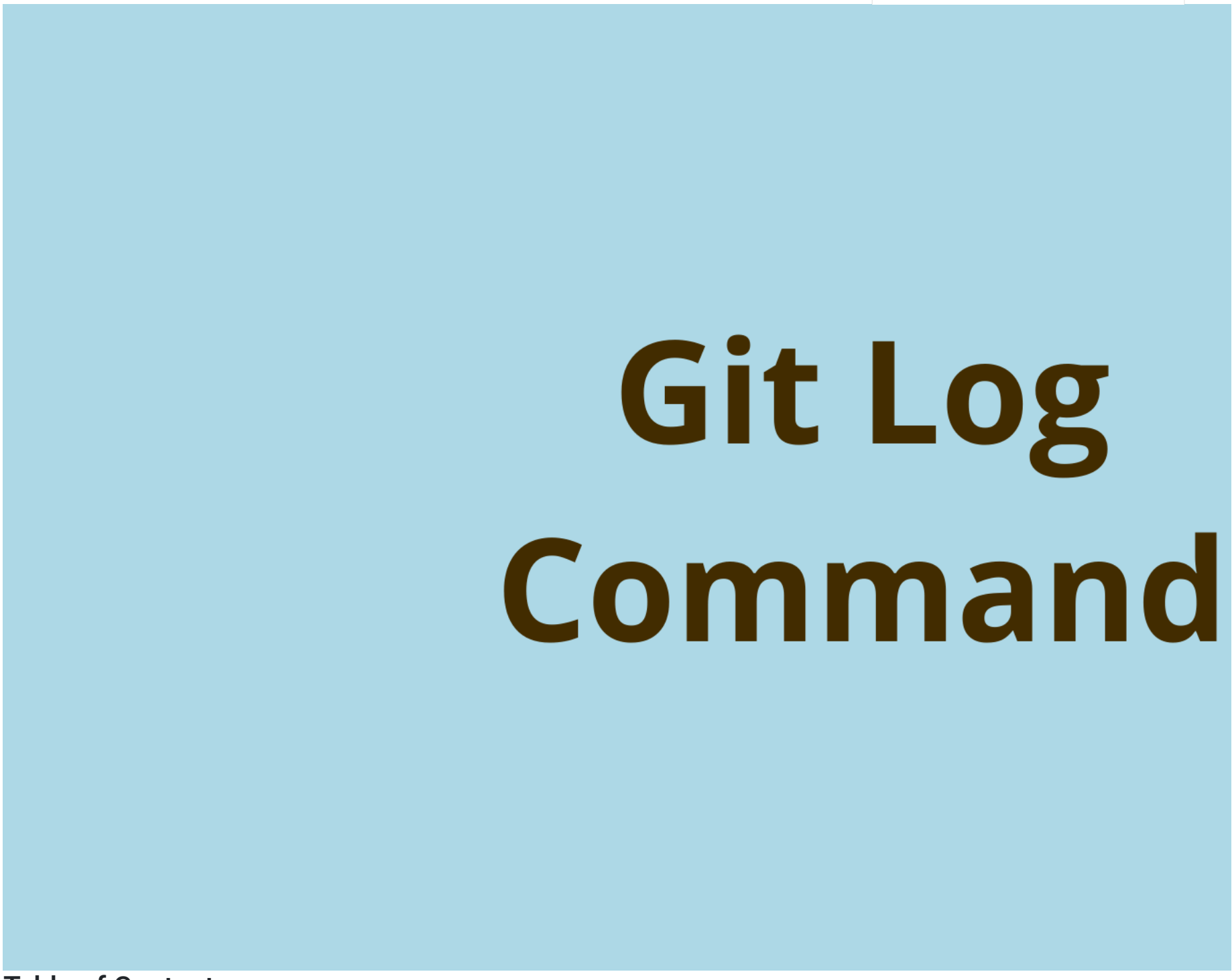
# git log | A Guide to Using the log Command in Git

Art Sphitz May 29, 2022


IC
INITIAL
COMMIT
Guideboo




## Table of Contents



## What is git log?

The `git log` command is used to view the history of committed changes within a Git repository. Each set of changes made by a developer is recorded as a commit in Git.

The git log command shows a default output for quickly reviewing the commit history.

The default output includes the commit ID, commit author, developer email, commit date and commit message string for each commit on the active branch.

Git's active branch is the one that is currently checked out in the working directory.

Git log also comes with a rich set of logging options for filtering, ordering, and formatting output.

