# Saving changes

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

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 main 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.

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

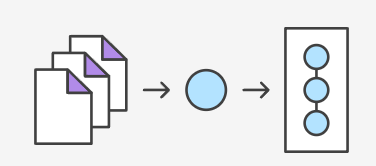
The git add command adds a change in the working directory to the staging area. It tells Git that you want to include updates to a particular file in the next commit. However, git add doesn't really affect the repository in any significant way—changes are not actually recorded until you run [git commit](https://www.atlassian.com/git/tutorials/saving-changes/git-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

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

## Git commit vs SVN commit

While they share the same name, git commit is nothing like svn commit. This shared term can be a point of confusion for Git newcomers who have a svn background, and it is important to emphasize the difference. To compare git commit vs svn commit is to compare a centralized application model (svn) vs a distributed application model (Git). In SVN, a commit pushes changes from the local SVN client, to a remote centralized shared SVN repository. 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.

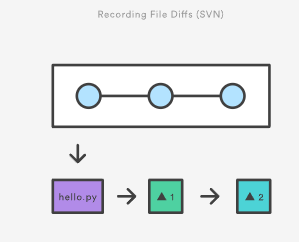
## How it works

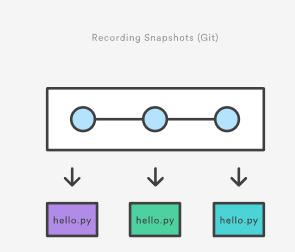
At a high-level, Git can be thought of as a timeline management utility. Commits are the core building block units of a Git project timeline. Commits can be thought of as snapshots or milestones along the timeline of a Git project. Commits are created with the git commit command to capture the state of a project at that point in time. Git Snapshots are always committed to the local repository. This is fundamentally different from SVN, wherein the working copy is committed to the central repository. In contrast, Git doesn’t force you to interact with the central repository until you’re ready. Just as the staging area is a buffer between the working directory and the project history, each developer’s local repository is a buffer between their contributions and the central repository.

This changes the basic development model for Git users. Instead of making a change and committing it directly to the central repo, Git developers have the opportunity to accumulate commits in their local repo. This has many advantages over SVN-style collaboration: it makes it easier to split up a feature into atomic commits, keep related commits grouped together, and clean up local history before publishing it to the central repository. It also lets developers work in an isolated environment, deferring integration until they’re at a convenient point to merge with other users. While isolation and deferred integration are individually beneficial, it is in a team's best interest to integrate frequently and in small units. For more information regarding best practices for Git team collaboration read how teams structure their [Git workflow](https://www.atlassian.com/git/tutorials/comparing-workflows).

## Snapshots, not differences

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.

## Common options

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.

## Examples

### Saving changes with a commit

The following example assumes you’ve edited some content in a file called hello.py on the current branch, and are ready to commit it to the project history. First, you need to stage the file with git add, then you can commit the staged snapshot.

git add hello.py

This command will add hello.py to the Git staging area. We can examine the result of this action by using the git status command.

git status  
On branch main  
Changes to be committed:  
  (use "git reset HEAD <file>..." to unstage)  
   new file: hello.py

The green output new file: hello.py indicates that hello.py will be saved with the next commit. From the commit is created by executing:

git commit

This will open a text editor (customizable via git config) asking for a commit log message, along with a list of what’s being committed:

# Please enter the commit message for your changes. Lines starting  
# with '#' will be ignored, and an empty message aborts the commit.  
# On branch main  
# Changes to be committed:  
# (use "git reset HEAD ..." to unstage)  
#  
#modified: hello.py

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.

## How to update (amend) a commit

To continue with the hello.py example above. Let's make further updates to hello.py and execute the following:

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.

## Summary

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.

The commit model of SVN and Git are significantly different but often confused, because of the shared terminology. If you are coming to Git from a personal history of SVN usage, it is good to learn that in Git, commits are cheap and should be used frequently. Whereas SVN commits are an expensive operation that makes a remote request, Git commits are done locally and with a more efficient algorithm.

