during the review and merge stages of a feature branch will create enforce a cohesive Git history of feature merges. A feature branching model is a great tool to promote collaboration within a team environment.

Go one click deeper into Git workflows by reading our comprehensive tutorial of the [Gitflow Workflow.](https://www.atlassian.com/git/tutorials/comparing-workflows/gitflow-workflow)

# Gitflow Workflow

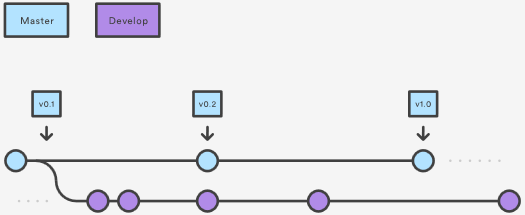
Gitflow Workflow is a Git workflow design that was first published and made popular by [Vincent Driessen at nvie](http://nvie.com/posts/a-successful-git-branching-model/). The Gitflow Workflow defines a strict branching model designed around the project release. This provides a robust framework for managing larger projects.

Gitflow is ideally suited for projects that have a scheduled release cycle. This workflow doesn’t add any new concepts or commands beyond what’s required for the [Feature Branch Workflow](http://www.atlassian.com/git/tutorials/comparing-workflows/feature-branch-workflow). Instead, it assigns very specific roles to different branches and defines how and when they should interact. In addition to feature branches, it uses individual branches for preparing, maintaining, and recording releases. Of course, you also get to leverage all the benefits of the Feature Branch Workflow: pull requests, isolated experiments, and more efficient collaboration.

## Getting Started

Gitflow is really just an abstract idea of a Git workflow. This means it dictates what kind of branches to set up and how to merge them together. We will touch on the purposes of the branches below. The git-flow toolset is an actual command line tool that has an installation process. The installation process for git-flow is straightforward. Packages for git-flow are available on multiple operating systems. On OSX systems, you can execute brew install git-flow. On windows you will need to [download and install git-flow](https://git-scm.com/download/win). After installing git-flow you can use it in your project by executing git flow init. Git-flow is a wrapper around Git. The git flow init command is an extension of the default [git init](https://www.atlassian.com/git/tutorials/setting-up-a-repository/git-init) command and doesn't change anything in your repository other than creating branches for you.

## How it works



### Develop and Master Branches

Instead of a single master branch, this workflow uses two branches to record the history of the project. The master branch stores the official release history, and the develop branch serves as an integration branch for features. It's also convenient to tag all commits in the master branch with a version number.

The first step is to complement the default master with a develop branch. A simple way to do this is for one developer to create an empty develop branch locally and push it to the server:

git branch develop

git push -u origin develop

This branch will contain the complete history of the project, whereas master will contain an abridged version. Other developers should now clone the central repository and create a tracking branch for develop.

When using the git-flow extension library, executing git flow init on an existing repo will create the develop branch:

$ git flow init

Initialized empty Git repository in ~/project/.git/

No branches exist yet. Base branches must be created now.

Branch name for production releases: [master]

Branch name for "next release" development: [develop]

How to name your supporting branch prefixes?

Feature branches? [feature/]

Release branches? [release/]

Hotfix branches? [hotfix/]

Support branches? [support/]

Version tag prefix? []

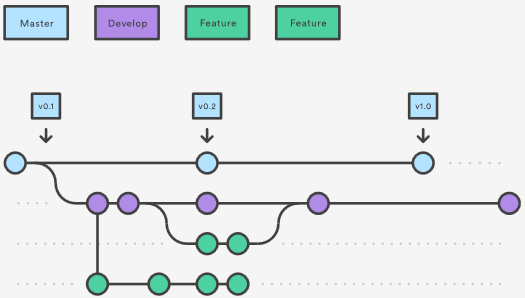
$ git branch

\* develop

 master

## Feature Branches

Each new feature should reside in its own branch, which can be [pushed to the central repository](https://www.atlassian.com/git/tutorials/syncing/git-push) for backup/collaboration. But, instead of branching off of master, feature branches use develop as their parent branch. When a feature is complete, it gets [merged back into develop](https://www.atlassian.com/git/tutorials/using-branches/git-merge). Features should never interact directly with master.



Note that feature branches combined with the develop branch is, for all intents and purposes, the Feature Branch Workflow. But, the Gitflow Workflow doesn’t stop there.

Feature branches are generally created off to the latest develop branch.

### Creating a feature branch

Without the git-flow extensions:

git checkout develop

git checkout -b feature\_branch

When using the git-flow extension:

git flow feature start feature\_branch

Continue your work and use Git like you normally would.

### Finishing a feature branch

When you’re done with the development work on the feature, the next step is to merge the feature\_branch into develop.

Without the git-flow extensions:

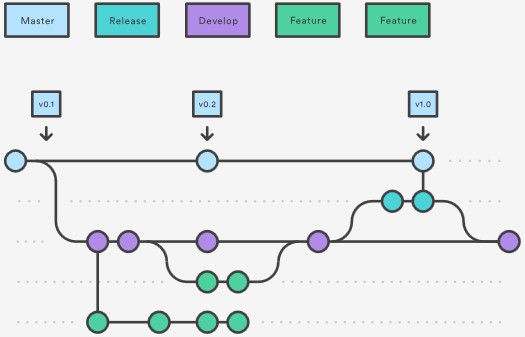
git checkout develop

git merge feature\_branch

Using the git-flow extensions:

git flow feature finish feature\_branch

## Release Branches



Once develop has acquired enough features for a release (or a predetermined release date is approaching), you fork a release branch off of develop. Creating this branch starts the next release cycle, so no new features can be added after this point—only bug fixes, documentation generation, and other release-oriented tasks should go in this branch. Once it's ready to ship, the release branch gets merged into master and tagged with a version number. In addition, it should be merged back into develop, which may have progressed since the release was initiated.

Using a dedicated branch to prepare releases makes it possible for one team to polish the current release while another team continues working on features for the next release. It also creates well-defined phases of development (e.g., it's easy to say, “This week we're preparing for version 4.0,” and to actually see it in the structure of the repository).

Making release branches is another straightforward branching operation. Like feature branches, release branches are based on the develop branch. A new release branch can be created using the following methods.

Without the git-flow extensions:

git checkout develop

git checkout -b release/0.1.0

When using the git-flow extensions:

$ git flow release start 0.1.0

Switched to a new branch 'release/0.1.0'

Once the release is ready to ship, it will get merged it into master and develop, then the release branch will be deleted. It’s important to merge back into develop because critical updates may have been added to the release branch and they need to be accessible to new features. If your organization stresses code review, this would be an ideal place for a pull request.

To finish a release branch, use the following methods:

Without the git-flow extensions:

git checkout develop

git merge release/0.1.0

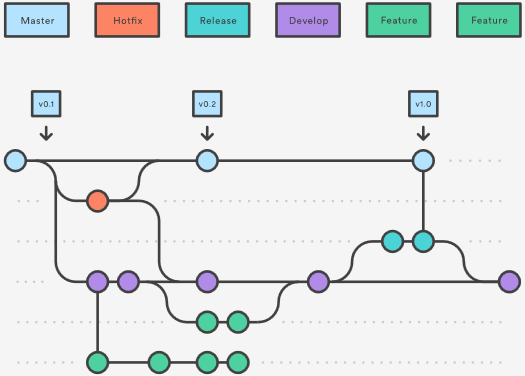
Or with the git-flow extension:

git checkout master

git checkout merge release/0.1.0

git flow release finish '0.1.0'

## Hotfix Branches



Maintenance or “hotfix” branches are used to quickly patch production releases. Hotfix branches are a lot like release branches and feature branches except they're based on master instead of develop. This is the only branch that should fork directly off of master. As soon as the fix is complete, it should be merged into both master and develop (or the current release branch), and master should be tagged with an updated version number.

Having a dedicated line of development for bug fixes lets your team address issues without interrupting the rest of the workflow or waiting for the next release cycle. You can think of maintenance branches as ad hoc release branches that work directly with master. A hotfix branch can be created using the following methods:

Without the git-flow extensions:

git checkout master

git checkout -b hotfix\_branch

When using the git-flow extensions:

$ git flow hotfix start hotfix\_branch

Similar to finishing a release branch, a hotfix branch gets merged into both master and develop.

git checkout master

git merge hotfix\_branch

git checkout develop

git merge hotfix\_branch

git branch -D hotfix\_branch

$ git flow hotfix finish hotfix\_branch

## Example

A complete example demonstrating a Feature Branch Flow is as follows. Assuming we have a repo setup with a master branch.

git checkout master  
git checkout -b develop  
git checkout -b feature\_branch  
# work happens on feature branch  
git checkout develop  
git merge feature\_branch  
git checkout master  
git merge develop  
git branch -d feature\_branch

In addition to the feature and release flow, a hotfix example is as follows:

git checkout master

git checkout -b hotfix\_branch

# work is done commits are added to the hotfix\_branch

git checkout develop

git merge hotfix\_branch

git checkout master

git merge hotfix\_branch

## Summary

Here we discussed the Gitflow Workflow. Gitflow is one of many styles of [Git workflows](https://www.atlassian.com/git/tutorials/comparing-workflows) you and your team can utilize.

Some key takeaways to know about Gitflow are:

* The workflow is great for a release-based software workflow.
* Gitflow offers a dedicated channel for hotfixes to production.

The overall flow of Gitflow is:

1. A develop branch is created from master
2. A release branch is created from develop
3. Feature branches are created from develop
4. When a feature is complete it is merged into the develop branch
5. When the release branch is done it is merged into develop and master
6. If an issue in master is detected a hotfix branch is created from master
7. Once the hotfix is complete it is merged to both develop and master

Next, learn about the [Forking Workflow](https://www.atlassian.com/git/tutorials/comparing-workflows/forking-workflow) or visit our [workflow comparison page](https://www.atlassian.com/git/tutorials/comparing-workflows).

# Forking Workflow

The Forking Workflow is fundamentally different than other popular Git workflows. Instead of using a single server-side repository to act as the “central” codebase, it gives every developer their own server-side repository. This means that each contributor has not one, but two Git repositories: a private local one and a public server-side one. The Forking Workflow is most often seen in public open source projects.

The main advantage of the Forking Workflow is that contributions can be integrated without the need for everybody to push to a single central repository. Developers push to their own server-side repositories, and only the project maintainer can push to the official repository. This allows the maintainer to accept commits from any developer without giving them write access to the official codebase.

The Forking Workflow typically follows a branching model based on the [Gitflow Workflow](https://www.atlassian.com/git/tutorials/comparing-workflows/gitflow-workflow). This means that complete feature branches will be purposed for merge into the original project maintainer's repository. The result is a distributed workflow that provides a flexible way for large, organic teams (including untrusted third-parties) to collaborate securely. This also makes it an ideal workflow for open source projects.

## How it works

As in the other [Git workflows](https://www.atlassian.com/git/tutorials/comparing-workflows), the Forking Workflow begins with an official public repository stored on a server. But when a new developer wants to start working on the project, they do not directly clone the official repository.

Instead, they fork the official repository to create a copy of it on the server. This new copy serves as their personal public repository—no other developers are allowed to push to it, but they can pull changes from it (we’ll see why this is important in a moment). After they have created their server-side copy, the developer performs a [git clone](https://www.atlassian.com/git/tutorials/setting-up-a-repository/git-clone) to get a copy of it onto their local machine. This serves as their private development environment, just like in the other workflows.

When they're ready to publish a local commit, they push the commit to their own public repository—not the official one. Then, they file a pull request with the main repository, which lets the project maintainer know that an update is ready to be integrated. The pull request also serves as a convenient discussion thread if there are issues with the contributed code. The following is a step-by-step example of this workflow.

1. A developer 'forks' an 'official' server-side repository. This creates their own server-side copy.
2. The new server-side copy is cloned to their local system.
3. A Git remote path for the 'official' repository is added to the local clone.
4. A new local feature branch is created.
5. The developer makes changes on the new branch.
6. New commits are created for the changes.
7. The branch gets pushed to the developer's own server-side copy.
8. The developer opens a pull request from the new branch to the 'official' repository.
9. The pull request gets approved for merge and is merged into the original server-side repository

To integrate the feature into the official codebase, the maintainer pulls the contributor’s changes into their local repository, checks to make sure it doesn’t break the project, merges it into their local master branch, then pushes the master branch to the official repository on the server. The contribution is now part of the project, and other developers should pull from the official repository to synchronize their local repositories.

It’s important to understand that the notion of an “official” repository in the Forking Workflow is merely a convention. In fact, the only thing that makes the official repository so official is that it’s the public repository of the project maintainer.

## Forking vs cloning

It's important to note that "forked" repositories and "forking" are not special operations. Forked repositories are created using the standard [git clone](https://www.atlassian.com/git/tutorials/setting-up-a-repository/git-clone) command. Forked repositories are generally "server-side clones" and usually managed and hosted by a 3rd party Git service like [Bitbucket](https://bitbucket.org/product). There is no unique Git command to create forked repositories. A clone operation is essentially a copy of a repository and its history.

## Branching in the Forking Workflow

All of these personal public repositories are really just a convenient way to share branches with other developers. Everybody should still be using branches to isolate individual features, just like in the [Feature Branch Workflow](https://www.atlassian.com/git/tutorials/comparing-workflows/feature-branch-workflow) and the [Gitflow Workflow.](https://www.atlassian.com/git/tutorials/comparing-workflows/gitflow-workflow) The only difference is how those branches get shared. In the Forking Workflow, they are pulled into another developer’s local repository, while in the Feature Branch and Gitflow Workflows they are pushed to the official repository.

## Fork a repository

All new developers to a Forking Workflow project need to fork the official repository. As previously stated, forking is just a standard git clone operation. It’s possible to do this by SSH’ing into the server and running git clone to copy it to another location on the server. Popular Git hosting services like Bitbucket, offer repo forking features that automate this step.

## Clone your fork

Next each developer needs to clone their own public forked repository. They can do this with the familiar git clone command.