# git log Example Default Output

Before diving in, let's assume we have a new Git repository at project root folder `git-log-example/` and look at the git log default output:

```
git-log-example/
    file1.ext
    file2.ext

    dir1/
        dir1file1.ext
```

Next, let's add a few lines of comments to the previously empty file `file1.ext`:

```
file1.ext
// Sample text for git log tutorial.
```

Now we'll add these changes to the Git staging area, and commit them:

```
$ git add file1.ext
$ git commit -m "Added sample text to file1.ext"
[master 85c0354] Added sample text to file1.ext
 1 file changed, 1 insertion(+)
```

So far, we've initialized a Git repo and made our initial commit, made some changes to one of the files, and committed the changes. Let's take a look at the output from running a simple git log command:

```
$ git log
commit 85c035458122ca9f90a56fc2fa167bb61d22580b (HEAD -> master)
Author: Initial Commit LLC <Example@domain.com>
Date:   Mon May 23 10:39:12 2022 -0500

    Added sample text to file1.ext

commit cd918fd09e0014eefbcf9516a6ad99c431315838
Author: Initial Commit LLC <Example@domain.com>
Date:   Mon May 23 10:27:34 2022 -0500

    Initial commit
```

As can be seen from the output, our log has returned both of our commits. As previously mentioned, git log defaults to showing us the commit hash, author and email, date, and the commit message. The git log sort order defaults to reverse chronological order, which essentially shows commits in descending order based on commit datetime. There are exceptions to this ordering rule, which will be seen in future examples.

This is a convenient default, but it's commonly required to filter the commits displayed in the log output. Before moving forward, we need to create some more commits.

Let's add some text to `file2.ext` and `dir1file1.ext` and record them as separate commits, then take a look at how we can accomplish some git log filtering.

# Filter git log

The Git log can be filtered in a variety of ways, such as restricting number of commits displayed, filtering by date range, commit author, commit message content, commit ID range, and more.

Let's take a closer look at some common ways developers filter the Git log.

## Limit Number of Commits in Git Log

Perhaps the most common filtering git log option, the `-[int]` flag allows us to limit the number of commits returned, starting with the most recent commit on your checked-out branch. Here's an example of how we can return the last two commits in the Git history:

```
$ git log -2
commit a7d29889d47dfc5e6c8f91974039f91231269d95 (HEAD -> master)
Author: Initial Commit LLC <Example@domain.com>
Date:   Tue May 24 13:40:55 2022 -0500

    Added text to dir1file1.ext

commit 0a2012fc2461cfcededd2d00d99a4b1d96ce02f8
Author: Initial Commit LLC <Example@domain.com>
Date:   Tue May 24 13:40:15 2022 -0500

    Added text to file2.ext
```

As can be seen from the output, we have successfully returned *only* the last two commits.

Note that "git log -[int]" is actually a convenient shortcut for the "-n" flag, such as "git log -n [int]"

Next we'll look at filtering Git log by date.

## Filter Git Log by Date

We can filter our git log by date using a few different techniques. Here are some of the more common ways this can be accomplished.

1. The `--after` option allows you to view all commits made after the given date:

```
git log --after="yyyy-mm-dd"
```

2. We can include the `--before` option to filter commits within a range of dates:

```
git log --after="yyyy-mm-dd" --before=="yyyy-mm-dd"
```

3. For additional convenience try using verbal time references, such as "1 week ago" or "yesterday":

```
git log --after="1 week ago" --before=="yesterday"
```

## Filter Git Log by Author

Use the `--author` flag to display commits only made by a specific author:

```
git log --author="Smith"
```

This returns only the commits with an author name that includes `Smith`. This option also allows regular expressions if you're interested in refining your author search even more.

The "--author" option includes the author email, which you'll note is displayed on the same line as the author in each Git log entry. So you can use this flag to search based on email as well. This search is case sensitive.

## Git Log Search by Commit Message

The `--grep` flag can be used to filter commits by message, allowing you to filter out a logical group of commits if your project used a specific convention for the commit messages. Usage is as follows:

```
git log --grep="Bugfix"
```

This will display commits in the log with the text "Bugfix" in the commit message.

You can use the `-i` flag with this command to ignore case sensitivity.

## Git Log Commit Range

It is often needed to pull a range of commits for review. This can be accomplished utilizing the `..` operator, along with the ref names for the start commit and end commit in the range.

The ref names can be tags, hashes, or references. Based on our current example project, we could return the second commit all the way to our current head using the following command:

```
git log cd918f..HEAD
```

The commits returned by the range operator are exclusive of the start commit in the range and inclusive of the end commit in the range.

