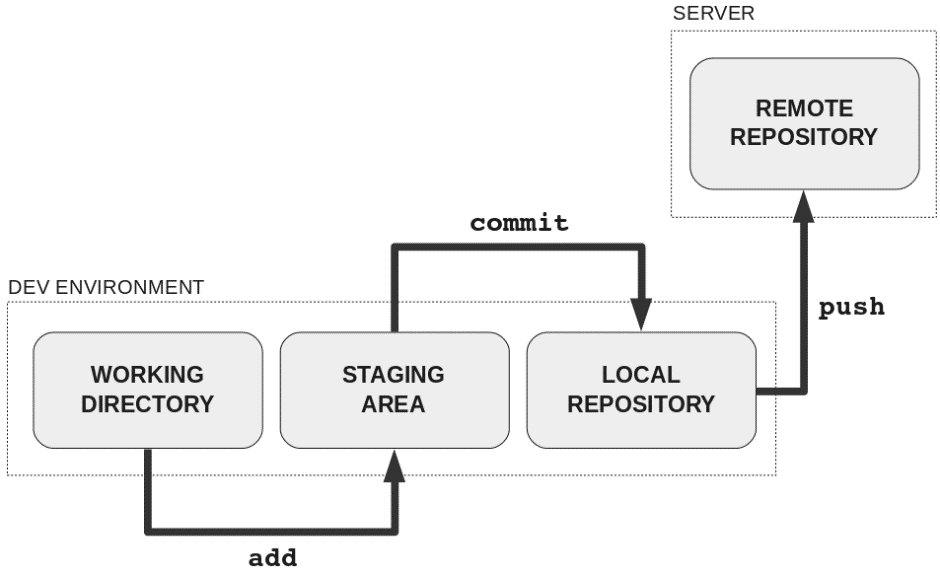
## Saving changes to the repository: git add and git commit



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.

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 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.

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

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

## 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.

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

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

## git 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).

### 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 commit

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

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

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

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

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

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

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

Change the message displayed by hello.py

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

- Change the sayGoodbye() function to a friendlier message

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

git add hello.py

git commit –amend

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

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

# 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)

## Normal Workflow

add files to the index:

$ git add file1 file2 file3

You are now ready to commit. You can see what is about to be committed using linkgit:git-diff[1] with the --cached option:

$ git diff --cached

You can also get a brief summary of the situation with git-status

$ git commit

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

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

$ git commit -a

which will automatically notice any modified (but not new) files, add them to the index, and commit, all in one step

Git's "add" command does something simpler and more powerful: git add is used both for new and newly modified files, and in both cases it takes a snapshot of the given files and stages that content in the index, ready for inclusion in the next commit.

# 2 Git Basics

**Short Status**

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

$ git status –s

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

**Viewing Your Staged and Unstaged Changes**

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

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

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

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

*Git Diff in an External Tool*

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

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

$ git commit -am 'added new benchmarks'

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

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

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

$ git rm --cached README

That means you can do things such as:

$ git rm log/\*.log

$ git rm /\*~

**Moving Files**

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

$ git mv file\_from file\_to

However, this is equivalent to running something like this:

$ mv README.md README

$ git rm README.md

$ git add README

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

the add/rm later, before you commit.