Assuming the use of Bitbucket to host these repositories, developers on a project should have their own Bitbucket account and they should clone their forked copy of the repository with:

git clone https://user@bitbucket.org/user/repo.git

## Adding a remote

Whereas other Git workflows use a single origin remote that points to the central repository, the Forking Workflow requires two remotes—one for the official repository, and one for the developer’s personal server-side repository. While you can call these remotes anything you want, a common convention is to use origin as the remote for your forked repository (this will be created automatically when you run git clone) and upstream for the official repository.

git remote add upstream https://bitbucket.org/maintainer/repo

You’ll need to create the upstream remote yourself using the above command. This will let you easily keep your local repository up-to-date as the official project progresses. Note that if your upstream repository has authentication enabled (i.e., it's not open source), you'll need to supply a username, like so:

git remote add upstream https://user@bitbucket.org/maintainer/repo.git

This requires users to supply a valid password before cloning or pulling from the official codebase.

## Working in a branch: making & pushing changes

In the developer's local copy of the forked repository they can edit code, commit changes, and create branches just like in other Git workflows:

git checkout -b some-feature

# Edit some code

git commit -a -m "Add first draft of some feature"

All of their changes will be entirely private until they push it to their public repository. And, if the official project has moved forward, they can access new commits with git pull:

git pull upstream master

Since developers should be working in a dedicated feature branch, this should generally result in a fast-forward merge.

## Making a Pull Request

Once a developer is ready to share their new feature, they need to do two things. First, they have to make their contribution accessible to other developers by pushing it to their public repository. Their origin remote should already be set up, so all they should have to do is the following:

git push origin feature-branch

This diverges from the other workflows in that the origin remote points to the developer’s personal server-side repository, not the main codebase.

Second, they need to notify the project maintainer that they want to merge their feature into the official codebase. Bitbucket provides a “pull request” button that leads to a form asking you to specify which branch you want to merge into the official repository. Typically, you’ll want to integrate your feature branch into the upstream remote’s master branch.

## Summary

To recap, the Forking Workflow is commonly used in public open-source projects. Forking is a git clone operation executed on a server copy of a projects repo. A Forking Workflow is often used in conjunction with a Git hosting service like Bitbucket. A high-level example of a Forking Workflow is:

1. You want to contribute to an open source library hosted at bitbucket.org/userA/open-project
2. Using Bitbucket you create a fork of the repo to bitbucket.org/YourName/open-project
3. On your local system you execute git clone on https://bitbucket.org/YourName/open-project to get a local copy of the repo
4. You create a new feature branch in your local repo
5. Work is done to complete the new feature and git commit is executed to save the changes
6. You then push the new feature branch to your remote forked repo
7. Using Bitbucket you open up a pull request for the new branch against the original repo at bitbucket.org/userA/open-project

The Forking Workflow helps a maintainer of a project open up the repository to contributions from any developer without having to manually manage authorization settings for each individual contributor. This gives the maintainer more of a "pull" style workflow. Most commonly used in open-source projects, the Forking Workflow can also be applied to private business workflows to give more authoritative control over what is merged into a release. This can be useful in teams that have Deploy Managers or strict release cycles.

Unsure what workflow is right for you? Check out our comprehensive [Git workflow comparison page.](https://www.atlassian.com/git/tutorials/comparing-workflows)

# Advanced Git Tutorials

[Atlassian’s Git tutorials](https://www.atlassian.com/git/tutorials) introduce the most common Git commands, and our [Git Workflows](https://www.atlassian.com/git/tutorials/comparing-workflows) modules discuss how these commands are typically used to facilitate collaboration. Alone, these are enough to get a development team up and running with Git. But, if you really want to leverage the full power of Git, you’re ready to dive into our Advanced Git articles.

Each of these articles provide an in-depth discussion of an advanced feature of Git. Instead of presenting new commands and concepts, they refine your existing Git skills by explaining what’s going on under the hood. Armed with this knowledge, you’ll be able to use familiar Git commands more effectively. More importantly, you’ll never be scared of breaking your Git repository because you’ll understand why it broke and how to fix it.

## Merging vs. Rebasing

Git is all about working with divergent history. Its git merge and git rebase commands offer alternative ways to integrate commits from different branches, and both options come with their own advantages. In this article, we’ll discuss how and when a basic git merge operation can be replaced with a rebase.

## Resetting, Checking Out, and Reverting

The git reset, git checkout, and git revert commands are all similar in that they undo some type of change in your repository. But, they all affect different combinations of the working directory, staged snapshot, and commit history. This article clearly defines how these commands differ and when each of them should be used in the standard Git workflows.

## Advanced Git Log

The git log command is what makes your project history useful. Without it, you wouldn’t be able to access any of your commits. But, if you’re like most aspiring Git users, you’ve probably only scratched the surface of what’s possible with git log. This article walks you through its advanced formatting and filtering options, giving you the power to extract all sorts of interesting information from your Git repository.

## Git Hooks

If you want to perform custom actions when a certain event takes place in a Git repository, hooks are your tool of choice. They let you normalize commit messages, automate testing suites, notify continuous integration systems, and much more. After this article, you’ll understand the many ways in which Git hooks can streamline your workflow.

## Refs and the Reflog

A **ref** is Git’s internal way of referring to a commit. You’re already familiar with many categories of refs, including commit hashes and branch names. But, there are many other types of refs, and virtually every Git command utilizes them in some form or another. You’ll walk away from this article with an intimate knowledge of Git’s inner workings.

# Merging vs. Rebasing

[Conceptual Overview](https://www.atlassian.com/git/tutorials/merging-vs-rebasing#conceptual-overview) [The Golden Rule of Rebasing](https://www.atlassian.com/git/tutorials/merging-vs-rebasing#the-golden-rule-of-rebasing) [Workflow Walkthrough](https://www.atlassian.com/git/tutorials/merging-vs-rebasing#workflow-walkthrough) [Summary](https://www.atlassian.com/git/tutorials/merging-vs-rebasing#summary)

The git rebase command has a reputation for being magical Git voodoo that beginners should stay away from, but it can actually make life much easier for a development team when used with care. In this article, we’ll compare git rebase with the related git merge command and identify all of the potential opportunities to incorporate rebasing into the typical Git workflow.

## Conceptual Overview

The first thing to understand about git rebase is that it solves the same problem as git merge. Both of these commands are designed to integrate changes from one branch into another branch—they just do it in very different ways.

Consider what happens when you start working on a new feature in a dedicated branch, then another team member updates the master branch with new commits. This results in a forked history, which should be familiar to anyone who has used Git as a collaboration tool.

Now, let’s say that the new commits in master are relevant to the feature that you’re working on. To incorporate the new commits into your feature branch, you have two options: merging or rebasing.

### The Merge Option

The easiest option is to merge the master branch into the feature branch using something like the following:

git checkout feature

git merge master

Or, you can condense this to a one-liner:

git merge feature master

This creates a new “merge commit” in the feature branch that ties together the histories of both branches, giving you a branch structure that looks like this:

Merging is nice because it’s a non-destructive operation. The existing branches are not changed in any way. This avoids all of the potential pitfalls of rebasing (discussed below).

On the other hand, this also means that the feature branch will have an extraneous merge commit every time you need to incorporate upstream changes. If master is very active, this can pollute your feature branch’s history quite a bit. While it’s possible to mitigate this issue with advanced git log options, it can make it hard for other developers to understand the history of the project.

### The Rebase Option

As an alternative to merging, you can rebase the feature branch onto master branch using the following commands:

git checkout feature

git rebase master

This moves the entire feature branch to begin on the tip of the master branch, effectively incorporating all of the new commits in master. But, instead of using a merge commit, rebasing re-writes the project history by creating brand new commits for each commit in the original branch.

The major benefit of rebasing is that you get a much cleaner project history. First, it eliminates the unnecessary merge commits required by git merge. Second, as you can see in the above diagram, rebasing also results in a perfectly linear project history—you can follow the tip of feature all the way to the beginning of the project without any forks. This makes it easier to navigate your project with commands like git log, git bisect, and gitk.

But, there are two trade-offs for this pristine commit history: safety and traceability. If you don’t follow the [Golden Rule of Rebasing](https://www.atlassian.com/git/tutorials/merging-vs-rebasing#the-golden-rule-of-rebasing), re-writing project history can be potentially catastrophic for your collaboration workflow. And, less importantly, rebasing loses the context provided by a merge commit—you can’t see when upstream changes were incorporated into the feature.

### Interactive Rebasing

Interactive rebasing gives you the opportunity to alter commits as they are moved to the new branch. This is even more powerful than an automated rebase, since it offers complete control over the branch’s commit history. Typically, this is used to clean up a messy history before merging a feature branch into master.

To begin an interactive rebasing session, pass the i option to the git rebase command:

git checkout feature

git rebase -i master

This will open a text editor listing all of the commits that are about to be moved:

pick 33d5b7a Message for commit #1

pick 9480b3d Message for commit #2

pick 5c67e61 Message for commit #3

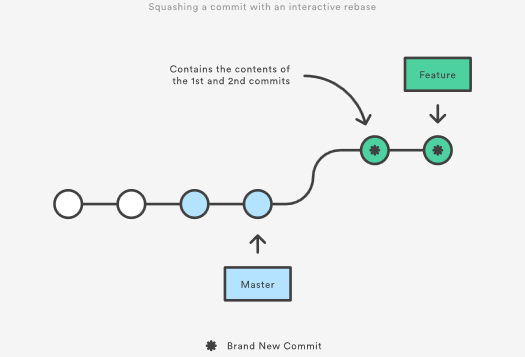
This listing defines exactly what the branch will look like after the rebase is performed. By changing the pick command and/or re-ordering the entries, you can make the branch’s history look like whatever you want. For example, if the 2nd commit fixes a small problem in the 1st commit, you can condense them into a single commit with the fixup command:

pick 33d5b7a Message for commit #1

fixup 9480b3d Message for commit #2

pick 5c67e61 Message for commit #3

When you save and close the file, Git will perform the rebase according to your instructions, resulting in project history that looks like the following:



Eliminating insignificant commits like this makes your feature’s history much easier to understand. This is something that git merge simply cannot do.

## The Golden Rule of Rebasing

Once you understand what rebasing is, the most important thing to learn is when not to do it. The golden rule of git rebase is to never use it on public branches.

For example, think about what would happen if you rebased master onto your feature branch:

The rebase moves all of the commits in master onto the tip of feature. The problem is that this only happened in your repository. All of the other developers are still working with the original master. Since rebasing results in brand new commits, Git will think that your master branch’s history has diverged from everybody else’s.

The only way to synchronize the two master branches is to merge them back together, resulting in an extra merge commit and two sets of commits that contain the same changes (the original ones, and the ones from your rebased branch). Needless to say, this is a very confusing situation.

So, before you run git rebase, always ask yourself, “Is anyone else looking at this branch?” If the answer is yes, take your hands off the keyboard and start thinking about a non-destructive way to make your changes (e.g., the git revert command). Otherwise, you’re safe to re-write history as much as you like.

### Force-Pushing

If you try to push the rebased master branch back to a remote repository, Git will prevent you from doing so because it conflicts with the remote master branch. But, you can force the push to go through by passing the --force flag, like so:

# Be very careful with this command!

git push --force

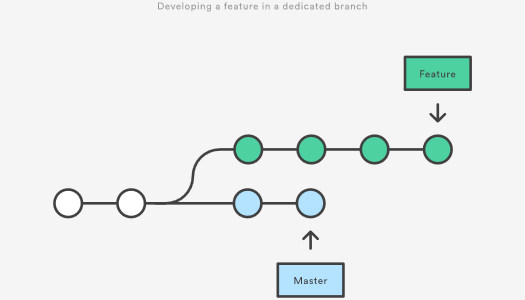
This overwrites the remote master branch to match the rebased one from your repository and makes things very confusing for the rest of your team. So, be very careful to use this command only when you know exactly what you’re doing.

One of the only times you should be force-pushing is when you’ve performed a local cleanup after you’ve pushed a private feature branch to a remote repository (e.g., for backup purposes). This is like saying, “Oops, I didn’t really want to push that original version of the feature branch. Take the current one instead.” Again, it’s important that nobody is working off of the commits from the original version of the feature branch.

## Workflow Walkthrough

Rebasing can be incorporated into your existing Git workflow as much or as little as your team is comfortable with. In this section, we’ll take a look at the benefits that rebasing can offer at the various stages of a feature’s development.

The first step in any workflow that leverages git rebase is to create a dedicated branch for each feature. This gives you the necessary branch structure to safely utilize rebasing:



### Local Cleanup

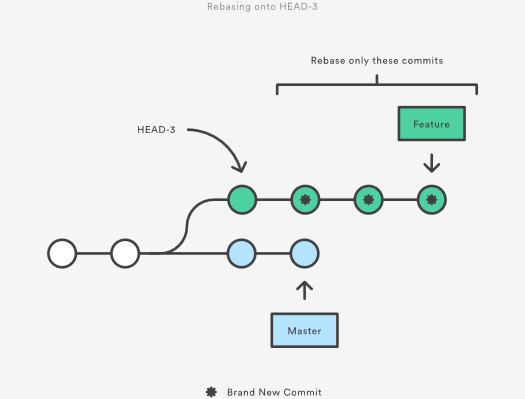
One of the best ways to incorporate rebasing into your workflow is to clean up local, in-progress features. By periodically performing an interactive rebase, you can make sure each commit in your feature is focused and meaningful. This lets you write your code without worrying about breaking it up into isolated commits—you can fix it up after the fact.

When calling git rebase, you have two options for the new base: The feature’s parent branch (e.g., master), or an earlier commit in your feature. We saw an example of the first option in the Interactive Rebasing section. The latter option is nice when you only need to fix up the last few commits. For example, the following command begins an interactive rebase of only the last 3 commits.

git checkout feature

git rebase -i HEAD~3

By specifying HEAD~3 as the new base, you’re not actually moving the branch—you’re just interactively re-writing the 3 commits that follow it. Note that this will not incorporate upstream changes into the feature branch.



If you want to re-write the entire feature using this method, the git merge-base command can be useful to find the original base of the feature branch. The following returns the commit ID of the original base, which you can then pass to git rebase:

git merge-base feature master

This use of interactive rebasing is a great way to introduce git rebase into your workflow, as it only affects local branches. The only thing other developers will see is your finished product, which should be a clean, easy-to-follow feature branch history.

But again, this only works for private feature branches. If you’re collaborating with other developers via the same feature branch, that branch is public, and you’re not allowed to re-write its history.

There is no git merge alternative for cleaning up local commits with an interactive rebase.