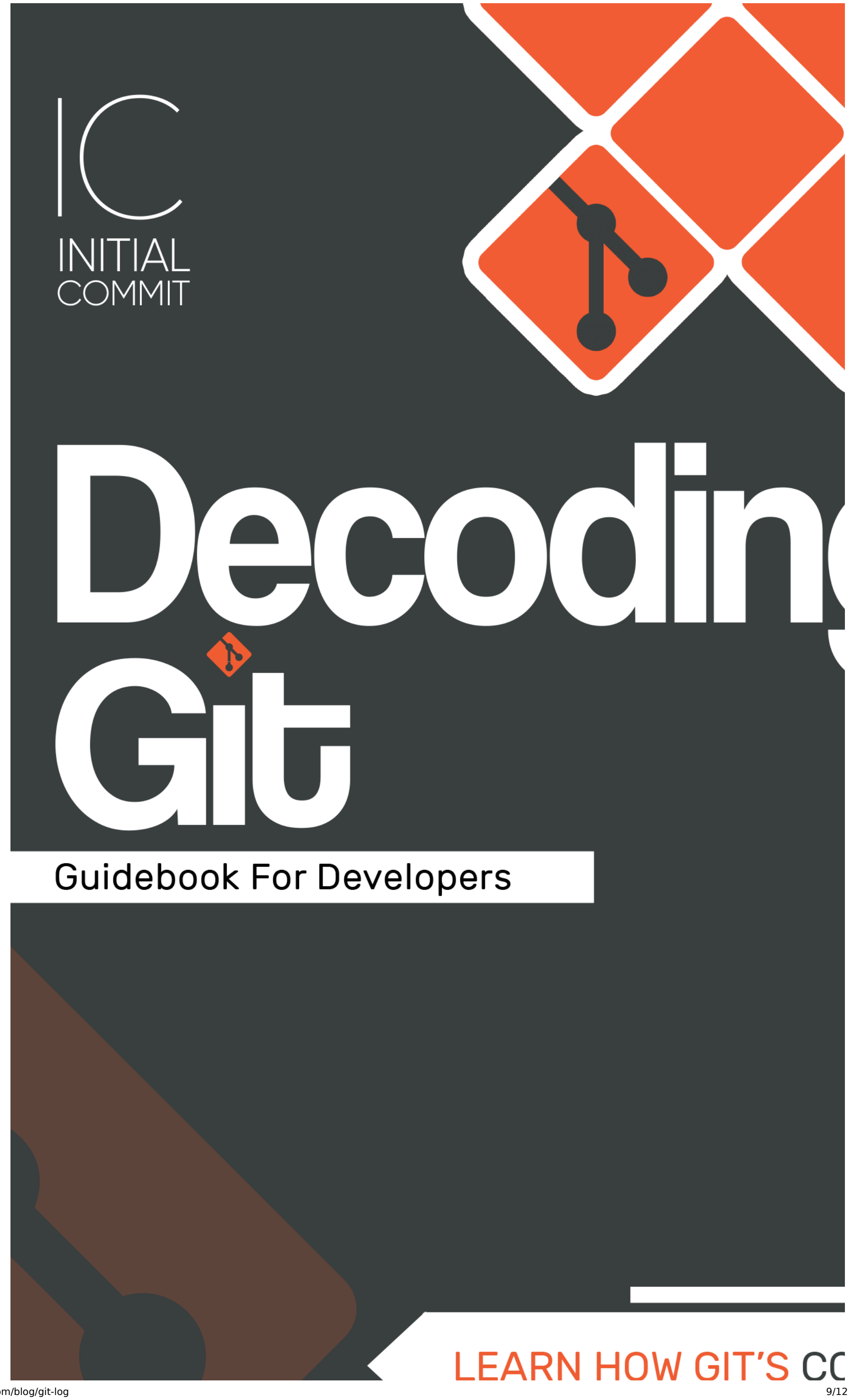
## Git Log Search for a File

It's also a common need to view only the commits that include changes to a specific file, or multiple files. This can be accomplished using the following syntax:

```
git log -- file1.ext file2.ext
```

When it comes to filtering our commit history, git log offers the tools for pretty much every circumstance you can think of.

### Learn how Git's code works... 🚀👨‍💻📚


IC
INITIAL
COMMIT
Decodin
Git
Guidebook For Developers
LEARN HOW GIT'S CO

## git log Formatting Options

Like many other Git commands, `git log` comes with several options for formatting the output. This include condensing the output, listing the filenames of files in each commit, sorting the commits in reverse order, including diffs, and more.

