

way of the hax0r  
n00b to console punk

## Chapter 14

# Keep Your Work

You have written scripts, startup files, and notes. Right now they are protected by nothing: an editor slip, a wrong `rm`, or a bad idea at midnight replaces the good version with no way back. **Version control** records each version you choose to keep, shows you what changed between them, and gets an old one back. Git is the version control system almost everyone uses, and its basics fit in one chapter.

The concepts in this chapter are:

- a **repository** and the **commits** that make up its history;
- the **working tree**, the **staging area**, and what `status` reports about them;
- `diff`: seeing what changed before you commit it;
- `add` and `commit`: choosing what to record and describing it;
- `log` and `show`: reading history;
- **restoring** a file from the last commit or an older one;
- **ignore rules**, and keeping passwords and keys out of the repository.

Git is a separate program. Linux distributions and Homebrew install it with a package manager, and Git for Windows installs it with Git Bash; `git --version` confirms it is on `PATH`. The commands are identical in every shell, because `git` is an external program in all of them.

## Repositories and Commits

A **repository** is a directory that Git tracks, plus a hidden `.git` directory inside it where Git keeps the history. A **commit** is a snapshot of the tracked files at one moment, with a message saying why, the author, and the time. Commits form a chain, each pointing at the one before, and that chain is the history.

Nothing is recorded until you ask. Editing a file changes the **working tree**, the ordinary files on disk. `git add` copies a file's current state into the **staging area**, also called the index, which is the set of changes the next commit will contain. `git commit` turns the staging area into a commit. The staging step is what lets you commit one fix while leaving unrelated edits for later.

Tell Git who you are once, so that commits carry your name:

```
$ git config --global user.name 'Student Name'
$ git config --global user.email 'student@example.com'
```

--global writes to ~/.gitconfig, which applies to every repository on this account. [1]

## Start a Repository

Turn the practice directory into a repository:

```
$ cd ~/hax0r-practice
$ git init
Initialized empty Git repository in /home/student/hax0r-practice/.git/
$ git status
On branch main
No commits yet
```

git init creates the .git directory and nothing else. git status is the command you will run most: it reports which files are changed, staged, or unknown to Git. If your Git names the first branch master, git config --global init.defaultBranch main changes the default for new repositories; the name does not matter for this chapter. [2], [3]

## Record a Change

Create a file, then look at what Git says:

```
$ printf 'hello\n' > notes.txt
$ git status --short
?? notes.txt
```

?? marks an **untracked** file: Git sees it and is not recording it. Stage it and commit:

```
$ git add notes.txt
$ git status --short
A notes.txt
$ git commit -m 'Add notes'
[main (root-commit) 5177c52] Add notes
1 file changed, 1 insertion(+)
```

A means added to the staging area. The commit message after -m should say what the change does and, when it is not obvious, why. git commit without -m opens the editor from Chapter 5, which is where --wait and EDITOR matter. [4], [5]

Now change the file and follow the same path, with one stop to look at the difference:

```
$ printf 'hello\nworld\n' > notes.txt
$ git status --short
M notes.txt
$ git diff
```

```
diff --git a/notes.txt b/notes.txt
--- a/notes.txt
+++ b/notes.txt
@@ -1 +1,2 @@
  hello
+world
```

M in the second column means modified in the working tree and not staged. `git diff` shows the change as lines removed with `-` and added with `+`, and reading it before staging is the habit that catches stray edits. After `git add`, `git diff` shows nothing, because it compares the working tree with the staging area; `git diff --staged` shows what the next commit will contain. [6]

```
$ git add notes.txt
$ git commit -m 'Add a second line'
```

`git add .` stages everything under the current directory, and `git commit -a` stages every modified tracked file and commits in one step. Both are convenient and both are how unrelated changes and unwanted files end up in a commit; run `git status` first.

## Read the History

```
$ git log
$ git log --oneline
1553927 Add a second line
5177c52 Add notes
$ git log --oneline --stat
$ git show HEAD
```

`git log` lists commits, newest first, with their hash, author, date, and message. `--oneline` compresses each to one line, and `--stat` adds which files changed. `git show` prints one commit with its diff; `HEAD` is the current commit, `HEAD~1` the one before, and any hash prefix names a specific one. `git show HEAD~1:notes.txt` prints the file as it was in that commit, without touching the working tree. [7], [8]

## Get an Old Version Back

Restoration is why you kept the history.

```
$ printf 'oops\n' > notes.txt
$ git restore notes.txt
$ cat notes.txt
hello
world
```

`git restore file` replaces the working-tree file with the version in the staging area, or the last commit if nothing is staged. The bad edit is gone, so be sure that is what you want; the command does not ask. [9]

Two related situations:

- **Staged by mistake.** `git restore --staged file` removes the file from the staging area and leaves the working tree alone.
- **Need a version from several commits ago.** `git restore --source HEAD~2 file` takes that commit's version.

Git also has `git checkout` for the same jobs, which older instructions use. `git restore` was added to make the two meanings of checkout separate; prefer it in new work. [9]

Commits themselves are not deleted by any command in this chapter. Even a commit you regret stays in the history, and that is the property the next section depends on.