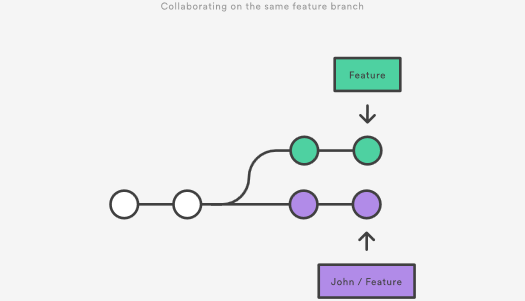
### Incorporating Upstream Changes Into a Feature

In the Conceptual Overview section, we saw how a feature branch can incorporate upstream changes from master using either git merge or git rebase. Merging is a safe option that preserves the entire history of your repository, while rebasing creates a linear history by moving your feature branch onto the tip of master.

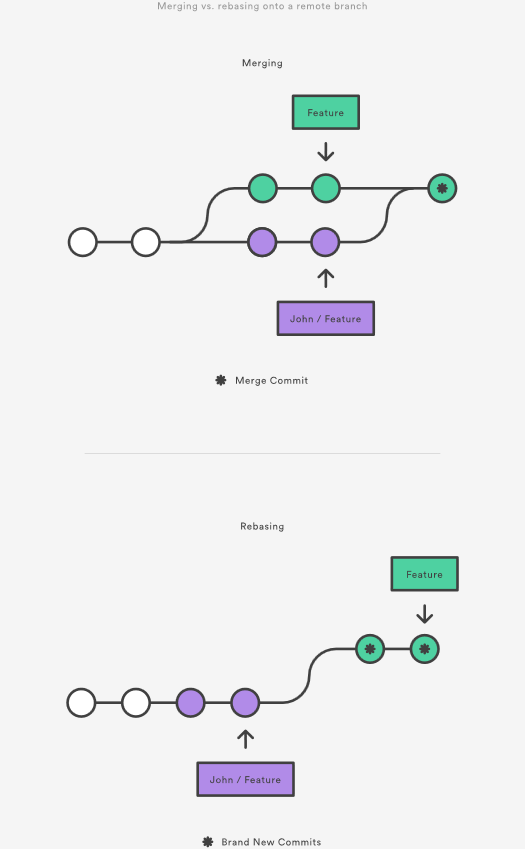
This use of git rebase is similar to a local cleanup (and can be performed simultaneously), but in the process it incorporates those upstream commits from master.

Keep in mind that it’s perfectly legal to rebase onto a remote branch instead of master. This can happen when collaborating on the same feature with another developer and you need to incorporate their changes into your repository.

For example, if you and another developer named John added commits to the feature branch, your repository might look like the following after fetching the remote feature branch from John’s repository:



You can resolve this fork the exact same way as you integrate upstream changes from master: either merge your local feature with john/feature, or rebase your local feature onto the tip of john/feature.



Note that this rebase doesn’t violate the Golden Rule of Rebasing because only your local feature commits are being moved—everything before that is untouched. This is like saying, “add my changes to what John has already done.” In most circumstances, this is more intuitive than synchronizing with the remote branch via a merge commit.

By default, the git pull command performs a merge, but you can force it to integrate the remote branch with a rebase by passing it the --rebase option.

### Reviewing a Feature With a Pull Request

If you use pull requests as part of your code review process, you need to avoid using git rebase after creating the pull request. As soon as you make the pull request, other developers will be looking at your commits, which means that it’s a public branch. Re-writing its history will make it impossible for Git and your teammates to track any follow-up commits added to the feature.

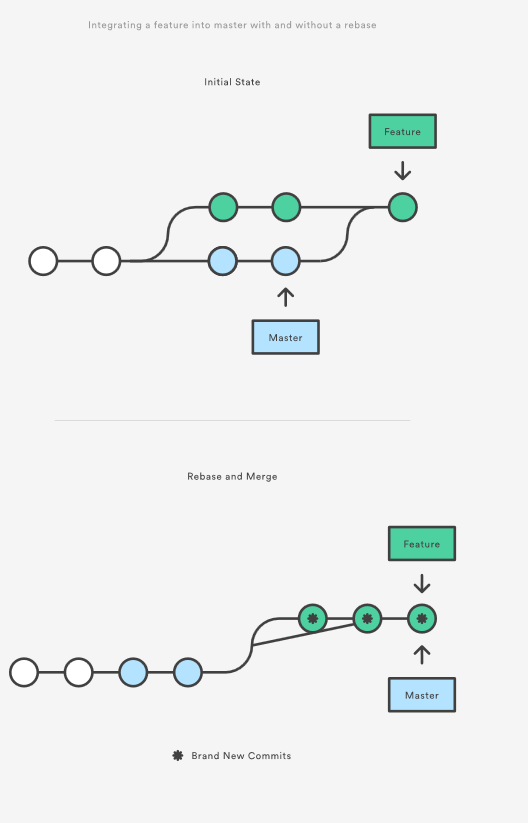
Any changes from other developers need to be incorporated with git merge instead of git rebase.

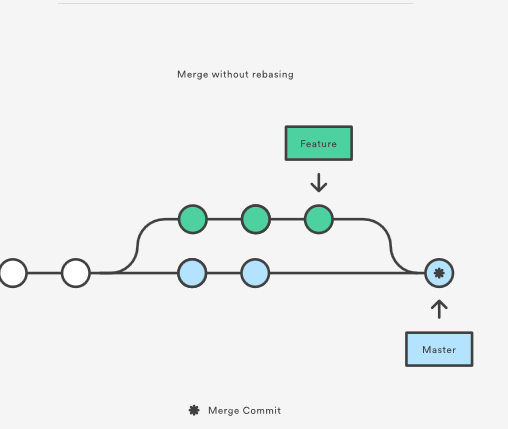
For this reason, it’s usually a good idea to clean up your code with an interactive rebase before submitting your pull request.

### Integrating an Approved Feature

After a feature has been approved by your team, you have the option of rebasing the feature onto the tip of the master branch before using git merge to integrate the feature into the main code base.

This is a similar situation to incorporating upstream changes into a feature branch, but since you’re not allowed to re-write commits in the master branch, you have to eventually use git merge to integrate the feature. However, by performing a rebase before the merge, you’re assured that the merge will be fast-forwarded, resulting in a perfectly linear history. This also gives you the chance to squash any follow-up commits added during a pull request.





If you’re not entirely comfortable with git rebase, you can always perform the rebase in a temporary branch. That way, if you accidentally mess up your feature’s history, you can check out the original branch and try again. For example:

git checkout feature

git checkout -b temporary-branch

git rebase -i master

# [Clean up the history]

git checkout master

git merge temporary-branch

## Summary

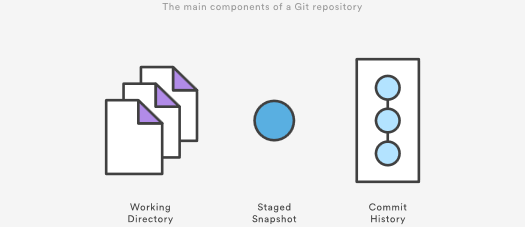
And that’s all you really need to know to start rebasing your branches. If you would prefer a clean, linear history free of unnecessary merge commits, you should reach for git rebase instead of git merge when integrating changes from another branch.

On the other hand, if you want to preserve the complete history of your project and avoid the risk of re-writing public commits, you can stick with git merge. Either option is perfectly valid, but at least now you have the option of leveraging the benefits of git rebase.

# Resetting, Checking Out & Reverting

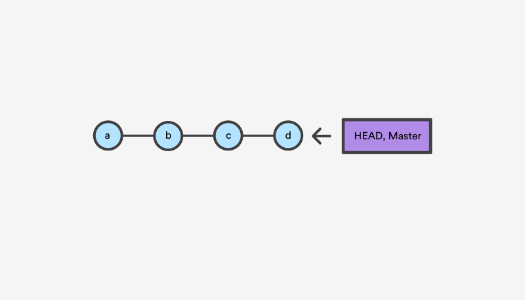
The [git reset](https://www.atlassian.com/git/tutorials/undoing-changes/git-reset), [git checkout](https://www.atlassian.com/git/tutorials/using-branches/git-checkout), and [git revert](https://www.atlassian.com/git/tutorials/undoing-changes/git-revert) commands are some of the most useful tools in your Git toolbox. They all let you undo some kind of change in your repository, and the first two commands can be used to manipulate either commits or individual files.

Because they’re so similar, it’s very easy to mix up which command should be used in any given development scenario. In this article, we’ll compare the most common configurations of git reset, git checkout, and git revert. Hopefully, you’ll walk away with the confidence to navigate your repository using any of these commands.

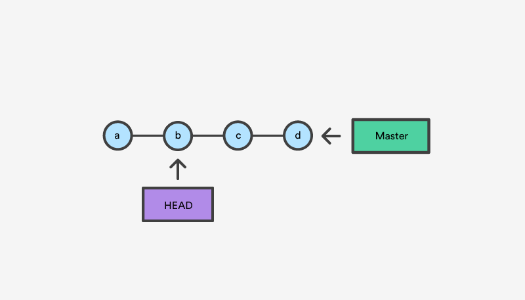


It helps to think about each command in terms of their effect on the three state management mechanisms of a Git repository: the working directory, the staged snapshot, and the commit history. These components are sometimes know as "The three trees" of Git. We explore the three trees in depth on the [git reset](https://www.atlassian.com/git/tutorials/undoing-changes/git-reset) page. Keep these mechanisms in mind as you read through this article.

A checkout is an operation that moves the HEAD ref pointer to a specified commit. To demonstrate this consider the following example.



This example demonstrates a sequence of commits on the master branch. The HEAD ref and master branch ref currently point to commit d. Now let us execute git checkout b



This is an update to the "Commit History" tree. The git checkout command can be used in a commit, or file level scope. A file level checkout will change the file's contents to those of the specific commit.

A revert is an operation that takes a specified commit and creates a new commit which inverses the specified commit. git revert can only be run at a commit level scope and has no file level functionality.

A reset is an operation that takes a specified commit and resets the "three trees" to match the state of the repository at that specified commit. A reset can be invoked in three different modes which correspond to the three trees.

Checkout and reset are generally used for making local or private 'undos'. They modify the history of a repository that can cause conflicts when pushing to remote shared repositories. Revert is considered a safe operation for 'public undos' as it creates new history which can be shared remotely and doesn't overwrite history remote team members may be dependent on.

## Git Reset vs Revert vs Checkout reference

The table below sums up the most common use cases for all of these commands. Be sure to keep this reference handy, as you’ll undoubtedly need to use at least some of them during your Git career.

| **Command** | **Scope** | **Common use cases** |
| --- | --- | --- |
| git reset | Commit-level | Discard commits in a private branch or throw away uncommited changes |
| git reset | File-level | Unstage a file |
| git checkout | Commit-level | Switch between branches or inspect old snapshots |
| git checkout | File-level | Discard changes in the working directory |
| git revert | Commit-level | Undo commits in a public branch |
| git revert | File-level | (N/A) |

## Commit Level Operations

The parameters that you pass to git reset and git checkout determine their scope. When you don’t include a file path as a parameter, they operate on whole commits. That’s what we’ll be exploring in this section. Note that git revert has no file-level counterpart.

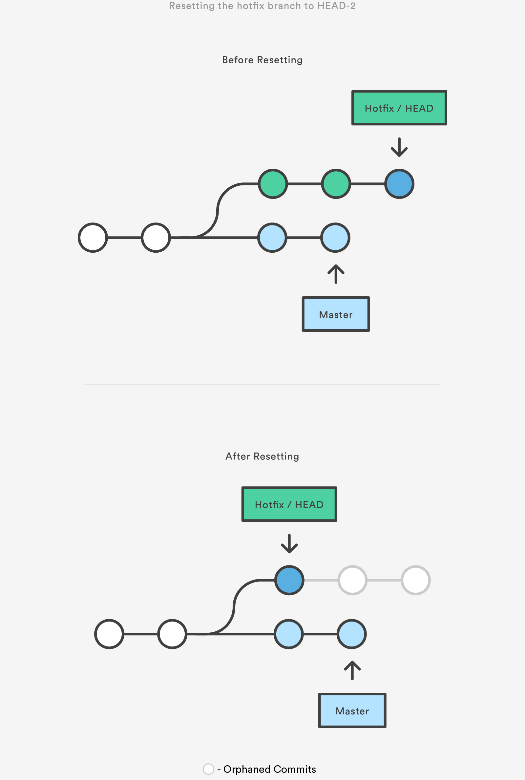
### Reset A Specific Commit

On the commit-level, resetting is a way to move the tip of a branch to a different commit. This can be used to remove commits from the current branch. For example, the following command moves the hotfix branch backwards by two commits.

git checkout hotfix

git reset HEAD~2

The two commits that were on the end of hotfix are now dangling, or orphaned commits. This means they will be deleted the next time Git performs a garbage collection. In other words, you’re saying that you want to throw away these commits. This can be visualized as the following:



This usage of git reset is a simple way to undo changes that haven’t been shared with anyone else. It’s your go-to command when you’ve started working on a feature and find yourself thinking, “Oh crap, what am I doing? I should just start over.”

In addition to moving the current branch, you can also get git reset to alter the staged snapshot and/or the working directory by passing it one of the following flags:

* --soft – The staged snapshot and working directory are not altered in any way.
* --mixed – The staged snapshot is updated to match the specified commit, but the working directory is not affected. This is the default option.
* --hard – The staged snapshot and the working directory are both updated to match the specified commit.

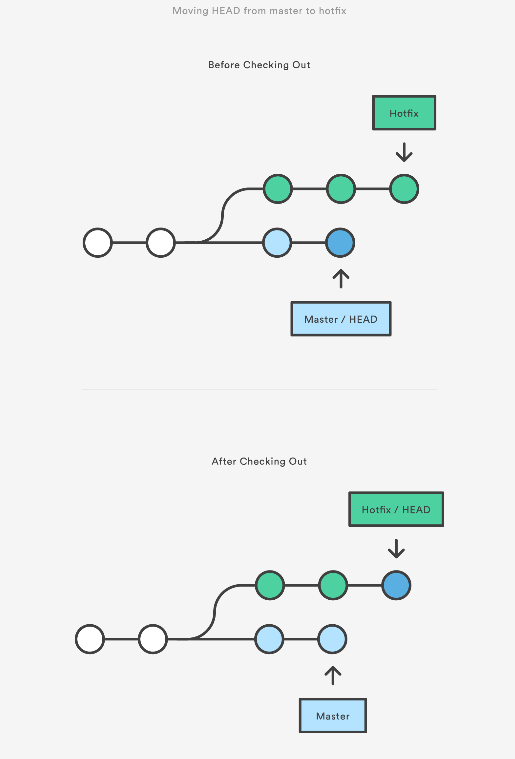
It’s easier to think of these modes as defining the scope of a git reset operation. For further detailed information visit the [git reset](https://www.atlassian.com/git/tutorials/undoing-changes/git-reset) page.

