03. Git Basics

▼ Initialize/Clone a git repository

▼ Initialize a git repository in an existing directory —

```
# If git is initialized in a folder where there are existing files,
# you should probably begin tracking those files and do an initial commit.
$ cd <directory path>
$ git init
```

▼ Cloning a remote repository to your local machine —

```
# By default, the remote is named, “origin”. When you are working with the remote
# you have to refer to it by the name origin.
$ git clone <url> <optional: preferred name of the cloned repo>
```

```
# To see the remote name
$ git remote
```

```
# To Rename a remote
$ git remote rename <previous name> <new name>
```

```
# To find what URLs are associated with the git repository for pulling (reading)
# and pushing (writing) to the remote
$ git remote -v
```

```
# For adding a new remote address to an already existing repository
$ git remote add <remote name> <url>
```

```
# Removing a remote
$ git remote remove <remote name>
```

```
# To see more information about a particular remote
$ git remote show <remote name>
```

▼ A handy trick: Find all local git repositories on your local machine —

```
$ find $HOME -type d -name ".git"
```

```
# Assuming you have locate, this should be much faster
$ locate .git |grep git
```

```
# If you have gnu locate, mlocate or, plocate you can also do this instead
$ locate -ber \\.git
```

▼ Working with remote repositories

▼ Fetching and Pulling from your remotes —

```
# To get data from your remote projects without merging
$ git fetch <remote name>
```

It's important to note that the `git fetch` command only downloads the data to your local repository — it doesn't automatically merge it with any of your work or modify what you're currently working on. You have to merge it manually into your work when you're ready.

```
# To get data from your remote projects, merged with the current branch
$ git pull <remote name>
```

Note: If you clone a repository, the command automatically adds that remote repository under the name "**origin**". Also note that fetching or pulling from a remote repository only downloads any new work that has been pushed to that server since you cloned (or last fetched/pulled from) it.

▼ Pushing to Your Remotes —

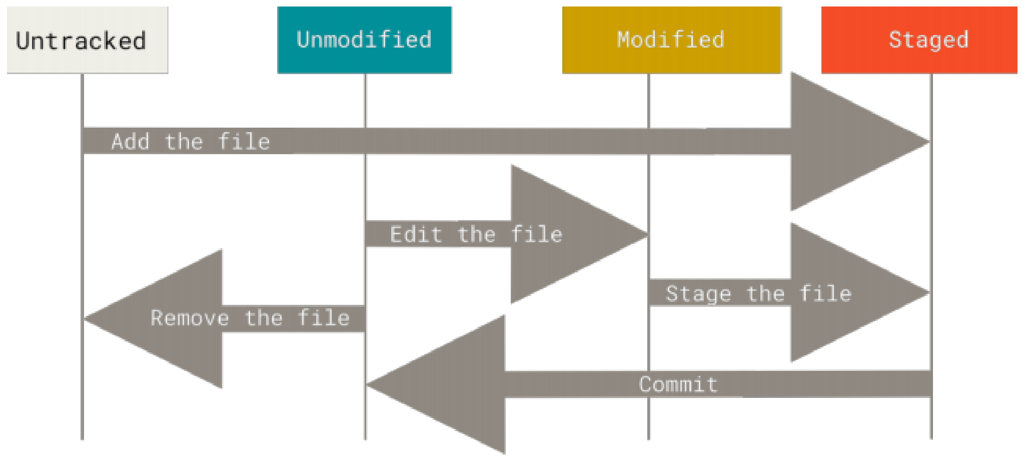
```
$ git push <remote name> <branch name>

# The branch you would usually want to push is “main” or “master”.
```

▼ Tracking Files and Recording Changes

Each file in the working directory can be in one of two states.

1. **Tracked** files are files that were in the last snapshot; they can be **unmodified, modified, or staged**. In short, tracked files are files that Git knows about.

2. **Untracked** files are everything else — any files in your working directory that were not in your last snapshot and are not in your staging area.



▼ Checking the Status of Your Files —

```
$ git status

# for short status
$ git status -s
```

▼ Tracking New Files/ Staging Tracked files —

```
$ git add <file path>

# to add i.e, stage all the files in the current directory
# that have been changed since last commit or are not tracked yet
$ git add .
```

`git add` is a multipurpose command. You use it to, begin tracking new files, to stage files, to mark merge-conflicted files as resolved, etc. It may be helpful to think of it more as “**add precisely this content to the next commit**” rather than “add this file to the project”. If you modify a file after you run git add, you have to run git add again to stage the latest version of the file.

- To see what you’ve changed but not yet staged for the next commit

  ```
  $ git diff
  ```

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

- To see what you've staged that will go into your next commit

```
$ git diff --staged
```

  This command compares your staged changes to your last commit.

