**Learn Enough Git to Be Dangerous Michael Hartl**

An Introduction to Version Control with Git

[**Chapter 1 Getting started**](https://www.learnenough.com/git-tutorial?single_page=1#cha-getting_started)

A version control system, or *VCS*, provides an automatic way to track changes in software projects, giving creators the power to view previous versions of files and directories, develop speculative features without disrupting the main development, securely back up the project and its history, and collaborate easily and conveniently with others. In addition, using version control also makes deploying production websites and web applications much easier.

Version control has evolved considerably over the years. The family line leading to Git includes programs called RCS, CVS, and Subversion, and there are many current alternatives as well, including Perforce, Bazaar, and Mercurial. I mention these examples not because you need to know what they are, but only to show what a bewildering variety there is. What’s worse, when you choose a version control system, you really *commit* to it,[1](https://www.learnenough.com/git-tutorial?single_page=1" \l "cha-1_footnote-1) and it is often difficult to switch from one to another. Happily, in the last few years an undisputed winner has emerged in the open-source VCS wars: Git.

Git has a combination of power, speed, and community adoption that leave it few rivals, but it can be tricky to learn, and other Git tutorials have a tendency to introduce lots of heavy theory, which can be interesting to learn but in practice is really only understood by a tiny handful of Git users (as illustrated in “[Git](https://m.xkcd.com/1597/)” via the webcomic [xkcd](https://xkcd.com/)).

[**1.1 Installation and setup**](https://www.learnenough.com/git-tutorial?single_page=1#sec-installation_and_setup)

After installing Git but before starting a project, we need to perform a couple of one-time setup steps, as shown in [Listing 1.1](https://www.learnenough.com/git-tutorial?single_page=1#code-global_config). These are *global* setups, meaning you only have to do them once per computer. (Don’t worry about the meaning or structure of these commands at this stage.)

Listing 1.1: One-time global configuration settings.

$ git config --global user.name "Your Name"

$ git config --global user.email your.email@example.com

These configuration settings allow Git to identify your changes by name and email address, which is especially helpful when collaborating with others ([Chapter 4](https://www.learnenough.com/git-tutorial?single_page=1#cha-collaborating)). Note that the name and email you use in [Listing 1.1](https://www.learnenough.com/git-tutorial?single_page=1#code-global_config) will be viewable in any projects you make public, so don’t expose any information you’d rather keep private.

Git stores global configuration settings in a hidden text file located in your home directory. By inspecting the file ~/.gitconfig with a tool of your choice (cat, less, a text editor, etc.), confirm that the configuration set up by [Listing 1.1](https://www.learnenough.com/git-tutorial?single_page=1#code-global_config) corresponds to simple text entries in this file.

[**1.2 Initializing the repo**](https://www.learnenough.com/git-tutorial?single_page=1#sec-initializing_the_repo)

We’ll begin by making a directory with the generic name website inside a repositories directory called repos:

[~]$ mkdir -p repos/website

Here the -p option, arranges for mkdir to create intermediate directories as required (in this case, repos). After making the directory, we can cd into it as follows:

[~]$ cd repos/website/

[website]$

Even though the website directory is empty, we can already convert it to a *repository*, which you can think of as a sort of enhanced directory with the additional ability to track changes to every file and subdirectory. The way to create a new repository with Git is with the init command (short for “initialize”), which creates a special hidden directory called .git where Git stores the information it needs to track our project’s changes. (It’s the presence of a properly configured .git directory that distinguishes a Git repository from a regular directory.)

All Git commands consist of the command-line program git followed by the name of the command, so the full command to initialize a repository is git init, as shown in [Listing 1.2](https://www.learnenough.com/git-tutorial?single_page=1#code-git_init).

Listing 1.2: Initializing a Git repository.

[website]$ git init

Initialized empty Git repository in /Users/mhartl/repos/website/.git/

[website (master)]$

[**1.3 Our first commit**](https://www.learnenough.com/git-tutorial?single_page=1#sec-our_first_commit)

Git won’t let us complete the initialization of the repository while it’s empty, so we need to make a change to the current directory. We’ll simply use touch to create an empty file.