### Checkout old commits

The git checkout command is used to update the state of the repository to a specific point in the projects history. When passed with a branch name, it lets you switch between branches.

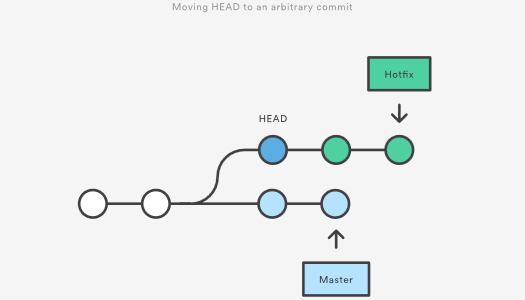
git checkout hotfix

Internally, all the above command does is move HEAD to a different branch and update the working directory to match. Since this has the potential to overwrite local changes, Git forces you to commit or [stash](https://www.atlassian.com/git/tutorials/saving-changes/git-stash) any changes in the working directory that will be lost during the checkout operation. Unlike git reset, git checkout doesn’t move any branches around.



You can also check out arbitrary commits by passing the commit reference instead of a branch. This does the exact same thing as checking out a branch: it moves the HEAD reference to the specified commit. For example, the following command will check out the grandparent of the current commit:

git checkout HEAD~2



his is useful for quickly inspecting an old version of your project. However, since there is no branch reference to the current HEAD, this puts you in a detached HEAD state. This can be dangerous if you start adding new commits because there will be no way to get back to them after you switch to another branch. For this reason, you should always create a new branch before adding commits to a detached HEAD.

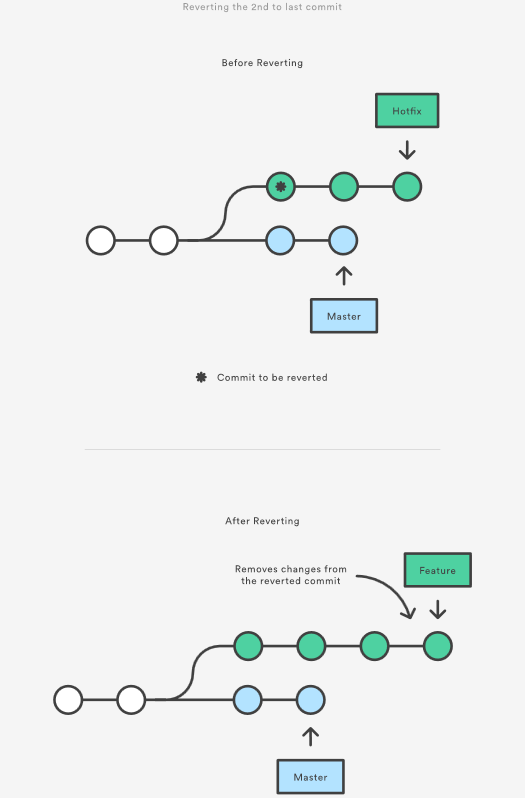
### Undo Public Commits with Revert

Reverting undoes a commit by creating a new commit. This is a safe way to undo changes, as it has no chance of re-writing the commit history. For example, the following command will figure out the changes contained in the 2nd to last commit, create a new commit undoing those changes, and tack the new commit onto the existing project.

git checkout hotfix

git revert HEAD~2

This can be visualized as the following:



Contrast this with git reset, which does alter the existing commit history. For this reason, git revert should be used to undo changes on a public branch, and git reset should be reserved for undoing changes on a private branch.

You can also think of git revert as a tool for undoing committed changes, while git reset HEAD is for undoing uncommitted changes.

Like git checkout, git revert has the potential to overwrite files in the working directory, so it will ask you to commit or [stash changes](https://www.atlassian.com/git/tutorials/saving-changes/git-stash) that would be lost during the revert operation.

## File-level Operations

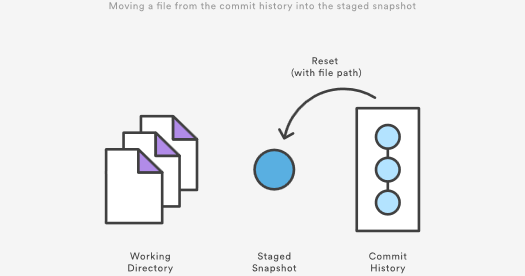
The git reset and git checkout commands also accept an optional file path as a parameter. This dramatically alters their behavior. Instead of operating on entire snapshots, this forces them to limit their operations to a single file.

### Git Reset A Specific File

When invoked with a file path, git reset updates the staged snapshot to match the version from the specified commit. For example, this command will fetch the version of foo.py in the 2nd-to-last commit and stage it for the next commit:

git reset HEAD~2 foo.py

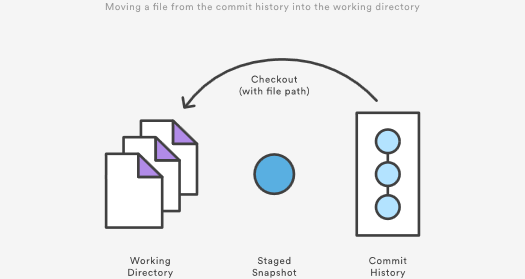
As with the commit-level version of git reset, this is more commonly used with HEAD rather than an arbitrary commit. Running git reset HEAD foo.py will unstage foo.py. The changes it contains will still be present in the working directory.



The --soft, --mixed, and --hard flags do not have any effect on the file-level version of git reset, as the staged snapshot is always updated, and the working directory is never updated.

### Git Checkout File

Checking out a file is similar to using git reset with a file path, except it updates the working directory instead of the stage. Unlike the commit-level version of this command, this does not move the HEAD reference, which means that you won’t switch branches.



For example, the following command makes foo.py in the working directory match the one from the 2nd-to-last commit:

git checkout HEAD~2 foo.py

Just like the commit-level invocation of git checkout, this can be used to inspect old versions of a project—but the scope is limited to the specified file.

If you stage and commit the checked-out file, this has the effect of “reverting” to the old version of that file. Note that this removes all of the subsequent changes to the file, whereas the git revert command undoes only the changes introduced by the specified commit.

Like git reset, this is commonly used with HEAD as the commit reference. For instance, git checkout HEAD foo.py has the effect of discarding unstaged changes to foo.py. This is similar behavior to git reset HEAD --hard, but it operates only on the specified file.

## Summary

You should now have all the tools you could ever need to undo changes in a Git repository. The [git reset](https://www.atlassian.com/git/tutorials/undoing-changes/git-reset), [git checkout](https://www.atlassian.com/git/tutorials/using-branches/git-checkout), and [git revert](https://www.atlassian.com/git/tutorials/undoing-changes/git-revert) commands can be confusing, but when you think about their effects on the working directory, staged snapshot, and commit history, it should be easier to discern which command fits the development task at hand.

# Advanced Git log

[Formatting Log Output](https://www.atlassian.com/git/tutorials/git-log#formatting-log-output) [Filtering the Commit History](https://www.atlassian.com/git/tutorials/git-log#filtering-the-commit-history) [Summary](https://www.atlassian.com/git/tutorials/git-log#summary)

The purpose of any version control system is to record changes to your code. This gives you the power to go back into your project history to see who contributed what, figure out where bugs were introduced, and revert problematic changes. But, having all of this history available is useless if you don’t know how to navigate it. That’s where the git log command comes in.

By now, you should already know the basic git log command for displaying commits. But, you can alter this output by passing many different parameters to git log.

The advanced features of git log can be split into two categories: formatting how each commit is displayed, and filtering which commits are included in the output. Together, these two skills give you the power to go back into your project and find any information that you could possibly need.

## Formatting Log Output

First, this article will take a look at the many ways in which git log’s output can be formatted. Most of these come in the form of flags that let you request more or less information from git log.

If you don’t like the default git log format, you can use git config’s aliasing functionality to create a shortcut for any of the formatting options discussed below. Please see in [The git config Command](https://www.atlassian.com/git/tutorials/setting-up-a-repository#git-config) for how to set up an alias.

### Oneline

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

0e25143 Merge branch 'feature'

ad8621a Fix a bug in the feature

16b36c6 Add a new feature

23ad9ad Add the initial code base

This is very useful for getting a high-level overview of your project.

### Decorating

Many times it’s useful to know which branch or tag each commit is associated with. The --decorate flag makes git log display all of the references (e.g., branches, tags, etc) that point to each commit.

This can be combined with other configuration options. For example, running git log --oneline --decorate will format the commit history like so:

0e25143 (HEAD, master) Merge branch 'feature'

ad8621a (feature) Fix a bug in the feature

16b36c6 Add a new feature

23ad9ad (tag: v0.9) Add the initial code base

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

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

### Diffs

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

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

commit f2a238924e89ca1d4947662928218a06d39068c3

Author: John <john@example.com>

Date: Fri Jun 25 17:30:28 2014 -0500

Add a new feature

hello.py | 105 ++++++++++++++++++++++++-----------------

1 file changed, 67 insertion(+), 38 deletions(-)

The amount of + and - signs next to the file name show the relative number of changes to each file altered by the commit. This gives you an idea of where the changes for each commit can be found.

If you want to see the actual changes introduced by each commit, you can pass the -p option to git log. This outputs the entire patch representing that commit:

commit 16b36c697eb2d24302f89aa22d9170dfe609855b

Author: Mary <mary@example.com>

Date: Fri Jun 25 17:31:57 2014 -0500

Fix a bug in the feature

diff --git a/hello.py b/hello.py

index 18ca709..c673b40 100644

--- a/hello.py

+++ b/hello.py

@@ -13,14 +13,14 @@ B

-print("Hello, World!")

+print("Hello, Git!")

For commits with a lot of changes, the resulting output can become quite long and unwieldy. More often than not, if you’re displaying a full patch, you’re probably searching for a specific change. For this, you want to use the pickaxe option.

### The Shortlog

The git shortlog command is a special version of git log intended for creating release announcements. It groups each commit by author and displays the first line of each commit message. This is an easy way to see who’s been working on what.

For example, if two developers have contributed 5 commits to a project, the git shortlog output might look like the following:

Mary (2):

Fix a bug in the feature

Fix a serious security hole in our framework

John (3):

Add the initial code base

Add a new feature

Merge branch 'feature'

By default, git shortlog sorts the output by author name, but you can also pass the -n option to sort by the number of commits per author.

### Graphs

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

git log --graph --oneline --decorate

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

\* 0e25143 (HEAD, master) Merge branch 'feature'

|\

| \* 16b36c6 Fix a bug in the new feature

| \* 23ad9ad Start a new feature

\* | ad8621a Fix a critical security issue

|/

\* 400e4b7 Fix typos in the documentation

\* 160e224 Add the initial code base

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

While this is a nice option for simple repositories, you’re probably better off with a more full-featured visualization tool like gitk or [Sourcetree](https://www.atlassian.com/software/sourcetree/overview) for projects that are heavily branched.

### Custom Formatting

For all of your other git log formatting needs, you can use the --pretty=format:"<string>" option. This lets you display each commit however you want using printf-style placeholders.

For example, the %cn, %h and %cd characters in the following command are replaced with the committer name, abbreviated commit hash, and the committer date, respectively.

git log --pretty=format:"%cn committed %h on %cd"

This results in the following format for each commit:

John committed 400e4b7 on Fri Jun 24 12:30:04 2014 -0500

John committed 89ab2cf on Thu Jun 23 17:09:42 2014 -0500

Mary committed 180e223 on Wed Jun 22 17:21:19 2014 -0500

John committed f12ca28 on Wed Jun 22 13:50:31 2014 -0500

The complete list of placeholders can be found in the [Pretty Formats](https://www.kernel.org/pub/software/scm/git/docs/git-log.html#_pretty_formats) section of the git log manual page.

Aside from letting you view only the information that you’re interested in, the --pretty=format:"<string>" option is particularly useful when you’re trying to pipe git log output into another command.

## Filtering the Commit History

Formatting how each commit gets displayed is only half the battle of learning git log. The other half is understanding how to navigate the commit history. The rest of this article introduces some of the advanced ways to pick out specific commits in your project history using git log. All of these can be combined with any of the formatting options discussed above.

### By Amount

The most basic filtering option for git log is to limit the number of commits that are displayed. When you’re only interested in the last few commits, this saves you the trouble of viewing all the commits in a page.

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

git log -3

### By Date

If you’re looking for a commit from a specific time frame, you can use the --after or --before flags for filtering commits by date. These both accept a variety of date formats as a parameter. For example, the following command only shows commits that were created after July 1st, 2014 (inclusive):

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

You can also pass in relative references like "1 week ago" and "yesterday":

git log --after="yesterday"

To search for a commits that were created between two dates, you can provide both a --before and --after date. For instance, to display all the commits added between July 1st, 2014 and July 4th, 2014, you would use the following:

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

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

### By Author

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

git log --author="John"

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

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

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

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

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

### By Message

To filter commits by their commit message, use the --grep flag. This works just like the --author flag discussed above, but it matches against the commit message instead of the author.

For example, if your team includes relevant issue numbers in each commit message, you can use something like the following to pull out all of the commits related to that issue:

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

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

### By File

Many times, you’re only interested in changes that happened to a particular file. To show the history related to a file, all you have to do is pass in the file path. For example, the following returns all commits that affected either the foo.py or the bar.py file:

git log -- foo.py bar.py

The -- parameter is used to tell git log that subsequent arguments are file paths and not branch names. If there’s no chance of mixing it up with a branch, you can omit the --.

### By Content

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

git log -S"Hello, World!"

