

chapter one

Why Git Exists

Version control records how a project changes. This chapter begins with folder copies, compares centralized and distributed systems, and introduces the snapshot model used throughout the book. It ends by configuring the identity Git writes into new commits.

The folder full of "final" files

Imagine you are building a personal website. You will follow this same project, a repository called `portfolio`, through the entire book. It starts life as a plain folder on your computer with a few files in it:

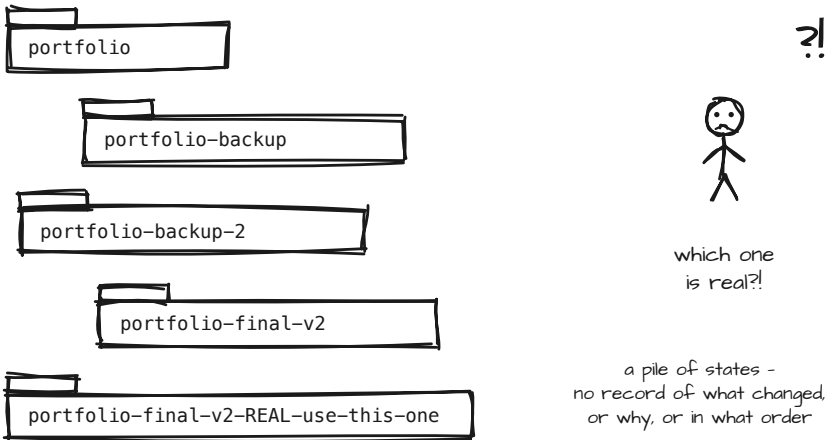
```
index.html
style.css
about.html
README.md
```

Before changing the layout in `index.html`, you copy the folder to `portfolio-backup`. Later copies acquire names such as `portfolio-backup-2` and `portfolio-final`. Soon the directory looks like this:

Backup names do not record relationships between versions.

```
portfolio
portfolio-backup
portfolio-backup-2
portfolio-final
portfolio-final-v2
portfolio-final-v2-REAL
portfolio-final-v2-REAL-use-this-one
```

The copies preserve states, but not a useful history. Comparing two versions requires comparing their files. The folder names do not say what changed or why, and collaboration adds another set of copies to reconcile.



Folder copies preserve states without recording a usable history between them.

What version control records

A commit identifies a project state and records metadata.

A version-control system, or VCS, records selected versions of a project. In Git, each recorded version is a **commit**. A commit lets you inspect an earlier state, compare two states, and connect a change to its author, timestamps, and message. Collaboration tools use that history to combine work from different people.

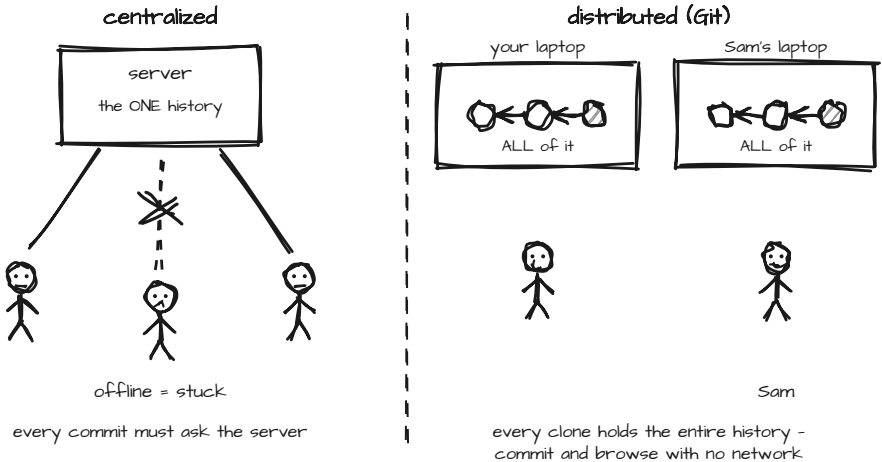
Centralized systems keep the authoritative history on a server.

In a **centralized** VCS such as CVS or Subversion, the authoritative repository lives on a server. A working copy contains the files needed for local work, while normal commits and many history operations contact that server. Availability and backup of the central repository therefore matter to the whole team.

Git is a **distributed** VCS. A normal clone contains the repository's reachable history, so commits, branches,

comparisons, and log queries work locally. Network commands exchange objects and refs with other repositories. Git also supports shallow and partial clones, but the full local clone is the model used in this book.

Most Git operations use the local repository.



A centralized working copy depends on the server repository; a normal Git clone has its own repository history.

Git presents history as snapshots

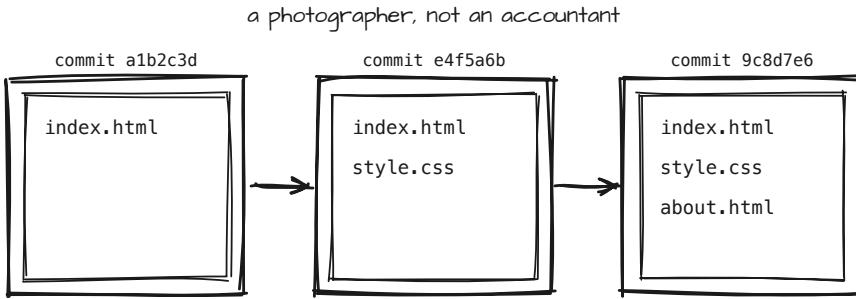
Git's data model presents each commit as a **snapshot**. More precisely, a commit names a root tree object, and that tree identifies the directories and file contents for the saved project state. You do not have to replay every earlier patch to identify that state.