[website (master)]$ touch index.html

Having created this first file, we can use the git status command to see the result:

We see here that the index.html file is “untracked”, which means Git doesn’t yet know about it. We can *add* it using the git add command:

[website (master)]$ git add -A

Here the -A option tells Git to add *all* untracked files, even though in this case there’s only one. In my experience, 99% of the time you add files you’ll want to add them all, so this is a good habit to cultivate, and learning how to add individual files is left as an exercise ([Section 1.3.1](https://www.learnenough.com/git-tutorial?single_page=1#sec-exercises_our_first_commit)). (By the way, the equivalent command git add ., where the dot refers to the [current directory](https://www.learnenough.com/r/learn_enough_command_line/directories/navigating_directories#sec-navigating_directories), is also common.)

As implied by the word “unstage”, the status of the file has been promoted from *untracked* to *staged*, which means the file is ready to be added to the repository. *Untracked/unstaged* and *staged* are two of the four states commonly used by Git, as shown in [Figure 1.1](https://www.learnenough.com/git-tutorial?single_page=1#fig-git_status_sequence). ([Technically](https://m.xkcd.com/1475/), untracked and unstaged are different states, but the distinction is rarely important because git add tracks and stages files at the same time.)

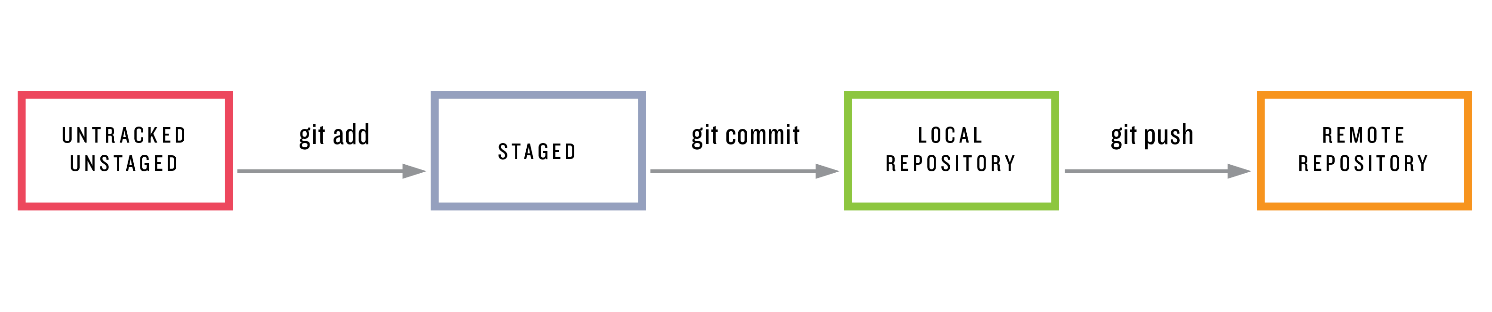


Figure 1.1: The main Git status sequence for a changing file.

After putting changes in the staging area we can make them part of the local repository by *committing* them using git commit. (We’ll cover the final step from [Figure 1.1](https://www.learnenough.com/git-tutorial?single_page=1#fig-git_status_sequence), git push, in [Section 2.3](https://www.learnenough.com/git-tutorial?single_page=1#sec-adding_a_readme).)

At this point, we can use git log to see a record of our commit:

The commit is identified by a *hash*, which is a unique string of letters and numbers that Git uses to label the commit and which lets Git retrieve the commit’s changes. In my case, the hash appears as

879392a6bd8dd505f21876869de99d73f40299cc

but since each hash is unique your result will differ. The hash is often referred to as a “SHA” (pronounced *shah*) because of the acronym for the [Secure Hash Algorithm](https://en.wikipedia.org/wiki/SHA-1) used to generate it.