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

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

### By Range

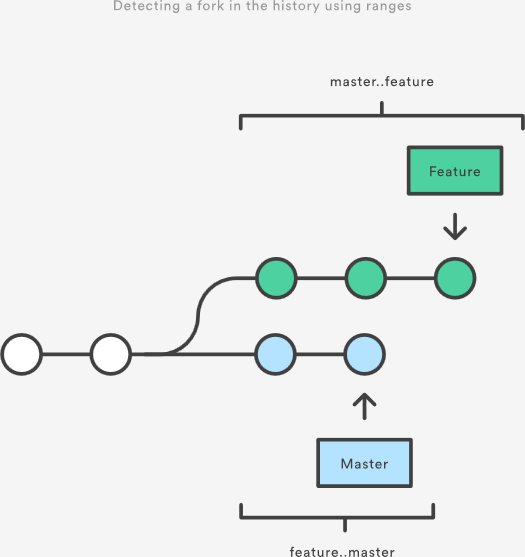
You can pass a range of commits to git log to show only the commits contained in that range. The range is specified in the following format, where <since> and <until> are commit references:

git log <since>..<until>

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

git log master..feature

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



Note that if you switch the order of the range (feature..master), you will get all of the commits in master, but not in feature. If git log outputs commits for both versions, this tells you that your history has diverged.

### Filtering Merge Commits

By default, git log includes merge commits in its output. But, if your team has an always-merge policy (that is, you merge upstream changes into topic branches instead of rebasing the topic branch onto the upstream branch), you’ll have a lot of extraneous merge commits in your project history.

You can prevent git log from displaying these merge commits by passing the --no-merges flag:

git log --no-merges

On the other hand, if you’re only interested in the merge commits, you can use the --merges flag:

git log --merges

This returns all commits that have at least two parents.

## Summary

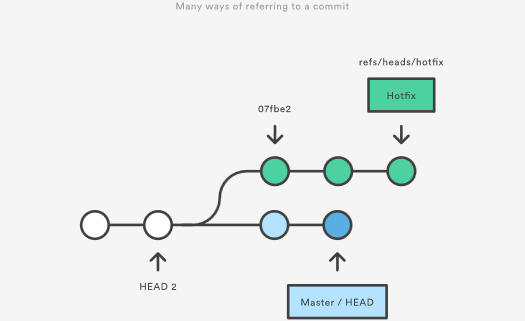
You should now be fairly comfortable using git log’s advanced parameters to format its output and select which commits you want to display. This gives you the power to pull out exactly what you need from your project history.

These new skills are an important part of your Git toolkit, but remember that git log is often used in conjunction other Git commands. Once you’ve found the commit you’re looking for, you typically pass it off to git checkout, git revert, or some other tool for manipulating your commit history. So, be sure to keep on learning about Git’s advanced features.

# Refs and the Reflog

[Hashes](https://www.atlassian.com/git/tutorials/refs-and-the-reflog#hashes) [Refs](https://www.atlassian.com/git/tutorials/refs-and-the-reflog#refs) [Packed Refs](https://www.atlassian.com/git/tutorials/refs-and-the-reflog#packed-refs) [Special Refs](https://www.atlassian.com/git/tutorials/refs-and-the-reflog#special-refs) [Refspecs](https://www.atlassian.com/git/tutorials/refs-and-the-reflog#refspecs) [Relative Refs](https://www.atlassian.com/git/tutorials/refs-and-the-reflog#relative-refs) [The Reflog](https://www.atlassian.com/git/tutorials/refs-and-the-reflog#the-reflog) [Summary](https://www.atlassian.com/git/tutorials/refs-and-the-reflog#summary)

Git is all about commits: you stage commits, create commits, view old commits, and transfer commits between repositories using many different Git commands. The majority of these commands operate on a commit in some form or another, and many of them accept a commit reference as a parameter. For example, you can use git checkout to view an old commit by passing in a commit hash, or you can use it to switch branches by passing in a branch name.



By understanding the many ways to refer to a commit, you make all of these commands that much more powerful. In this chapter, we’ll shed some light on the internal workings of common commands like git checkout, git branch, and git push by exploring the many methods of referring to a commit.

We’ll also learn how to revive seemingly “lost” commits by accessing them through Git’s reflog mechanism.

## Hashes

The most direct way to reference a commit is via its SHA-1 hash. This acts as the unique ID for each commit. You can find the hash of all your commits in the git log output.

commit 0c708fdec272bc4446c6cabea4f0022c2b616eba

Author: Mary Johnson <mary@example.com>

Date: Wed Jul 9 16:37:42 2014 -0500

Some commit message

When passing the commit to other Git commands, you only need to specify enough characters to uniquely identify the commit. For example, you can inspect the above commit with git show by running the following command:

git show 0c708f

It’s sometimes necessary to resolve a branch, tag, or another indirect reference into the corresponding commit hash. For this, you can use the git rev-parse command. The following returns the hash of the commit pointed to by the master branch:

git rev-parse master

This is particularly useful when writing custom scripts that accept a commit reference. Instead of parsing the commit reference manually, you can let git rev-parse normalize the input for you.

## Refs

A **ref** is an indirect way of referring to a commit. You can think of it as a user-friendly alias for a commit hash. This is Git’s internal mechanism of representing branches and tags.

Refs are stored as normal text files in the .git/refs directory, where .git is usually called .git. To explore the refs in one of your repositories, navigate to .git/refs. You should see the following structure, but it will contain different files depending on what branches, tags, and remotes you have in your repo:

.git/refs/

heads/

master

some-feature

remotes/

origin/

master

tags/

v0.9

The heads directory defines all of the local branches in your repository. Each filename matches the name of the corresponding branch, and inside the file you’ll find a commit hash. This commit hash is the location of the tip of the branch. To verify this, try running the following two commands from the root of the Git repository:

# Output the contents of `refs/heads/master` file:

cat .git/refs/heads/master

# Inspect the commit at the tip of the `master` branch:

git log -1 master

The commit hash returned by the cat command should match the commit ID displayed by git log.

To change the location of the master branch, all Git has to do is change the contents of the refs/heads/master file. Similarly, creating a new branch is simply a matter of writing a commit hash to a new file. This is part of the reason why Git branches are so lightweight compared to SVN.

The tags directory works the exact same way, but it contains tags instead of branches. The remotes directory lists all remote repositories that you created with git remote as separate subdirectories. Inside each one, you’ll find all the remote branches that have been fetched into your repository.

### Specifying Refs

When passing a ref to a Git command, you can either define the full name of the ref, or use a short name and let Git search for a matching ref. You should already be familiar with short names for refs, as this is what you’re using each time you refer to a branch by name.

git show some-feature

The some-feature argument in the above command is actually a short name for the branch. Git resolves this to refs/heads/some-feature before using it. You can also specify the full ref on the command line, like so:

git show refs/heads/some-feature

This avoids any ambiguity regarding the location of the ref. This is necessary, for instance, if you had both a tag and a branch called some-feature. However, if you’re using proper naming conventions, ambiguity between tags and branches shouldn’t generally be a problem.

We’ll see more full ref names in the Refspecs section.

## Packed Refs

For large repositories, Git will periodically perform a garbage collection to remove unnecessary objects and compress refs into a single file for more efficient performance. You can force this compression with the garbage collection command:

git gc

This moves all of the individual branch and tag files in the refs folder into a single file called packed-refs located in the top of the .git directory. If you open up this file, you’ll find a mapping of commit hashes to refs:

00f54250cf4e549fdfcafe2cf9a2c90bc3800285 refs/heads/feature

0e25143693cfe9d5c2e83944bbaf6d3c4505eb17 refs/heads/master

bb883e4c91c870b5fed88fd36696e752fb6cf8e6 refs/tags/v0.9

On the outside, normal Git functionality won’t be affected in any way. But, if you’re wondering why your .git/refs folder is empty, this is where the refs went.

## Special Refs

In addition to the refs directory, there are a few special refs that reside in the top-level .git directory. They are listed below:

* HEAD – The currently checked-out commit/branch.
* FETCH\_HEAD – The most recently fetched branch from a remote repo.
* ORIG\_HEAD – A backup reference to HEAD before drastic changes to it.
* MERGE\_HEAD – The commit(s) that you’re merging into the current branch with git merge.
* CHERRY\_PICK\_HEAD – The commit that you’re cherry-picking.

These refs are all created and updated by Git when necessary. For example, The git pull command first runs git fetch, which updates the FETCH\_HEAD reference. Then, it runs git merge FETCH\_HEAD to finish pulling the fetched branches into the repository. Of course, you can use all of these like any other ref, as I’m sure you’ve done with HEAD.

These files contain different content depending on their type and the state of your repository. The HEAD ref can contain either a **symbolic ref**, which is simply a reference to another ref instead of a commit hash, or a commit hash. For example, take a look at the contents of HEAD when you’re on the master branch:

git checkout master

cat .git/HEAD

This will output ref: refs/heads/master, which means that HEAD points to the refs/heads/master ref. This is how Git knows that the master branch is currently checked out. If you were to switch to another branch, the contents of HEAD would be updated to reflect the new branch. But, if you were to check out a commit instead of a branch, HEAD would contain a commit hash instead of a symbolic ref. This is how Git knows that it’s in a detached HEAD state.

For the most part, HEAD is the only reference that you’ll be using directly. The others are generally only useful when writing lower-level scripts that need to hook into Git’s internal workings.

## Refspecs

A refspec maps a branch in the local repository to a branch in a remote repository. This makes it possible to manage remote branches using local Git commands and to configure some advanced git push and git fetch behavior.

A refspec is specified as [+]<src>:<dst>. The <src> parameter is the source branch in the local repository, and the <dst> parameter is the destination branch in the remote repository. The optional + sign is for forcing the remote repository to perform a non-fast-forward update.

Refspecs can be used with the git push command to give a different name to the remote branch. For example, the following command pushes the master branch to the origin remote repo like an ordinary git push, but it uses qa-master as the name for the branch in the origin repo. This is useful for QA teams that need to push their own branches to a remote repo.

git push origin master:refs/heads/qa-master

You can also use refspecs for deleting remote branches. This is a common situation for feature-branch workflows that push the feature branches to a remote repo (e.g., for backup purposes). The remote feature branches still reside in the remote repo after they are deleted from the local repo, so you get a build-up of dead feature branches as your project progresses. You can delete them by pushing a refspec that has an empty <src> parameter, like so:

git push origin :some-feature

This is very convenient, since you don’t need to log in to your remote repository and manually delete the remote branch. Note that as of Git v1.7.0 you can use the --delete flag instead of the above method. The following will have the same effect as the above command:

git push origin --delete some-feature

By adding a few lines to the Git configuration file, you can use refspecs to alter the behavior of git fetch. By default, git fetch fetches all of the branches in the remote repository. The reason for this is the following section of the .git/config file:

[remote "origin"]

url = https://git@github.com:mary/example-repo.git

fetch = +refs/heads/\*:refs/remotes/origin/\*

The fetch line tells git fetch to download all of the branches from the origin repo. But, some workflows don’t need all of them. For example, many continuous integration workflows only care about the master branch. To fetch only the master branch, change the fetch line to match the following:

[remote "origin"]

url = https://git@github.com:mary/example-repo.git

fetch = +refs/heads/master:refs/remotes/origin/master

You can also configure git push in a similar manner. For instance, if you want to always push the master branch to qa-master in the origin remote (as we did above), you would change the config file to:

[remote "origin"]

url = https://git@github.com:mary/example-repo.git

fetch = +refs/heads/master:refs/remotes/origin/master

push = refs/heads/master:refs/heads/qa-master

Refspecs give you complete control over how various Git commands transfer branches between repositories. They let you rename and delete branches from your local repository, fetch/push to branches with different names, and configure git push and git fetch to work with only the branches that you want.

## Relative Refs

You can also refer to commits relative to another commit. The ~ character lets you reach parent commits. For example, the following displays the grandparent of HEAD:

git show HEAD~2

But, when working with merge commits, things get a little more complicated. Since merge commits have more than one parent, there is more than one path that you can follow. For 3-way merges, the first parent is from the branch that you were on when you performed the merge, and the second parent is from the branch that you passed to the git merge command.

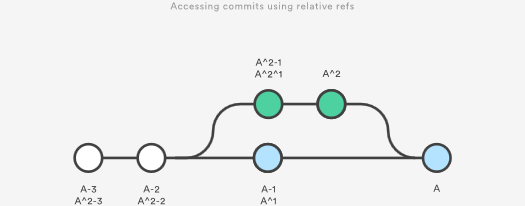
The ~ character will always follow the first parent of a merge commit. If you want to follow a different parent, you need to specify which one with the ^ character. For example, if HEAD is a merge commit, the following returns the second parent of HEAD.

git show HEAD^2

You can use more than one ^ character to move more than one generation. For instance, this displays the grandparent of HEAD (assuming it’s a merge commit) that rests on the second parent.

git show HEAD^2^1

To clarify how ~ and ^ work, the following figure shows you how to reach any commit from A using relative references. In some cases, there are multiple ways to reach a commit.



Relative refs can be used with the same commands that a normal ref can be used. For example, all of the following commands use a relative reference:

# Only list commits that are parent of the second parent of a merge commit

git log HEAD^2

# Remove the last 3 commits from the current branch

git reset HEAD~3

# Interactively rebase the last 3 commits on the current branch

git rebase -i HEAD~3

## The Reflog

The reflog is Git’s safety net. It records almost every change you make in your repository, regardless of whether you committed a snapshot or not. You can think of it as a chronological history of everything you’ve done in your local repo. To view the reflog, run the git reflog command. It should output something that looks like the following:

400e4b7 HEAD@{0}: checkout: moving from master to HEAD~2

0e25143 HEAD@{1}: commit (amend): Integrate some awesome feature into `master`

00f5425 HEAD@{2}: commit (merge): Merge branch ';feature';

ad8621a HEAD@{3}: commit: Finish the feature

This can be translated as follows:

* You just checked out HEAD~2
* Before that you amended a commit message
* Before that you merged the feature branch into master
* Before that you committed a snapshot

The HEAD{<n>} syntax lets you reference commits stored in the reflog. It works a lot like the HEAD~<n> references from the previous section, but the <n> refers to an entry in the reflog instead of the commit history.