- Stop Tracking Files

```
$ git rm --cached <file / directory path / file-glob patterns>
```

  Note that, using only `$ git rm <file name>` will delete the file from your local machine. So be sure to include `--cached` if you just want to stop tracking a file and not delete it.

▼ Ignoring Files —

Often, you'll have a class of files that you don't want Git to automatically add or even show you as being untracked. These are generally automatically generated files, such as log files or files produced by your build system. In such cases, you can create a file listing patterns named `.gitignore` to match those files and folders. Git automatically ignores the patterns listed in the .gitignore file.

If you want a starting point for your project, GitHub maintains a fairly comprehensive list of good .gitignore file examples for dozens of projects and languages at https://github.com/github/gitignore.

▼ Commit your changes —

```
# launch your editor of choice where you can put the commit message
$ git commit
```

```
$ git commit -v
# This puts the diff of your change in the editor so you can see exactly
# what changes you're committing
```

```
$ git commit -m "commit message"
# This way you can write the commit message inline
```

Remember that anything that is still unstaged (any files you have created or modified that you haven't run 'git add' on since you edited them) won't go into this commit. They will stay as modified files on your disk.

```
# To skip the staging area you can use
$ git commit -a -m "commit message"

# Adding the -a option to the git commit command makes Git automatically stage every file
# that is already tracked before doing the commit, letting you skip the git add part
```

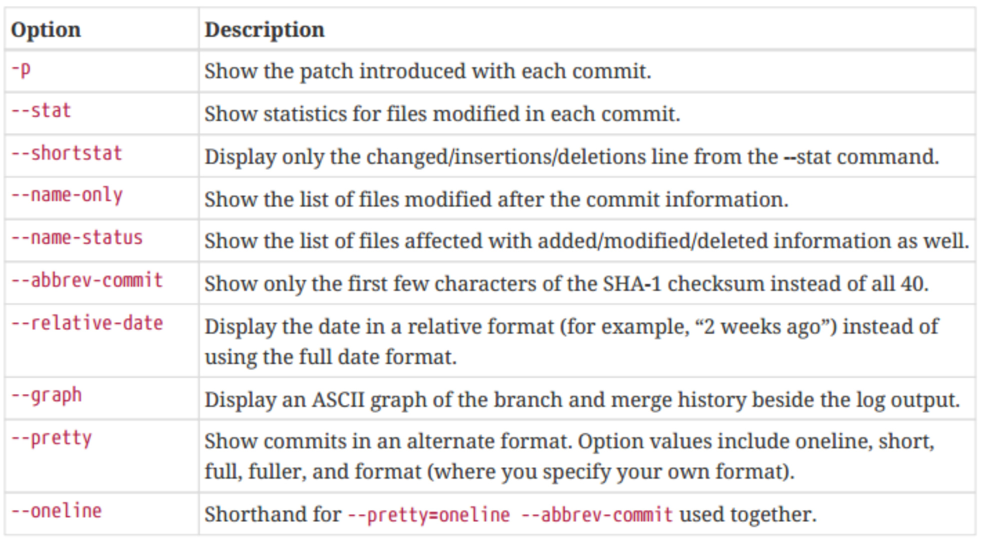
▼ **Git Log: Viewing your commit history —**

The most basic and powerful tool to view the commit history is the `git log` command.

By default, with no arguments, `git log` lists the commits made in that repository in reverse chronological order; that is, the most recent commits show up first. This command lists each commit with its SHA-1 checksum, the author's name and email, the date written, and the commit message. If you have multiple branches then the `git log` command will list the commit history of those branches too.

- Common options to `git log` command —

Table 2. Common options to `git log`



Option	Description
`-p`	Show the patch introduced with each commit.
`--stat`	Show statistics for files modified in each commit.
`--shortstat`	Display only the changed/insertions/deletions line from the --stat command.
`--name-only`	Show the list of files modified after the commit information.
`--name-status`	Show the list of files affected with added/modified/deleted information as well.
`--abbrev-commit`	Show only the first few characters of the SHA-1 checksum instead of all 40.
`--relative-date`	Display the date in a relative format (for example, "2 weeks ago") instead of using the full date format.
`--graph`	Display an ASCII graph of the branch and merge history beside the log output.
`--pretty`	Show commits in an alternate format. Option values include oneline, short, full, fuller, and format (where you specify your own format).
`--oneline`	Shorthand for `--pretty=oneline --abbrev-commit` used together.