## Condensing Git Log Output to One Line with --oneline

Use the `--oneline` option to condense each commit's log data on a single line:

```
$ git log --oneline
a7d2988 (HEAD -> master) Added text to dir1file1.ext
0a2012f Added text to file2.ext
85c0354 Added sample text to file1.ext
cd918fd Initial commit
```

As the name implies, this option condenses each commit log into one line by truncating the commit hash and showing only the commit message.

## Git Log Include Filenames with --stat

The `--stat` flag is great for viewing the files that were modified in each commit, along with the number of lines added or removed. It also provides a handy summary line that shows the total number of lines and files that were modified.

This command is rather verbose, so let's combine it with our previously mentioned filter that limits our log to that last two commits:

```
$ git log --stat -2
commit a7d29889d47dfc5e6c8f91974039f91231269d95 (HEAD -> master)
Author: Initial Commit LLC <Example@domain.com>
Date:   Tue May 24 13:40:55 2022 -0500

    Added text to dir1file1.ext

 dir1/dir1file1.ext | 1 +
 1 file changed, 1 insertion(+)

commit 0a2012fc2461cfcededd2d00d99a4b1d96ce02f8
Author: Initial Commit LLC <Example@domain.com>
Date:   Tue May 24 13:40:15 2022 -0500

    Added text to file2.ext

 file2.ext | 1 +
 1 file changed, 1 insertion(+)
```

This option is great for quickly determining which files were modified in a given set of commits.

## Git Log Include Diff with --patch or -p

Git is able to construct a textual diff between any two commits, and git log is able to display that too. To view the textual diff in your output, simply include the `-p` or `--patch` flag. Again, we'll combine this with our filtering technique to return only the last commit made:

```
$ git log -p -1
commit a7d29889d47dfc5e6c8f91974039f91231269d95 (HEAD -> master)
Author: Initial Commit LLC <Example@domain.com>
Date:   Tue May 24 13:40:55 2022 -0500

    Added text to dir1file1.ext

diff --git a/dir1/dir1file1.ext b/dir1/dir1file1.ext
index e69de29..4636ea0 100644
--- a/dir1/dir1file1.ext
+++ b/dir1/dir1file1.ext
@@ -0,0 +1 @@
+// Some more text inside dir1file1.ext for git log tutorial.
\ No newline at end of file
```

We can see the text added for this commit in the textual diff given in the output.

## Git Log Reverse Order with --reverse

Sometimes, it can be helpful to reverse the order of a git log output. This is simple with the `--reverse` option, which reverses the default order of the output. This returns the initial commit first, followed by the rest of your commit history in an ascending order:

```
git log --reverse
```

## Git Log Pretty Format with --pretty

If Git's built-in format options don't suit your needs, the `--pretty` flag allows you to customize the log output. Here's a simple example that displays an abbreviated hash, the commit date, and the message body:

```
git log --pretty=format:"%h - %cd - %B"
```

Options for git log –pretty are pretty extensive, so for the sake of brevity, this example will suffice. However, if you'd like an exhaustive list of options, you can head over to the git --pretty docs.

## Visualize Git Log History with --graph

It can be helpful to visualize your Git commit history. This can be accomplished using the `git log --graph` command.

For this example, let's initialize a new Git repo and tailor the commit history to best accommodate this example.

We'll assume we have a feature branch named `bugfix`, make some commits, and merge these commits to the master branch. Note that the most recent common ancestor of two branches is called the merge base. Let's see what that looks like in the logging output, using our `--pretty` flag to clean things up a bit:

```
$ git log --graph --pretty=format:"%h - %cd - %B %d"
* 86b7821 - Wed May 25 09:56:51 2022 -0500 - Fourth commit to master, after merging bugfix.
|   (HEAD -> master)
*   44e3afd - Wed May 25 09:56:11 2022 -0500 - Merge branch 'bugfix'
|\
| * 7c51a70 - Wed May 25 09:54:17 2022 -0500 - Second commit to bugfix branch.
| |   (bugfix)
| * f70b486 - Wed May 25 09:53:42 2022 -0500 - First commit to bugfix branch.
| |
* | 3099a91 - Wed May 25 09:56:05 2022 -0500 - Third commit to master branch, after checking out master but before mergin
| |
* | 0c4c9ab - Wed May 25 09:48:32 2022 -0500 - Third commit to master branch, after creating bugfix but not checking out.
|/
* 9147f56 - Wed May 25 09:47:41 2022 -0500 - Second commit to master branch, before creating bugfix branch.
|
* 9885ea0 - Wed May 25 09:47:04 2022 -0500 - Initial commit.
```

Viewing the output shows us exactly where our history splits into two branches. However, close examination will reveal that we seem to have lost our reverse chronological order.