[**1.4 Viewing the diff**](https://www.learnenough.com/git-tutorial?single_page=1#sec-viewing_the_diff)

It’s often useful to be able to view the changes represented by a potential commit before making it. To see how this works, let’s add a little bit of content to index.html by [redirecting the output](https://www.learnenough.com/r/learn_enough_command_line/manipulating_files/redirecting_and_appending#sec-redirecting_and_appending) of echo to make a “hello, world” page:

[website (master)]$ echo "hello, world" > index.html

The Unix diff utility lets us compare two files foo and bar by typing

$ diff foo bar

Git has a similar function, git diff, which by default just shows the difference between the last commit and unstaged changes in the current project:

We can commit this change by passing the -a option (for “all”) to git commit, which arranges to commit all the changes in currently existing files ([Listing 1.3](https://www.learnenough.com/git-tutorial?single_page=1#code-git_commit_a_m)).

Note that the -a option includes changes only to files already added to the repository, so when there are new files it’s important to run git add -A as in [Section 1.3](https://www.learnenough.com/git-tutorial?single_page=1#sec-our_first_commit) to make sure they’re added properly. It’s easy to get in the habit of running git commit -a and forget to add new files explicitly; learning how to deal with this situation is left as an exercise ([Section 1.4.1](https://www.learnenough.com/git-tutorial?single_page=1#sec-exercises_viewing_the_diff)).

Having added and committed the changes, there’s now no diff:

[website (master)]$ git diff

[website (master)]$

(In fact, simply adding the changes is sufficient; running git add -A would also lead to there being no diff. To see the difference between staged changes and the previous version of the repo, use git diff --staged.)

We can confirm that the change went through by running git log:

Important commands from this section are summarized in [Table 1.1](https://www.learnenough.com/git-tutorial?single_page=1#table-getting_started).

|  |  |  |
| --- | --- | --- |
| **Command** | **Description** | **Example** |
| git help | Get help on a command | $ git help push |
| git config | Configure Git | $ git config --global … |
| source <file> | Activate Bash changes | $ source ~/.bash\_profile |
| mkdir -p | Make intermediate directories as necessary | $ mkdir -p repos/website |
| git status | Show the status of the repository | $ git status |
| touch <name> | Create empty file | $ touch foo |
| git add -A | Add all files or directories to staging area | $ git add -A |
| git add <name> | Add given file or directory to staging area | $ git add foo |
| git commit -m | Commit staged changes with a message | $ git commit -m "Add thing" |
| git commit -am | Stage and commit changes with a message | $ git commit -am "Add thing" |
| git diff | Show diffs between commits, branches, etc. | $ git diff |
| git commit --amend | Amend the last commit | $ git commit --amend |
| git show <SHA> | Show diff vs. the SHA | $ git show fb738e… |

Table 1.1: Important commands from [Chapter 1](https://www.learnenough.com/git-tutorial?single_page=1#cha-getting_started).

[**Chapter 2 Backing up and sharing**](https://www.learnenough.com/git-tutorial?single_page=1#cha-sharing)

If you don’t already have a GitHub account, you can get started by visiting the [GitHub signup page](https://github.com/join) ([Figure 2.1](https://www.learnenough.com/git-tutorial?single_page=1#fig-join_github)) and following the instructions. Use your technical sophistication ([Box 1.2](https://www.learnenough.com/git-tutorial?single_page=1#aside-technical_sophistication)) if you get stuck.

[**2.2 Remote repo**](https://www.learnenough.com/git-tutorial?single_page=1#sec-remote_repo)

Important commands from this section are summarized in [Table 2.1](https://www.learnenough.com/git-tutorial?single_page=1#table-sharing).

|  |  |  |
| --- | --- | --- |
| **Command** | **Description** | **Example** |
| git remote add | Add remote repo | $ git remote add origin |
| git push -u <loc> <br> | Push branch to remote | $ git push -u origin master |
| git push | Push to default remote | $ git push |

Table 2.1: Important commands from [Chapter 2](https://www.learnenough.com/git-tutorial?single_page=1#cha-sharing).