You can use this to revert to a state that would otherwise be lost. For example, lets say you just scrapped a new feature with git reset. Your reflog might look something like this:

ad8621a HEAD@{0}: reset: moving to HEAD~3

298eb9f HEAD@{1}: commit: Some other commit message

bbe9012 HEAD@{2}: commit: Continue the feature

9cb79fa HEAD@{3}: commit: Start a new feature

The three commits before the git reset are now dangling, which means that there is no way to reference them—except through the reflog. Now, let’s say you realize that you shouldn’t have thrown away all of your work. All you have to do is check out the HEAD@{1} commit to get back to the state of your repository before you ran git reset.

git checkout HEAD@{1}

This puts you in a detached HEAD state. From here, you can create a new branch and continue working on your feature.

## Summary

You should now be quite comfortable referring to commits in a Git repository. We learned how branches and tags were stored as refs in the .git subdirectory, how to read a packed-refs file, how HEAD is represented, how to use refspecs for advanced pushing and fetching, and how to use the relative ~ and ^ operators to traverse a branch hierarchy.

We also took a look at the reflog, which is a way to reference commits that are not available through any other means. This is a great way to recover from those little “Oops, I shouldn’t have done that” situations.

The point of all this was to be able to pick out exactly the commit that you need in any given development scenario. It’s very easy to leverage the skills you learned in this article against your existing Git knowledge, as some of the most common commands accept refs as arguments, including git log, git show, git checkout, git reset, git revert, git rebase, and many others.

Terminology

### Branch

A branch represents an independent line of development. Branches serve as an abstraction for the edit/stage/commit process discussed in Git Basics, the first module of this series. You can think of them as a way to request a brand new working directory, staging area, and project history. New commits are recorded in the history for the current branch, which results in a fork in the history of the project.

### HEAD

Git’s way of referring to the current snapshot. Internally, the git checkout command simply updates the HEAD to point to either the specified branch or commit. When it points to a branch, Git doesn't complain, but when you check out a commit, it switches into a “detached HEAD” state.

### Hook

A script that runs automatically every time a particular event occurs in a Git repository. Hooks let you customize Git’s internal behavior and trigger customizable actions at key points in the development life cycle.

### Master

The default development branch. Whenever you create a git repository, a branch named "master" is created, and becomes the active branch.

### Tag

A reference typically used to mark a particular point in the commit chain. In contrast to a head, a tag is not updated by the commit command.

### Version Control

A system that records changes to a file or set of files over time so that you can recall specific versions later.

### Working Tree

The tree of actual checked out files, normally containing the contents of the HEAD commit's tree and any local changes you've made but haven't yet committed.

# Git commands

### git add

Moves changes from the working directory to the staging area. This gives you the opportunity to prepare a snapshot before committing it to the official history.

### git branch

This command is your general-purpose branch administration tool. It lets you create isolated development environments within a single repository.

### git checkout

In addition to checking out old commits and old file revisions, git checkout is also the means to navigate existing branches. Combined with the basic Git commands, it’s a way to work on a particular line of development.

### git clean

Removes untracked files from the working directory. This is the logical counterpart to git reset, which (typically) only operates on tracked files.

### git clone

Creates a copy of an existing Git repository. Cloning is the most common way for developers to obtain a working copy of a central repository.

### git commit

Takes the staged snapshot and commits it to the project history. Combined with git add, this defines the basic workflow for all Git users.

### git commit --amend

Passing the --amend flag to git commit lets you amend the most recent commit. This is very useful when you forget to stage a file or omit important information from the commit message.

### git config

A convenient way to set configuration options for your Git installation. You’ll typically only need to use this immediately after installing Git on a new development machine.

### git fetch

Fetching downloads a branch from another repository, along with all of its associated commits and files. But, it doesn't try to integrate anything into your local repository. This gives you a chance to inspect changes before merging them with your project.

### git init

Initializes a new Git repository. If you want to place a project under revision control, this is the first command you need to learn.

### git log

Lets you explore the previous revisions of a project. It provides several formatting options for displaying committed snapshots.

### git merge

A powerful way to integrate changes from divergent branches. After forking the project history with git branch, git merge lets you put it back together again.

### git pull

Pulling is the automated version of git fetch. It downloads a branch from a remote repository, then immediately merges it into the current branch. This is the Git equivalent of svn update.

### git push

Pushing is the opposite of fetching (with a few caveats). It lets you move a local branch to another repository, which serves as a convenient way to publish contributions. This is like svn commit, but it sends a series of commits instead of a single changeset.

### git rebase

Rebasing lets you move branches around, which helps you avoid unnecessary merge commits. The resulting linear history is often much easier to understand and explore.

### git rebase -i

The -i flag is used to begin an interactive rebasing session. This provides all the benefits of a normal rebase, but gives you the opportunity to add, edit, or delete commits along the way

### git reflog

Git keeps track of updates to the tip of branches using a mechanism called reflog. This allows you to go back to changesets even though they are not referenced by any branch or tag.

### git remote

A convenient tool for administering remote connections. Instead of passing the full URL to the fetch, pull, and push commands, it lets you use a more meaningful shortcut.

### git reset

Undoes changes to files in the working directory. Resetting lets you clean up or completely remove changes that have not been pushed to a public repository.

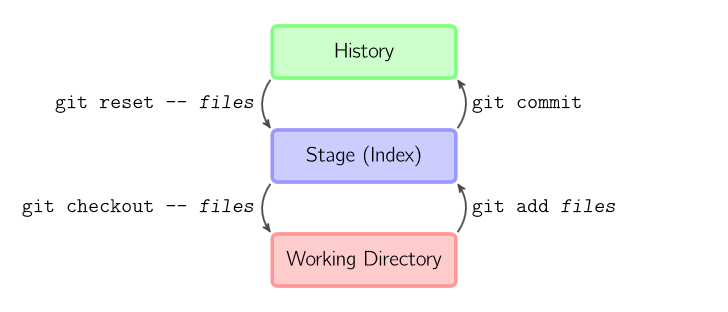
### git revert

Undoes a committed snapshot. When you discover a faulty commit, reverting is a safe and easy way to completely remove it from the code base.

### git status

Displays the state of the working directory and the staged snapshot. You’ll want to run this in conjunction with git add and git commit to see exactly what’s being included in the next snapshot.

<http://marklodato.github.io/visual-git-guide/index-en.html>

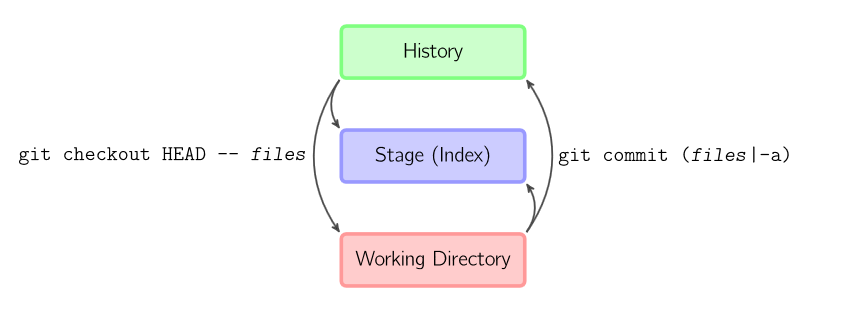


The four commands above copy files between the working directory, the stage (also called the index), and the history (in the form of commits).

* git add *files* copies *files* (at their current state) to the stage.
* git commit saves a snapshot of the stage as a commit.
* git reset -- *files* unstages files; that is, it copies *files* from the latest commit to the stage. Use this command to "undo" a git add *files*. You can also git reset to unstage everything.
* git checkout -- *files* copies *files* from the stage to the working directory. Use this to throw away local changes.

You can use git reset -p, git checkout -p, or git add -p instead of (or in addition to) specifying particular files to interactively choose which hunks copy.

It is also possible to jump over the stage and check out files directly from the history or commit files without staging first.

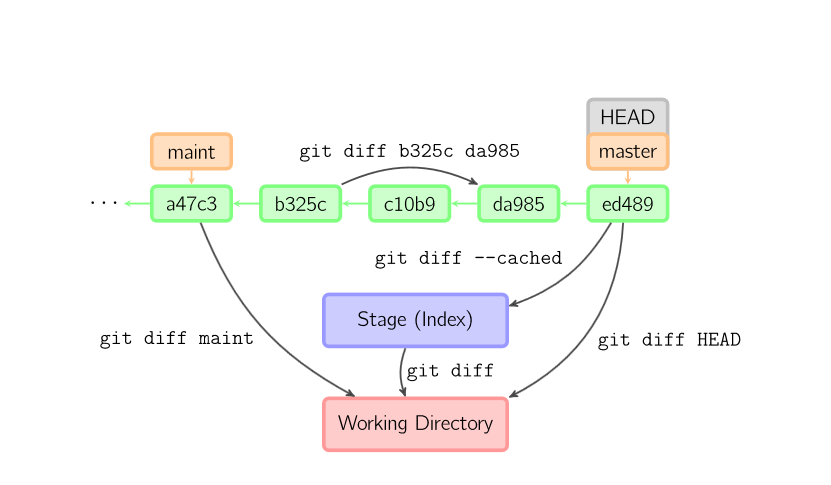


* git commit -a is equivalent to running git add on all filenames that existed in the latest commit, and then running git commit.
* git commit *files* creates a new commit containing the contents of the latest commit, plus a snapshot of *files* taken from the working directory. Additionally, *files* are copied to the stage.
* git checkout HEAD -- *files* copies *files* from the latest commit to both the stage and the working directory.

## Commands in Detail

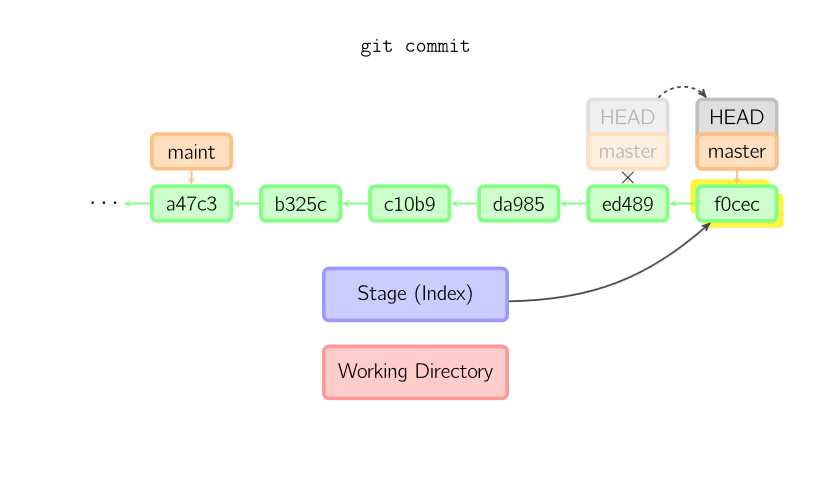
### Diff

There are various ways to look at differences between commits. Below are some common examples. Any of these commands can optionally take extra filename arguments that limit the differences to the named files

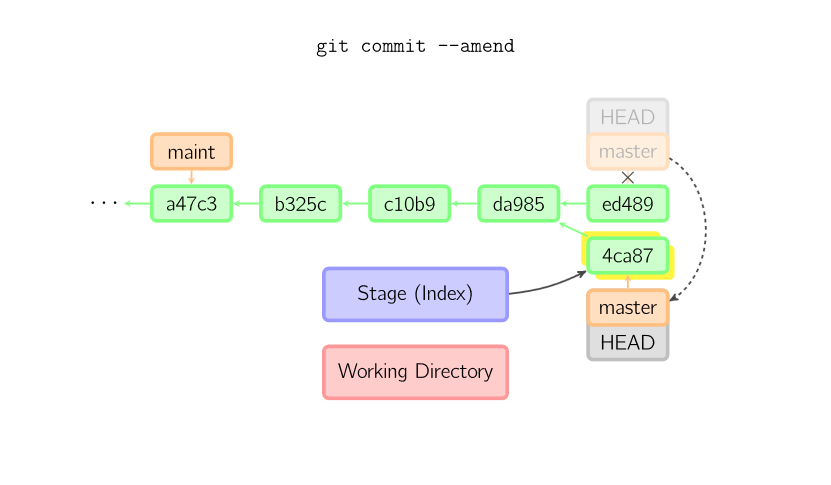


### Commit

When you commit, git creates a new commit object using the files from the stage and sets the parent to the current commit. It then points the current branch to this new commit. In the image below, the current branch is master. Before the command was run, master pointed to ed489. Afterward, a new commit, f0cec, was created, with parent ed489, and then master was moved to the new commit



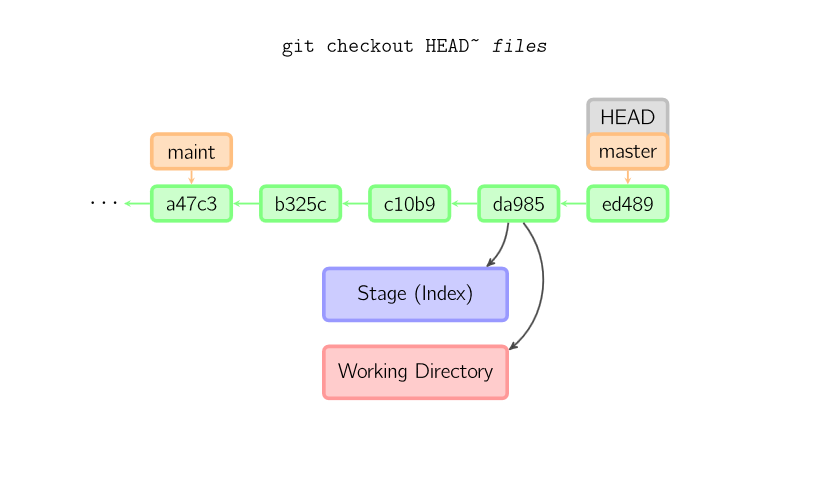
Sometimes a mistake is made in a commit, but this is easy to correct with git commit --amend. When you use this command, git creates a new commit with the same parent as the current commit. (The old commit will be discarded if nothing else references it.)