Take a look at commit hash **3099a91**. Clearly, according to the timestamp, this commit is not in chronological order. That's because when using the `--graph` flag, Git sorting changes from reverse chronological order to a Git sort order called **topological ordering**.

According to the Git docs, topological ordering will "Show no parents before all of its children are shown, and avoid showing commits on multiple lines of history intermixed."

Because the merge hasn't taken place yet at the time of commit **3099a91**, Git considers this commit a child of the merge commit, and will therefore place such commits prior to any commits made within the `bugfix` branch itself.

If we want to force reverse chronological ordering, we can utilize the `--date-order` flag, which may be desirable if you're looking for a strict commit timeline on your project:

```
git log --graph --pretty=format:"%h - %cd - %B %d" --date-order
* 86b7821 - Wed May 25 09:56:51 2022 -0500 - Fourth commit to master, after merging bugfix.
|   (HEAD -> master)
*   44e3afd - Wed May 25 09:56:11 2022 -0500 - Merge branch 'bugfix'
|\
* | 3099a91 - Wed May 25 09:56:05 2022 -0500 - Third commit to master branch, after checking out master but before mergin
| |
| * 7c51a70 - Wed May 25 09:54:17 2022 -0500 - Second commit to bugfix branch.
| |   (bugfix)
| * f70b486 - Wed May 25 09:53:42 2022 -0500 - First commit to bugfix branch.
| |
* | 0c4c9ab - Wed May 25 09:48:32 2022 -0500 - Third commit to master branch, after creating bugfix but not checking out.
|/
* 9147f56 - Wed May 25 09:47:41 2022 -0500 - Second commit to master branch, before creating bugfix branch.
|
* 9885ea0 - Wed May 25 09:47:04 2022 -0500 - Initial commit.
```

As can be seen from the output, commit **3099a91** is not in the proper order according to commit date.

## git log vs git show

The git show command is another tool that can be used to show details of specific Git objects, including commits, trees, and blobs.

As we've seen, git log defaults to showing us a full history of commits, showing only the metadata such as hash and author. In contrast, git show is used for individual objects (not the whole commit history), and defaults to showing the most recent commit with textual diff included.

Here's a basic example of output using git show:

```
$ git show
commit 86b782196279e04b03709dbd58f97fe8d20010b9 (HEAD -> master)
Author: Initial Commit LLC <example@domain.com>
Date:   Wed May 25 09:56:51 2022 -0500

    Fourth commit to master, after merging bugfix.

diff --git a/file2.ext b/file2.ext
index 3695f67..311030c 100644
--- a/file2.ext
+++ b/file2.ext
@@ -2,4 +2,6 @@

 // First commit to bugfix branch.

-// Second commit to bugfix branch.
\ No newline at end of file
+// Second commit to bugfix branch.
+
+// Fourth commit to master, after merging bugfix.
\ No newline at end of file
```

As the output reflects, git show is a tool to consider when you need a closer look at a specific commit or other Git object.

## Summary

We've shown that git log is a useful tool that comes with the Git software package. Git tracks the history of file changes, known as commits, and git log is a tool used to review these commits in detail.

Git's log defaults to a reverse chronological order, but if used with the --graph flag to get a visual representation of your history, Git changes to a topological ordering system. The ordering of the log can be user-defined as well, giving developers the option to view commits in whichever order is needed.

Overall, `git log` is a fundamental tool to become familiar with if you plan on utilizing Git as a tracking mechanism for your project.

## Next Steps

If you're interested in learning more about how Git works under the hood, check out our Baby Git Guidebook for Developers, which dives into Git's code in an accessible way. We wrote it for curious developers to learn how Git works **at the code level**. To do this we documented the first version of Git's code and discuss it in detail.

We hope you enjoyed this post! Feel free to shoot me an email at jacob@initialcommit.io with any questions or comments.

## References

1. Stackoverflow, Git Double Dot Notation - https://stackoverflow.com/questions/462974/what-are-the-differences-between-double-dot-and-triple-dot-in-git-com
2. Git SCM Docs, Git Pretty Formats - https://git-scm.com/docs/pretty-formats

## Final Notes

Recommended product: Decoding Git Guidebook for Developers

### Related Articles

Use Git-Mv To Rename And Move Files

To Use Git-Mv To Rename And Move Files


Back to Blog