[**Chapter 3 Intermediate workflow**](https://www.learnenough.com/git-tutorial?single_page=1#cha-intermediate_workflow)

[**3.2 Ignoring files**](https://www.learnenough.com/git-tutorial?single_page=1#sec-ignoring_files)

A frequent issue when dealing with Git repositories is coming across files you *don’t* want to commit. These include files containing secret credentials, configuration files that aren’t shared across computers, temporary files, log files, etc.

In order to avoid this annoyance, Git lets us *ignore* such files using a special hidden configuration file called .gitignore. To ignore .DS\_Store, create a file called .gitignore using your favorite text editor and then fill it with the contents shown in [Listing 3.2](https://www.learnenough.com/git-tutorial?single_page=1#code-gitignore_ds_store).

Listing 3.2: Configuring Git to ignore a file. ~/repos/website/.gitignore

.DS\_Store

This is an excellent start, but it would be inconvenient if we had to add the name of every file we want to ignore. For instance, the Vim text editor ([covered briefly](https://www.learnenough.com/r/learn_enough_text_editor/vim/mvv#sec-mvv) in [*Learn Enough Command Line to Be Dangerous*](https://www.learnenough.com/command-line)) sometimes creates *temporary files* whose names involve appending a tilde ~ to the end of the normal filename, so you might be editing a file called foo and end up with a file called foo~ in your directory. In such a case, we would want to ignore *all* files ending in a tilde. To support this case, the .gitignore file also lets us use *wildcards*, where the asterisk \* represents “anything”:[3](https://www.learnenough.com/git-tutorial?single_page=1#cha-3_footnote-3)

\*~

Adding the line above to .gitignore would cause all temporary Vim files to be ignored by Git. We can also add directories to .gitignore, so that, e.g.,

tmp/

would arrange to ignore all files in the tmp/ directory.

[**3.3 Branching and merging**](https://www.learnenough.com/git-tutorial?single_page=1#sec-branching_and_merging)

One of the most powerful features of Git is its ability to make *branches*, which are effectively complete self-contained copies of the project source, together with the ability to *merge* one branch into another, thereby incorporating the changes into the original branch. The best thing about a branch is that you can make your changes to the project in isolation from the master copy of the code, and then merge your changes in only when they’re done. This is especially helpful when collaborating with other users ([Chapter 4](https://www.learnenough.com/git-tutorial?single_page=1#cha-collaborating)); having a separate branch lets you make changes independently from other developers, reducing the risk of accidental conflicts.

We’ll use the addition of a second HTML page, an “About page”, as an example of how to use Git branches. Our first step is to use git checkout with the -b option, which makes a new branch called about-page and checks it out at the same time, as shown in [Listing 3.3](https://www.learnenough.com/git-tutorial?single_page=1#code-checkout_about_page).

We can view the current branches using the git branch command:

[website (about-page)]$ git branch

\* about-page

master

Because about.html is a new file, we have to add it and then commit, and I sometimes like to combine these two steps using && as [described](https://www.learnenough.com/r/learn_enough_command_line/directories/navigating_directories#aside-combining_commands) in [*Learn Enough Command Line to Be Dangerous*](https://www.learnenough.com/command-line):

[website (about-page)]$ git add -A && git commit -m "Add About page"

</html>

We can get a handle on which changes we’ll be merging in by using git diff; we saw in [Section 1.4](https://www.learnenough.com/git-tutorial?single_page=1#sec-viewing_the_diff) that this command can be used by itself to see the difference between unstaged changes and our last commit, but the same command can be used to show diffs between branches. This can take the form git diff branch-1 branch-2, but if you leave the branch unspecified Git automatically diffs against the current branch. This means we can diff about-page vs. master as follows:

[website (about-page)]$ git diff master

There is a possibility that changes on master would *conflict* with the merged changes, but Git is good at automatically merging content. Even when conflict is unavoidable, Git is good at marking conflicts explicitly so that we can resolve them by hand. We’ll see a concrete example of this in [Section 4.2](https://www.learnenough.com/git-tutorial?single_page=1#sec-merge_conflicts).