## Ignore What Should Not Be Tracked

Some files belong in the directory and not in the history: build output, editor backups, downloaded dependencies, and anything containing a secret. A `.gitignore` file lists patterns for Git to skip:

```
build/  
*.log  
.env  
.DS_Store
```

A pattern ending in `/` matches a directory, `*` works as in a glob, and a leading `!` re-cludes something an earlier pattern excluded. Commit `.gitignore` itself, so that everyone who clones the repository ignores the same things. `git status` no longer lists ignored files, and `git add .` skips them. [10]

Ignoring stops Git from adding a file. It does not remove a file that is already tracked; for that, `git rm --cached file` stops tracking it while leaving it on disk.

## Keep Secrets Out

A **secret** is a password, an API token, an SSH private key, or anything else that grants access. Once committed, a secret is in the history, and the history is what gets copied when a repository is shared or pushed. Removing the file in a later commit does not help:

```
$ git rm --cached secret.txt  
$ git commit -m 'Stop tracking secret'  
$ git show HEAD~1:secret.txt  
TOKEN=abc
```

The token is still there for anyone with the repository. Treat a committed secret as leaked: change the token, then clean the history if the repository was shared, using tools such as `git filter-repo`, which are outside this chapter. [11]

Prevention is cheaper:

- Put secrets in a file such as `.env` and list it in `.gitignore` before creating it.
- Read `git status` and `git diff --staged` before every commit, especially after `git add ..`
- Keep keys in `~/.ssh` and configuration in `~/.config`, outside any repository.

- Consider a pre-commit scanner such as `gitLeaks`, which refuses commits containing things that look like credentials.



**Trap:** `git add .` inside your home directory, or in a directory containing `.ssh`, `.env`, or a browser profile, stages everything there. Initialize repositories in project directories, never in your home directory itself. If you did, `rm -r .git` in the home directory removes the repository without touching the files.

## Key Points

- A repository holds commits, each a snapshot with a message; `git init` starts one and `git status` reports its state.
- Changes move from the working tree to the staging area with `git add`, and into history with `git commit -m`.
- `git diff` shows unstaged changes, `git diff --staged` shows what the next commit contains, and `git log --oneline` lists history.
- `git restore file` discards working-tree changes, `git restore --staged file` unstages, and `--source` fetches an older version.
- `.gitignore` keeps build output and secrets from being added; `git rm --cached` stops tracking a file already added.
- A committed secret is in the history until the history is rewritten; rotate it and prevent the next one with ignore rules and a look at `git diff --staged`.

## Level Up

n00bs finish the chapter here. You can record versions of your work, see what changed, and get an old version back. Hackers use history as a tool, not just a backup.

Learn **branches**: `git switch -c experiment` starts one, and `git merge` brings it back. Explain what a branch is in terms of commits pointing at commits, and try `git log --graph --oneline --all` after a merge.

Put your repository on a **remote**: GitHub, GitLab, or a bare repository on the lab machine from Chapter 12 over SSH. Learn `git remote`, `git push`, `git pull`, and `git clone`, and explain what origin is.

Investigate `git stash`, `git bisect`, and `git blame`, and describe a situation where each one saved you time. Then read about `git reflog` and explain why a commit is hard to lose even after a mistake.

Look at how other people write commit messages: read the log of a project you use. Read about **Conventional Commits** and decide whether the structure helps.

Start the dotfiles repository suggested in Chapter 10, and use `git diff` the next time a shell setting breaks.

## Exercises

1. **What does this do?** Predict the output of `git status --short` after creating a file, then after `git add`, then after editing it again without adding, and check each step.
2. **What does this do?** Predict what `git diff` and `git diff --staged` each show after `git add notes.txt` followed by one more edit to `notes.txt`.
3. **What is the problem?** A teammate's `.gitignore` lists `*.log`, but `git status` still shows `debug.log` as modified. Explain why and give the command that fixes it.
4. **What is the problem?** A commit contains a `.env` file with an API key, and the author deleted it in the next commit. Explain why the key must still be changed.
5. **Why is it like this?** Git has a staging area between editing and committing, when many tools just save. Explain one thing the staging area makes possible.
6. **Why is it like this?** `git restore notes.txt` does not ask for confirmation, while `rm -i` does. Explain what Git assumes about the version it restores from.
7. **How do you** see the version of a file from three commits ago without changing your working tree?
8. **How do you** set up a new project so that build output and secrets are ignored before the first commit?

# References

- [1] Git Project, “Git-config: Get and set repository or global options.” Available: <https://git-scm.com/docs/git-config>
- [2] Git Project, “Git-init: Create an empty git repository.” Available: <https://git-scm.com/docs/git-init>
- [3] Git Project, “Git-status: Show the working tree status.” Available: <https://git-scm.com/docs/git-status>
- [4] Git Project, “Git-add: Add file contents to the index.” Available: <https://git-scm.com/docs/git-add>
- [5] Git Project, “Git-commit: Record changes to the repository.” Available: <https://git-scm.com/docs/git-commit>
- [6] Git Project, “Git-diff: Show changes between commits, commit and working tree, etc.” Available: <https://git-scm.com/docs/git-diff>
- [7] Git Project, “Git-log: Show commit logs.” Available: <https://git-scm.com/docs/git-log>
- [8] Git Project, “Git-show: Show various types of objects.” Available: <https://git-scm.com/docs/git-show>
- [9] Git Project, “Git-restore: Restore working tree files.” Available: <https://git-scm.com/docs/git-restore>
- [10] Git Project, “Gitignore: Specifies intentionally untracked files to ignore.” Available: <https://git-scm.com/docs/gitignore>
- [11] GitHub, “Removing sensitive data from a repository.” Available: <https://docs.github.com/en/authentication/keeping-your-account-and-data-secure/removing-sensitive-data-from-a-repository>