## 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

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

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

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

$ git config user.email

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

* **--local**

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

* **--global**

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

* **--system**

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

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

### Writing a value

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

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

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

### Writing a value

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

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

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

## git config editor - core.editor

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

## Merge tools

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

git config --global merge.tool kdiff3

## Aliases

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

git config --global alias.ci commit

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

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

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

**Formatting & whitespace**

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

The following features are enabled by default:

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

The following features are disabled by default

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

## Summary

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

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

# 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.

# .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.