# Git diff

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

## Comparing changes with git diff

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

## Reading diffs: outputs

### Raw output format

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

$:> mkdir diff\_test\_repo  
$:> cd diff\_test\_repo  
$:> touch diff\_test.txt  
$:> echo "this is a git diff test example" > diff\_test.txt  
$:> git init .  
Initialized empty Git repository in /Users/kev/code/test/.git/  
$:> git add diff\_test.txt  
$:> git commit -am"add diff test file"  
[main (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 main 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-stash) [**git commit**](https://www.atlassian.com/git/tutorials/saving-changes/git-stash) [**git diff**](https://www.atlassian.com/git/tutorials/saving-changes/git-stash) [**git stash**](https://www.atlassian.com/git/tutorials/saving-changes/git-stash) [**.gitignore**](https://www.atlassian.com/git/tutorials/saving-changes/git-stash)

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 main  
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 main: 5002d47 our new homepage  
HEAD is now at 5002d47 our new homepage  
  
$ git status  
On branch main  
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 main  
nothing to commit, working tree clean  
$ git stash pop  
On branch main  
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 main  
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/saving-changes/gitignore)

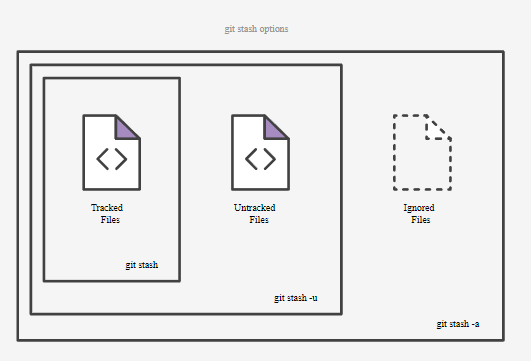
So if we add a third file to our example above, but don't stage it (i.e. we don't run git add), git stash won't stash it.

$ script.js  
  
$ git status  
On branch main  
Changes to be committed:  
  
    new file:   style.css  
  
Changes not staged for commit:  
  
    modified:   index.html  
  
Untracked files:  
  
    script.js  
  
$ git stash  
Saved working directory and index state WIP on main: 5002d47 our new homepage  
HEAD is now at 5002d47 our new homepage  
  
$ git status  
On branch main  
Untracked files:  
  
    script.js

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

$ git status  
On branch main  
Changes to be committed:  
  
    new file:   style.css  
  
Changes not staged for commit:  
  
    modified:   index.html  
  
Untracked files:  
  
    script.js  
  
$ git stash -u  
Saved working directory and index state WIP on main: 5002d47 our new homepage  
HEAD is now at 5002d47 our new homepage  
  
$ git status  
On branch main  
nothing to commit, working tree clean

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 main: 5002d47 our new homepage  
stash@{1}: WIP on main: 5002d47 our new homepage  
stash@{2}: WIP on main: 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 main: add style to our site  
HEAD is now at 5002d47 our new homepage  
  
$ git stash list  
stash@{0}: On main: add style to our site  
stash@{1}: WIP on main: 5002d47 our new homepage  
stash@{2}: WIP on main: 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(+)

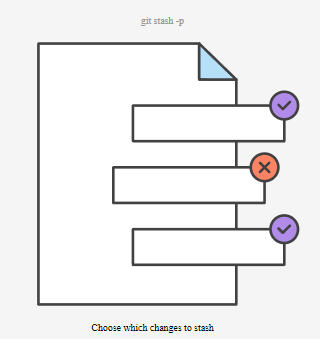
Or pass the -p option (or --patch) to view the full diff of a stash:

$ 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



You can hit **?** for a full list of hunk commands. Commonly useful ones are:

| **Command** | **Description** |
| --- | --- |
| / | search for a hunk by regex |
| ? | help |
| n | don't stash this hunk |
| q | quit (any hunks that have already been selected will be stashed) |
| s | split this hunk into smaller hunks |
| y | stash this hunk |

There is no explicit "abort" command, but hitting CTRL-C(SIGINT) will abort the stash process.

## 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:

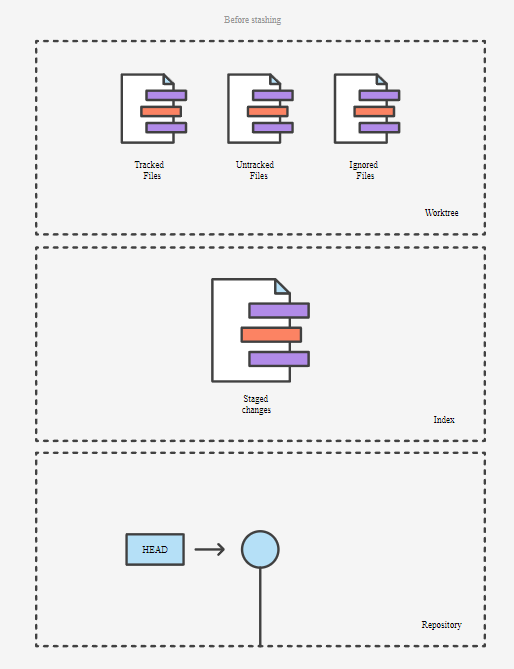
$ git log --oneline --graph stash@{0}  
\*-.   953ddde WIP on main: 5002d47 our new homepage  
|\ \   
| | \* 24b35a1 untracked files on main: 5002d47 our new homepage  
| \* 7023dd4 index on main: 5002d47 our new homepage  
|/   
\* 5002d47 our new homepage