Here are three commits to `portfolio`, drawn as snapshots:

commit a1b2c3d	commit e4f5a6b	commit 9c8d7e6
+-----+	+-----+	+-----+
index.html	index.html	index.html
	style.css	style.css
		about.html
+-----+	+-----+	+-----+

A commit names the root tree for one saved state.

Each box represents the project tree named by one commit. Asking Git for `e4f5a6b` starts from that commit's tree rather than reconstructing the state by applying every earlier change in sequence.

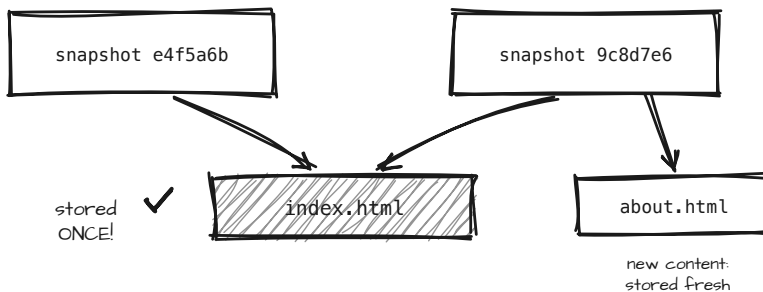


each snapshot = the WHOLE project, frozen

Each commit identifies a complete project tree at that point in history.

Snapshot semantics and delta-compressed storage can coexist.

Snapshot semantics do not require a separate copy of every unchanged file. Identical file contents have the same blob object ID, so trees can reuse the same blob. Git initially writes complete loose objects; when it packs objects, it may also use delta compression between similar objects. The snapshot model describes what a commit identifies, not a promise about the bytes used to store it.



unchanged files are stored once and shared

Two project trees can reuse the same blob for unchanged file content.

Parent links form history

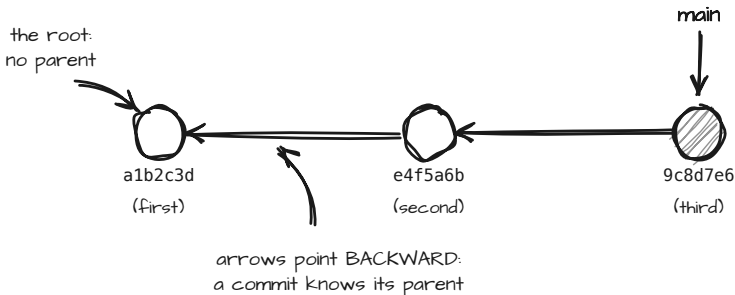
A commit also records its **parent** commit or commits. The first commit in a history has no parent. An ordinary later commit usually has one, while a merge commit has two or more.

For a simple sequence of three commits, the parent links form a line:

```
a1b2c3d <--- e4f5a6b <--- 9c8d7e6   (main)
(first)      (second)   (third)
```

The arrows point from each child commit to its parent. Parent links do not point forward to later children. When branching and merging add more lines, the same links form a directed graph rather than a single chain.

Parent links point backward; a merge records multiple parents.



Each arrow points from child to parent, and **main** names the tip commit.

The **main** label is a **branch ref**. In this example it names the newest commit. Creating another commit while **main** is checked out advances that ref to the new commit. Chapter 5 examines refs and **HEAD** in detail.

remember this!

An object ID such as the abbreviated `a1b2c3d` is calculated from the serialized object. A commit object includes its tree, parents, author and committer data, and message. Changing any of that data produces a different object ID. Chapter 4 examines the format.

Setting up Git

Git is available through common operating-system package managers and from `git-scm.com`. After installing it, print the installed version:

```
git --version
```

The examples in this book use Git 2 syntax. Some output and defaults can vary by version and configuration.

Set the default name and email Git will place in the author and committer fields of new commits:

```
git config --global user.name "Your Name"
git config --global user.email "you@example.com"
```

The `--global` flag writes user-level defaults, commonly in `~/.gitconfig`. A repository's local configuration can override them. To inspect the global values, run:

```
git config --global --list
```

```
user.name=Your Name
user.email=you@example.com
```

These values identify commits; they are not credentials for a remote hosting service. Some services associate

Commit identity and
hosting-service
authentication are
separate.

commits with accounts when the email matches a verified account address.

Recap

Version control records project states and the history connecting them. In Git, a commit names a project tree, records metadata, and links to its parent commits. Branch refs name positions in that graph. The next chapter creates the `portfolio` repository and records its first commits.

Exercises

Exercise 1.1. A friend says: "If Git presents every commit as a snapshot, it must store a separate full copy of every file for every commit." Explain why snapshot semantics do not imply that physical layout. Include blob reuse and packed-object delta compression.

Exercise 1.2. You are offline and want to commit to `portfolio` and browse its existing history. Explain why a normal Git clone can do both locally and why a centralized working copy normally needs its server for a commit. What changes if the Git clone is shallow?

Exercise 1.3. Draw the commit chain for a `portfolio` repo with three commits whose short hashes are `a1b2c3d`, `e4f5a6b`, and `9c8d7e6`, made in that order, with `main` at the tip. Draw the parent arrows in the correct direction, and say which commit is the root (has no parent).

Exercise 1.4. You run `git hash-object --no-filters` on the same bytes as a teammate using the same Git object format. Predict whether the object IDs match and explain why. Why can two commits that name the same project tree still have different object IDs?