[**3.4 Recovering from errors**](https://www.learnenough.com/git-tutorial?single_page=1#sec-recovering_from_errors)

[**3.4.1 Exercises**](https://www.learnenough.com/git-tutorial?single_page=1#sec-exercises_recovering_from_errors)

1. The git checkout -f trick works only with files that are staged for commit or are already part of the repository, but sometimes you want to get rid of new files as well. Using touch, create a file with a name of your choice, then git add it. Verify that running git checkout -f gets rid of it.

[**3.5 Summary**](https://www.learnenough.com/git-tutorial?single_page=1#sec-summary_intermediate_workflow)

Important commands from this section are summarized in [Table 3.1](https://www.learnenough.com/git-tutorial?single_page=1#table-intermediate_workflow).

|  |  |  |
| --- | --- | --- |
| **Command** | **Description** | **Example** |
| .gitignore | Tell Git which things to ignore | $ echo .DS\_store >> .gitignore |
| git checkout <br> | Check out a branch | $ git checkout master |
| git checkout -b <br> | Check out & create a branch | $ git checkout -b about-page |
| git branch | Display local branches | $ git branch |
| git merge <br> | Merge in a branch | $ git merge about-page |
| git rebase | Do something possibly weird & confusing | See “[Git Commit](https://m.xkcd.com/1296/)” |
| git branch -d <br> | Delete branch (if merged) | $ git branch -d about-page |
| git branch -D <br> | Delete branch (even if unmerged) **(dangerous)** | $ git branch -D other-branch |
| git checkout -f | Force checkout, discarding changes **(dangerous)** | $ git add -A && git checkout -f |

Table 3.1: Important commands from [Chapter 3](https://www.learnenough.com/git-tutorial?single_page=1#cha-intermediate_workflow).

[**Chapter 4 Collaborating**](https://www.learnenough.com/git-tutorial?single_page=1#cha-collaborating)

[**4.1 Clone, push, pull**](https://www.learnenough.com/git-tutorial?single_page=1#sec-clone_push_pull)

[**4.2 Pulling and merge conflicts**](https://www.learnenough.com/git-tutorial?single_page=1#sec-merge_conflicts)

[**4.2.1 Non-conflicting changes**](https://www.learnenough.com/git-tutorial?single_page=1#sec-non_conflicting_changes)

[**4.2.2 Conflicting changes**](https://www.learnenough.com/git-tutorial?single_page=1#sec-conflicting_changes)

Even though Git’s merge algorithms can often figure out how to combine changes from different collaborators, sometimes there’s no avoiding a conflict.

[**4.3 Pushing branches**](https://www.learnenough.com/git-tutorial?single_page=1#sec-pushing_branches)

To start work on fix-trademark on his local copy, Bob just needs to check it out. By using the same name (i.e., fix-trademark), he arranges for it to be associated with the upstream branch on GitHub, which means that git push will automatically push up his changes:

[**4.5 Summary**](https://www.learnenough.com/git-tutorial?single_page=1#sec-summary_collaborating)

Important commands from this section are summarized in [Table 4.1](https://www.learnenough.com/git-tutorial?single_page=1#table-collaborating).

|  |  |  |
| --- | --- | --- |
| **Command** | **Description** | **Example** |
| git clone <URL> | Copy repo (incl. full history) to local disk | $ git clone https://ex.co/repo.git |
| git pull | Pull in changes from remote repository | $ git pull |
| git branch -a | List all branches | $ git branch -a |
| git checkout <br> | Check out remote branch and configure for push | $ git checkout fix-trademark |

Table 4.1: Important commands from [Chapter 4](https://www.learnenough.com/git-tutorial?single_page=1#cha-collaborating).