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

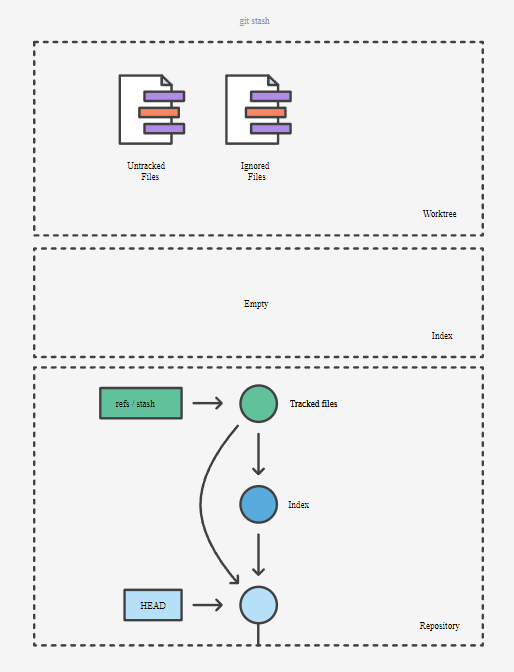
* stash@{0}, a new commit to store the tracked files that were in your working copy when you ran git stash
* stash@{0}'s first parent, the pre-existing commit that was at HEAD when you ran git stash
* stash@{0}'s second parent, a new commit representing the index when you ran git stash
* stash@{0}'s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

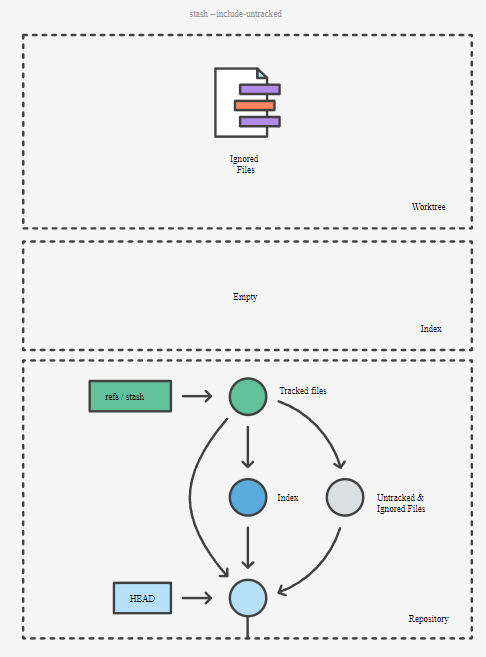
* Before stashing, your worktree may contain changes to tracked files, untracked files, and ignored files. Some of these changes may also be staged in the index.



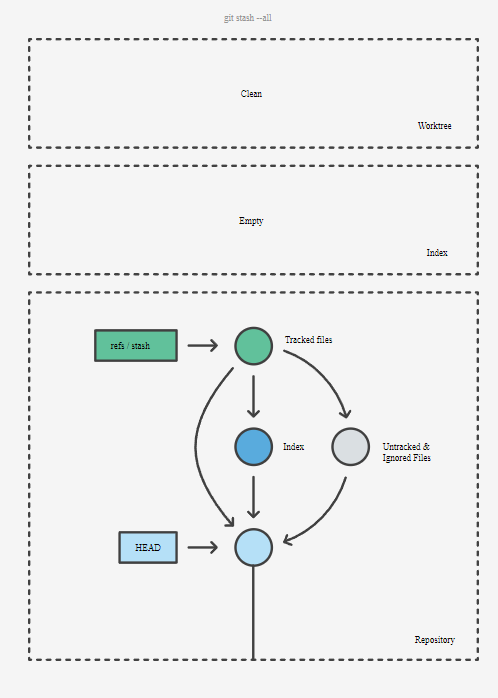
* Invoking git stash encodes any changes to tracked files as two new commits in your DAG: one for unstaged changes, and one for changes staged in the index. The special refs/stash ref is updated to point to them.



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