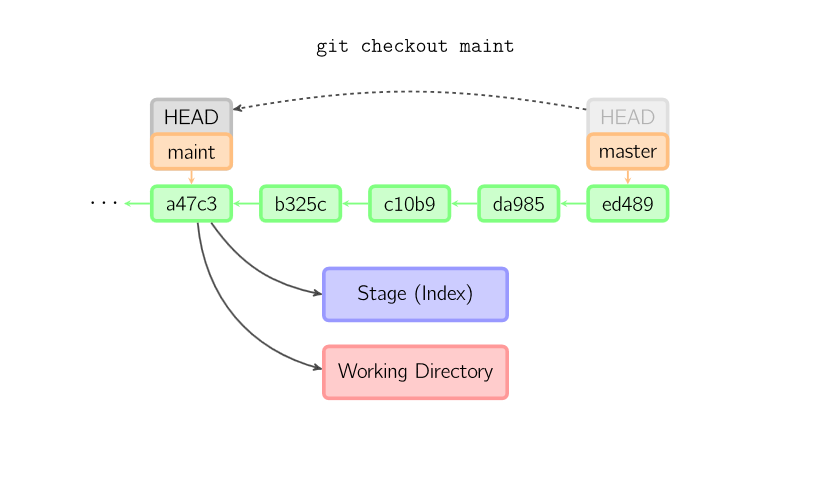
### Checkout

The checkout command is used to copy files from the history (or stage) to the working directory, and to optionally switch branches.

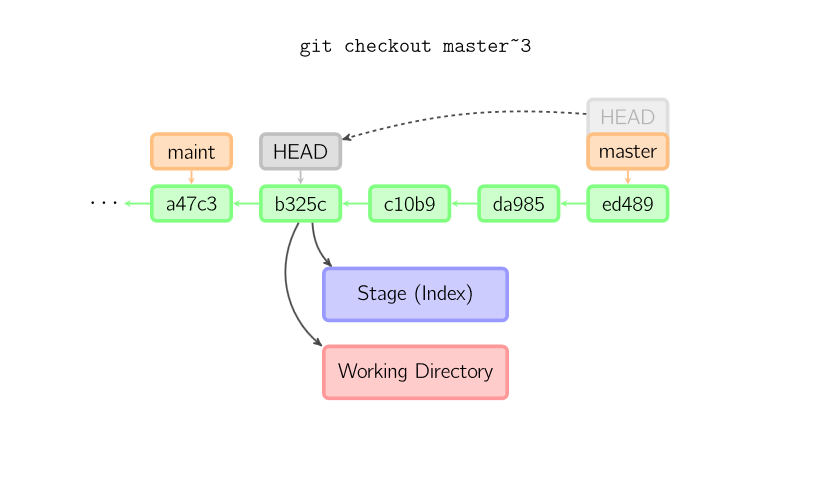
When a filename (and/or -p) is given, git copies those files from the given commit to the stage and the working directory. For example, git checkout HEAD~ foo.c copies the file foo.c from the commit called HEAD~ (the parent of the current commit) to the working directory, and also stages it. (If no commit name is given, files are copied from the stage.) Note that the current branch is not changed.



When a filename is not given but the reference is a (local) branch, HEAD is moved to that branch (that is, we "switch to" that branch), and then the stage and working directory are set to match the contents of that commit. Any file that exists in the new commit (a47c3 below) is copied; any file that exists in the old commit (ed489) but not in the new one is deleted; and any file that exists in neither is ignored.

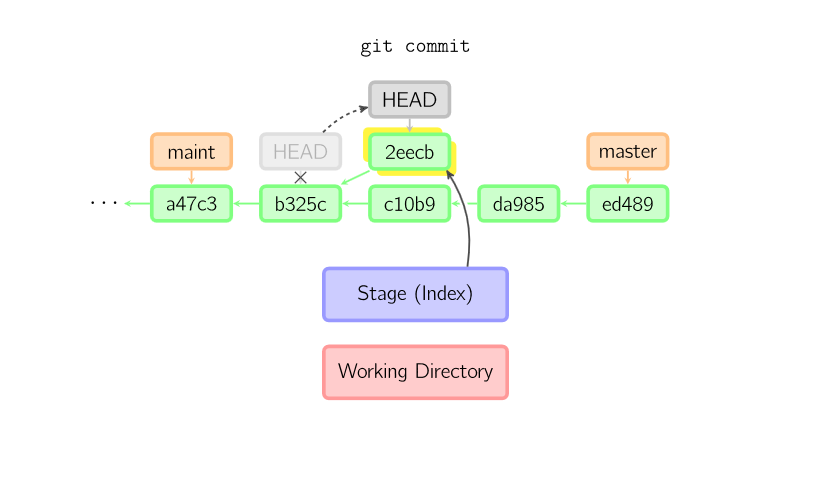


When a filename is not given and the reference is not a (local) branch — say, it is a tag, a remote branch, a SHA-1 ID, or something like master~3 — we get an anonymous branch, called a detached HEAD. This is useful for jumping around the history. Say you want to compile version 1.6.6.1 of git. You can git checkout v1.6.6.1 (which is a tag, not a branch), compile, install, and then switch back to another branch, say git checkout master. However, committing works slightly differently with a detached HEAD; this is covered [below](http://marklodato.github.io/visual-git-guide/index-en.html#detached).

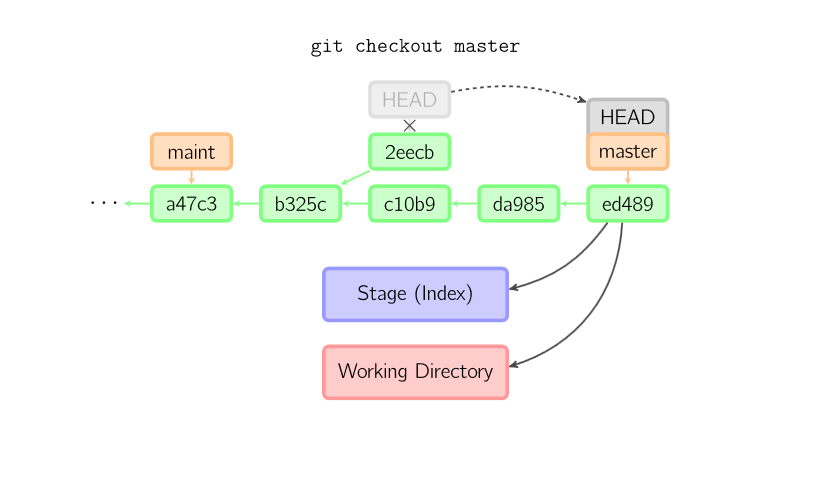


### Committing with a Detached HEAD

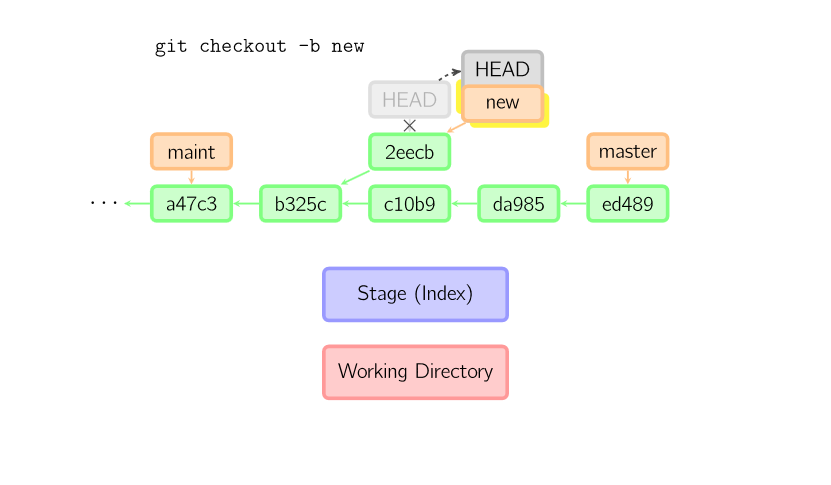
When HEAD is detached, commits work like normal, except no named branch gets updated. (You can think of this as an anonymous branch.)



Once you check out something else, say master, the commit is (presumably) no longer referenced by anything else, and gets lost. Note that after the command, there is nothing referencing 2eecb.



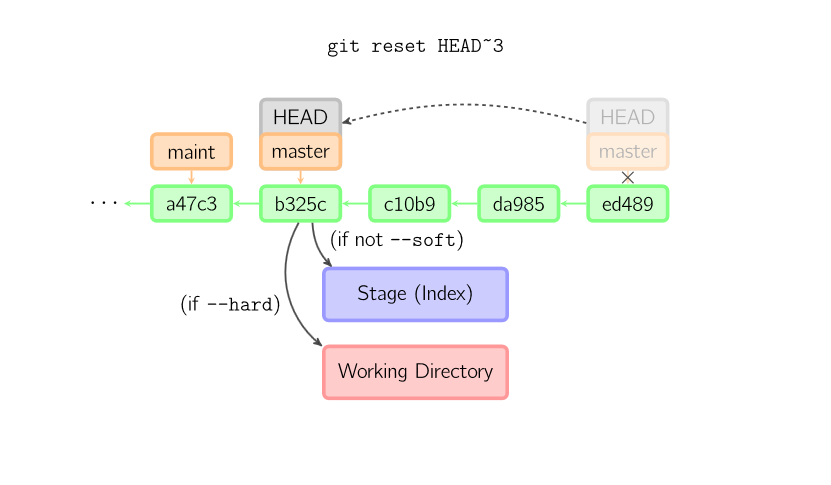
If, on the other hand, you want to save this state, you can create a new named branch using git checkout -b name



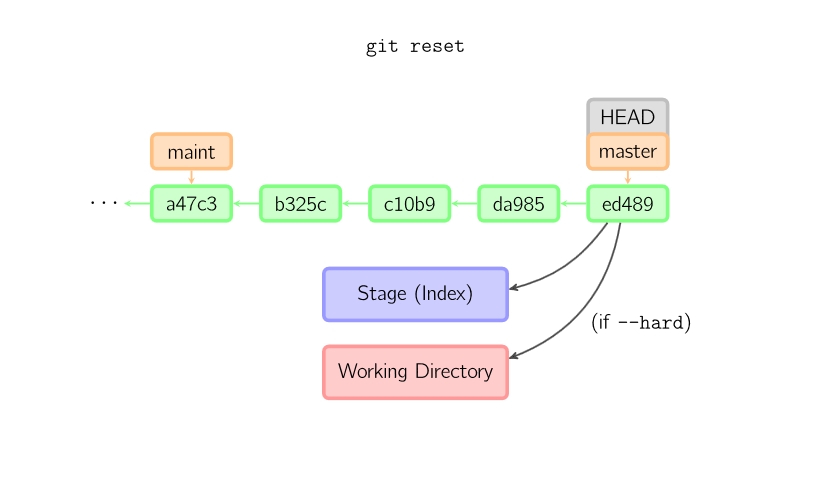
### Reset

The reset command moves the current branch to another position, and optionally updates the stage and the working directory. It also is used to copy files from the history to the stage without touching the working directory.

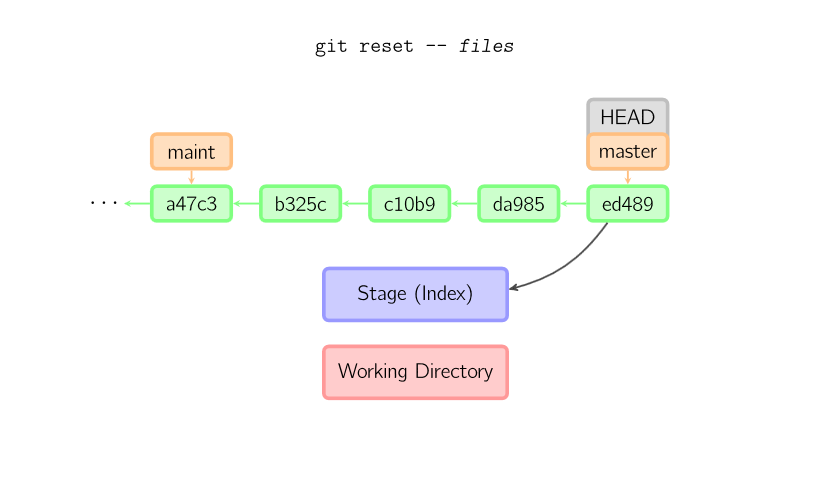
If a commit is given with no filenames, the current branch is moved to that commit, and then the stage is updated to match this commit. If --hard is given, the working directory is also updated. If --soft is given, neither is updated.



If a commit is not given, it defaults to HEAD. In this case, the branch is not moved, but the stage (and optionally the working directory, if --hard is given) are reset to the contents of the last commit

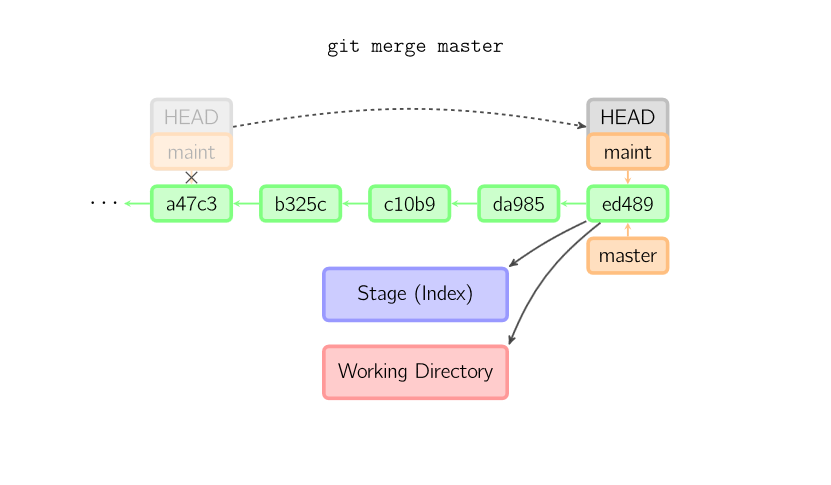


If a filename (and/or -p) is given, then the command works similarly to [checkout](http://marklodato.github.io/visual-git-guide/index-en.html#checkout) with a filename, except only the stage (and not the working directory) is updated. (You may also specify the commit from which to take files, rather than HEAD.)