Table 1. Useful specifiers for `git log --pretty=format`

Specifier	Description of Output
`%H`	Commit hash
`%h`	Abbreviated commit hash
`%T`	Tree hash
`%t`	Abbreviated tree hash
`%P`	Parent hashes
`%p`	Abbreviated parent hashes
`%an`	Author name
`%ae`	Author email
`%ad`	Author date (format respects the --date=option)
`%ar`	Author date, relative
`%cn`	Committer name
`%ce`	Committer email
`%cd`	Committer date
`%cr`	Committer date, relative
`%s`	Subject

- Limiting the log output —

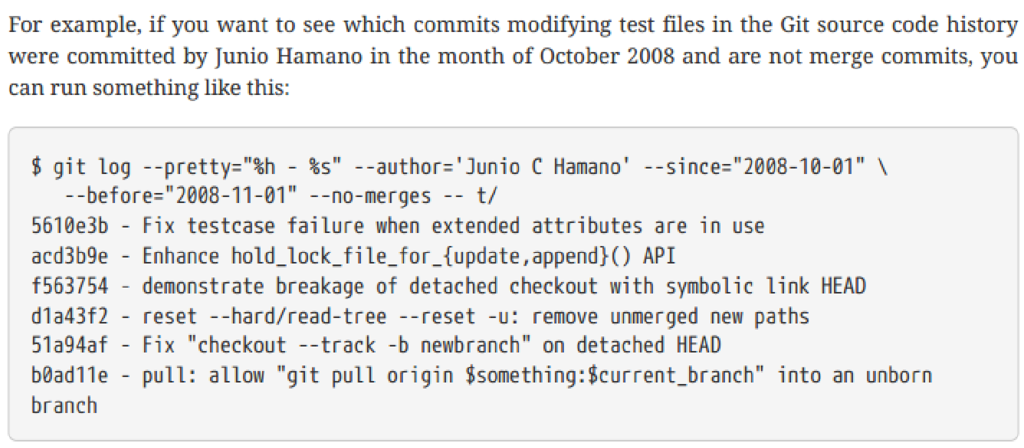
Table 3. Options to limit the output of `git log`

Option	Description
`-<n>`	Show only the last n commits
`--since`, `--after`	Limit the commits to those made after the specified date.
`--until`, `--before`	Limit the commits to those made before the specified date.
`--author`	Only show commits in which the author entry matches the specified string.
`--committer`	Only show commits in which the committer entry matches the specified string.
`--grep`	Only show commits with a commit message containing the string

Option	Description
`-S`	Only show commits adding or removing code matching the string

```
# to limit the log output to commits that introduced a change to a particular file
$ git log --<file path>
```

- ### Usage example



For example, if you want to see which commits modifying test files in the Git source code history were committed by Junio Hamano in the month of October 2008 and are not merge commits, you can run something like this:

```
$ git log --pretty="%h - %s" --author='Junio C Hamano' --since="2008-10-01" \
   --before="2008-11-01" --no-merges -- t/
5610e3b - Fix testcase failure when extended attributes are in use
acd3b9e - Enhance hold_lock_file_for_{update,append}() API
f563754 - demonstrate breakage of detached checkout with symbolic link HEAD
d1a43f2 - reset --hard/read-tree --reset -u: remove unmerged new paths
51a94af - Fix "checkout --track -b newbranch" on detached HEAD
b0ad11e - pull: allow "git pull origin $something:$current_branch" into an unborn
branch
```

Undoing things —

- Amend your latest commit

```
# when you commit too early and possibly forget to add some files, or you mess up your
# commit message, or if you want to redo that commit, make the additional changes you forgot,
# stage them, and commit again (without cluttering your repository history with
# commit messages of the form, “Oops, forgot to add a file” or “Darn, fixing a typo
$ git commit --amend
```

It's important to understand that **when you're amending your last commit, you're not so much fixing it as replacing it entirely with a new, improved commit** that pushes the old commit out of the way and puts the new commit in its place.

- Unstaging a Staged File

To check-out a particular commit without deleting your local changes you can use `git reset HEAD`

How do I undo the most recent local commits in Git?

I accidentally committed the wrong files to Git, but didn't push the commit to the server yet.
How do I undo those commits from the local repository?

https://stackoverflow.com/questions/927358/how-do-i-undo-the-most-recent-local-commits-in-git

```
$ git reset HEAD <file path>
```

- Unmodifying a Modified File

```
# to checkout a file at the last commit
$ git checkout --<file path>

# Understand that this is a dangerous command. Any local changes you made to that
# file are gone. Git just replaced that file with the last staged or committed version.
# Don’t ever use this command unless you absolutely know that you don’t want
# those unsaved local changes.

# to checkout a file at a particular commit
$ git checkout <checksum string> --<file path>
```

▼ Renaming or moving a file —

```
# Renaming a file
$ git mv <current filename> <new filename>
```

```
# Moving a file
$ git mv <source directory> <destination directory>
```

Keep in mind that, by design, git does not track moves at all.

```
# To see the history through renames try using
$ git log --follow <file name>
```