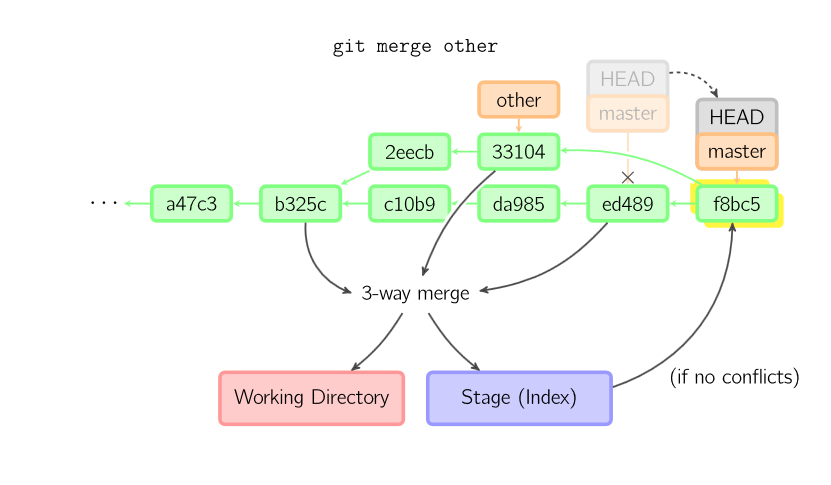


### Merge

A merge creates a new commit that incorporates changes from other commits. Before merging, the stage must match the current commit. The trivial case is if the other commit is an ancestor of the current commit, in which case nothing is done. The next most simple is if the current commit is an ancestor of the other commit. This results in a fast-forward merge. The reference is simply moved, and then the new commit is checked out.

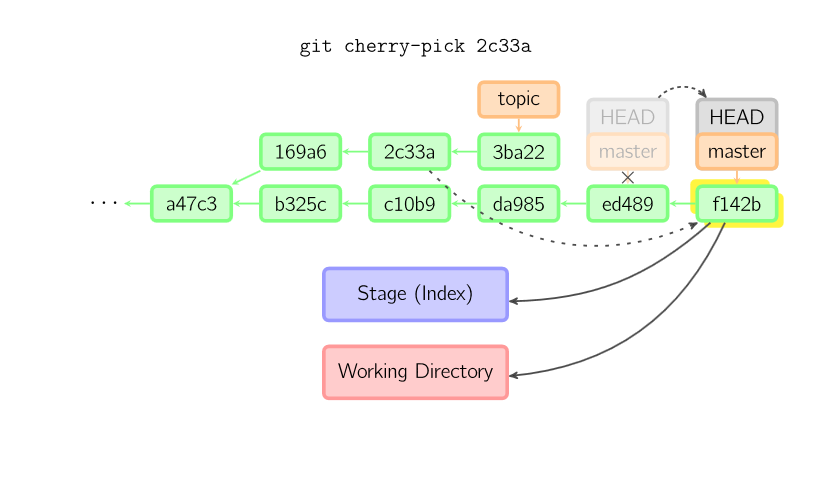


Otherwise, a "real" merge must occur. You can choose other strategies, but the default is to perform a "recursive" merge, which basically takes the current commit (ed489 below), the other commit (33104), and their common ancestor (b325c), and performs a [three-way merge](http://en.wikipedia.org/wiki/Three-way_merge). The result is saved to the working directory and the stage, and then a commit occurs, with an extra parent (33104) for the new commit.



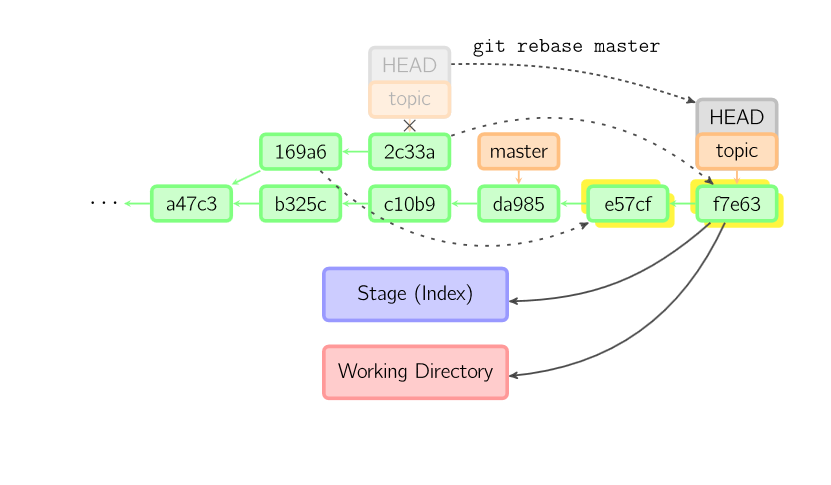
### Cherry Pick

The cherry-pick command "copies" a commit, creating a new commit on the current branch with the same message and patch as another commit.



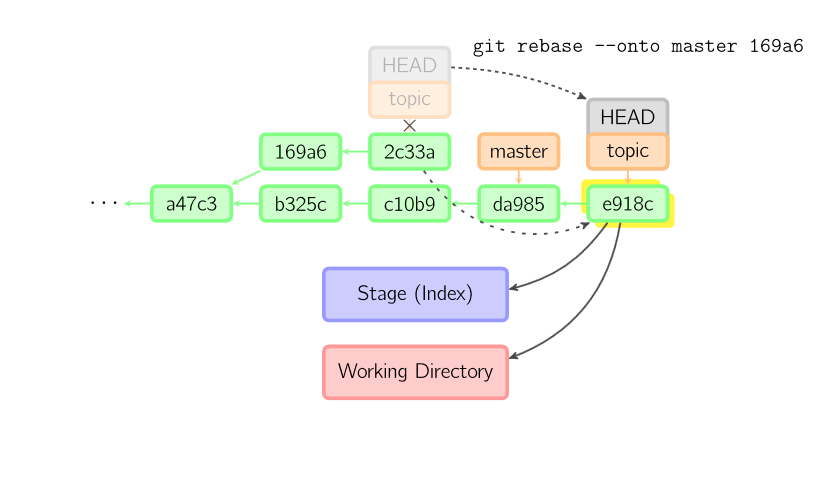
### Rebase

A rebase is an alternative to a [merge](http://marklodato.github.io/visual-git-guide/index-en.html#merge) for combining multiple branches. Whereas a merge creates a single commit with two parents, leaving a non-linear history, a rebase replays the commits from the current branch onto another, leaving a linear history. In essence, this is an automated way of performing several [cherry-pick](http://marklodato.github.io/visual-git-guide/index-en.html#cherry-pick)s in a row.



The above command takes all the commits that exist in topic but not in master (namely 169a6 and 2c33a), replays them onto master, and then moves the branch head to the new tip. Note that the old commits will be garbage collected if they are no longer referenced.

To limit how far back to go, use the --onto option. The following command replays onto master the most recent commits on the current branch since 169a6 (exclusive), namely 2c33a



There is also git rebase --interactive, which allows one to do more complicated things than simply replaying commits, namely dropping, reordering, modifying, and squashing commits. There is no obvious picture to draw for this; see [git-rebase(1)](http://www.kernel.org/pub/software/scm/git/docs/git-rebase.html#_interactive_mode) for more details.

The contents of files are not actually stored in the index (.git/index) or in commit objects. Rather, each file is stored in the object database (.git/objects) as a blob, identified by its SHA-1 hash. The index file lists the filenames along with the identifier of the associated blob, as well as some other data. For commits, there is an additional data type, a tree, also identified by its hash. Trees correspond to directories in the working directory, and contain a list of trees and blobs corresponding to each filename within that directory. Each commit stores the identifier of its top-level tree, which in turn contains all of the blobs and other trees associated with that commit.

If you make a commit using a detached HEAD, the last commit really is referenced by something: the reflog for HEAD. However, this will expire after a while, so the commit will eventually be garbage collected, similar to commits discarded with git commit --amend or git rebase.

## Walkthrough: Watching the effect of commands

The following walks you through changes to a repository so you can see the effect of the command in real time, similar to how [Visualizing Git Concepts with D3](http://onlywei.github.io/explain-git-with-d3/) simulates them visually. Hopefully you find this useful.

Start by creating some repository:

$ **git init foo**

$ **cd foo**

$ **echo 1 > myfile**

$ **git add myfile**

$ **git commit -m "version 1"**

Now, define the following functions to help us show information:

show\_status() {

echo "HEAD: $(git cat-file -p HEAD:myfile)"

echo "Stage: $(git cat-file -p :myfile)"

echo "Worktree: $(cat myfile)"

}

initial\_setup() {

echo 3 > myfile

git add myfile

echo 4 > myfile

show\_status

}

Initially, everything is at version 1.

$ **show\_status**

HEAD: 1

Stage: 1

Worktree: 1

We can watch the state change as we add and commit.

$ **echo 2 > myfile**

$ **show\_status**

HEAD: 1

Stage: 1

Worktree: 2

$ **git add myfile**

$ **show\_status**

HEAD: 1

Stage: 2

Worktree: 2

$ **git commit -m "version 2"**

[master 4156116] version 2

1 file changed, 1 insertion(+), 1 deletion(-)

$ **show\_status**

HEAD: 2

Stage: 2

Worktree: 2

Now, let's create an initial state where the three are all different.

$ **initial\_setup**

HEAD: 2

Stage: 3

Worktree: 4

Let's watch what each command does. You will see that they match the diagrams above.

git reset -- myfile copies from HEAD to stage:

$ **initial\_setup**

HEAD: 2

Stage: 3

Worktree: 4

$ **git reset -- myfile**

Unstaged changes after reset:

M myfile

$ **show\_status**

HEAD: 2

Stage: 2

Worktree: 4

git checkout -- myfile copies from stage to worktree:

$ **initial\_setup**

HEAD: 2

Stage: 3

Worktree: 4

$ **git checkout -- myfile**

$ **show\_status**

HEAD: 2

Stage: 3

Worktree: 3

git checkout HEAD -- myfile copies from HEAD to both stage and worktree:

$ **initial\_setup**

HEAD: 2

Stage: 3

Worktree: 4

$ **git checkout HEAD -- myfile**

$ **show\_status**

HEAD: 2

Stage: 2

Worktree: 2

git commit myfile copies from worktree to both stage and HEAD:

$ **initial\_setup**

HEAD: 2

Stage: 3

Worktree: 4

$ **git commit myfile -m "version 4"**

[master 679ff51] version 4

1 file changed, 1 insertion(+), 1 deletion(-)

$ **show\_status**

HEAD: 4

Stage: 4

Worktree: 4

<https://davidwalsh.name/tutorials/git>

How to Batch Update Git Commit Messages

**Prepending to Commit Messages**

To prepend text to every commit message in a given range, you'd execute a message like:

git filter-branch --msg-filter 'echo "bug ###### - \c" && cat' master..HEAD

You can also sed to achieve this:

git filter-branch -f --msg-filter 'sed "s/^/bug ###### - /"' master..HEAD

## Appending to Commit Messages

The case for appending to commit messages could be where you want to add the reviewer name(s) to the message.  Appending is roughly the same:

git filter-branch -f --msg-filter 'cat && echo "[Reviewer Walsh]"' master..HEAD

Quick git Commit Searching

One frequent git task is searching a list of commits on master branch which match a given keyword.  Here's how I do that:

git log -i --grep='ckeditor'

Create a Repository Archive with git

By [David Walsh](http://davidwalsh.name) on September 28, 2015

One feature I recently found out about is its archive feature which allows for exporting an entire repository to a zip or tar file.

# Format: git archive {branchname} --format={compression} --output={filename}

git archive master --format=tar --output=kuma.tar

git archive some-feature-branch --format=tar --output=kuma.tar

Sure you could use any archiving utility to archive a given repo, this feature allows for quick archiving of any branch or repository state!

List Recent git Commits from Command Line

By [David Walsh](http://davidwalsh.name) on January 5, 2016

.  I found a useful command for listing commits newest to oldest so on [commandlinefu](http://www.commandlinefu.com/commands/view/15064/show-a-git-log-with-offsets-relative-to-head):

git log --oneline | nl -v0 | sed 's/^ \+/&HEAD~/'

That command will render a listing with the latest commits first and original commit last:

Track Empty Directories with git

# Track Empty Directories with git

By [David Walsh](http://davidwalsh.name) on May 30, 2017

There are times when you'd like to track an empty directory within git but there's a problem: git wont allow you to add a directory that doesn't have a file in it.  The easy solution is putting an empty stub file within the directory, and the industry standard for that stub file name is .gitkeep

You can quickly create the file and commit the "empty" directory from command line:

touch my-empty-dir/.gitkeep

git add my-empty-dir/.gitkeep

git commit -m "Adding my empty directory"

The problem is simple, the solution is easy, but I wanted to highlight that .gitkeep is the industry standard.

Checkout the Previous Branch with git

By [David Walsh](http://davidwalsh.name) on June 29, 2017

I recently found out that you can switch the the previous branch you were on using the following command:

git checkout master

# Do whatever

git pull remote master

# Go back to the previous branch

git checkout -

Using - references the previous branch name, thus allowing you to navigate branches with ease!

Undo File Changes with Git

# Undo File Changes with Git

By [David Walsh](http://davidwalsh.name) on July 10, 2017

* To quickly undo file changes with git, execute the following two commands:
* git reset HEAD path/to/file.ext
* git checkout path/to/file.ext
* The second command (checkout) is required or you'll still see the file listed when running git status again.  With both of those executions, you'll no longer see the file listed with git status.
* git makes version control easy but the two steps needed to essentially revert changes to a file aren't intuitive, thus I thought I would share on this blog.  Happy coding!
* How to Delete a git Remote Branch

# How to Delete a git Remote Branch

* By [David Walsh](http://davidwalsh.name) on March 4, 2018

Once a branch is merged, for example, [we no longer need it around](https://davidwalsh.name/delete-merged-branches-git).

Deleting a branch on a local host machine repo is easy:

git branch -d <branch\_name>

To remove a branch from the remote git repository, like a GitHub-hosted repository, you can execute:

git push <remote\_name> --delete <branch\_name>

Reset File Changes with git

# Reset File Changes with git

By [David Walsh](http://davidwalsh.name) on April 24, 2018

You can restore a file's contents before a patch with the following:

git reset origin/master path/to/file-to-be-changed.ext

Once this shell snippet is executed, the file's contents are restored and can be re-commited to restore the file contents.

Search Git Commits Between Dates

# Search Git Commits Between Dates

By [David Walsh](http://davidwalsh.name) on July 11, 2018

You can use the following git command to list commits between two dates:

$ git log --after="2018-06-30" --before="2018-07-03" --oneline

Especially nice is the --oneline modifier to keep the commit list concise.