O'REILLY®

3rd Edition

Version Control with Git

Powerful Tools and Techniques for Collaborative Software Development

Prem Kumar Ponuthorai
& Jon Loeliger

O'REILLY®

Version Control with Git

Get up to speed on Git for tracking, branching, merging, and managing code revisions. Through a series of step-by-step tutorials, this practical guide takes you quickly from Git fundamentals to advanced techniques, and provides friendly yet rigorous advice for navigating the many functions of this open source system for version control.

Authors Prem Kumar Ponuthorai and Jon Loeliger break down Git concepts according to level of proficiency. This thoroughly revised edition also includes tips for manipulating trees, extended coverage of the reflog and stash, and a complete introduction to GitHub. Git lets you manage code development in a virtually endless variety of ways, once you understand how to harness the system's flexibility. This book shows you how.

- **Leverage the advantages of a distributed version control system**
- **Learn how to use Git for several real-world development scenarios**
- **Gain insight into Git's common use cases, initial tasks, and basic functions**
- **Learn how to manage merges, conflicts, patches, and diffs**
- **Apply advanced techniques such as rebasing and hooks**
- **Migrate projects from different version control systems to Git**
- **Navigate, use, and contribute to repositories hosted on GitHub using lightweight branch-based workflow**

"This book progresses from the most basic Git concepts to advanced usage, so it will serve both readers who are new to Git as well as experienced users looking for tips and tricks."

—**Jeff King**
Git Contributor, Open Source Developer

Prem Kumar Ponuthorai is responsible for strategizing and enabling GitHub's offerings for the Expert Services Delivery organization. Having built on his software engineering background by becoming a Git convert, Prem has given Git workshops at conferences and provided training in Git for enterprise customers across diverse industries.

Jon Loeliger is a freelance software engineer who contributes to open source projects such as Linux, U-Boot, and Git. He's given tutorial presentations on Git at many conferences including Linux World, and has written several papers on Git for *Linux Magazine*. Jon holds degrees in computer science from Purdue University.

SOFTWARE DEVELOPMENT

US $65.99 CAN $82.99
ISBN: 978-1-492-09119-6

Twitter: @oreillymedia
linkedin.com/company/oreilly-media
youtube.com/oreillymedia

THIRD EDITION

Version Control with Git

Powerful Tools and Techniques for Collaborative Software Development

Prem Kumar Ponuthorai and Jon Loeliger

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

Version Control with Git
by Prem Kumar Ponuthorai and Jon Loeliger

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://oreilly.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Acquisitions Editor: Melissa Duffield
Development Editors: Virginia Wilson and Shira Evans
Production Editor: Beth Kelly
Copyeditor: Audrey Doyle
Proofreader: Piper Editorial Consulting, LLC
Indexer: Sue Klefstad
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Kate Dullea

May 2009: First Edition
August 2012: Second Edition
November 2022: Third Edition

Revision History for the Third Edition
2022-10-21: First Release
2022-11-04: Second Release
2023-02-03: Third Release

See *http://oreilly.com/catalog/errata.csp?isbn=9781492091196* for release details.

978-1-492-09119-6

[LSI]

Table of Contents

Preface

Git is a free, open source, distributed version control system created by Linus Torvalds. Git requires low operational overhead, yet is flexible and powerful enough to support the demands of complex, large-scale, distributed software development projects.

Our goal in this book is to show you how to get the most out of Git and how to manage a Git repository with ease. By the end, you will have learned Git's philosophy, fundamental concepts, and intermediate to advanced skills for tracking content, collaborating, and managing your projects across teams.

Who This Book Is For

We wrote this book with software engineers (developers, infrastructure engineers, DevOps, etc.) in mind as our primary audience. As such, most of the concepts and examples we use relate to the daily routines and tasks of folks in the software development industry. However, Git is robust enough to track content in areas as varied as data science, graphic design, and book authoring, just to name a few. (Case in point: we used Git as our underlying versioning system to keep track of reviews and edits while writing this book!) Regardless of your title or level of proficiency, if you are using Git as your version control system, you will find value in these pages.

Essential Know-How

Prior experience with any version control system, its aims, and its goals will be a helpful foundation to understand how Git works and to build upon your knowledge as you read this book. You should have some familiarity with using any command-line tool, such as the Unix shell, along with basic knowledge of shell commands, because we use a lot of command-line instructions in the examples and discussions in the book. A general understanding of programming concepts is also a plus.

We developed the examples on the macOS and Ubuntu environments. The examples should work under other platforms such as Debian, Solaris, and Windows (using Git-installed command-line tools, such as Git for Windows), but you can expect slight variations.

Some exercises in the examples may require system-level operations that need root access on machines. Naturally, in such situations you should have a clear understanding of the responsibilities of operations that need root access.

New in This Revision

In this third edition, we take an entirely new, modular approach to the topics by breaking down the concepts of Git. We start by introducing you to the basics and the fundamental philosophy of Git, then gradually build upon intermediate commands to help you efficiently supplement your daily development workflow, and finally conclude with advanced `git` commands and concepts to help you become proficient in understanding the inner mechanics of how Git works under the hood.

Another change we made in this edition was adding more illustrations to explain complex Git concepts to give you a mental model for easier comprehension. We also highlight features from the latest release of Git and provide you with examples and tips that can help improve your current distributed development workflow.

Navigating the Book

We organized this edition into categories according to the reader's familiarity and experience using Git. While we categorize the sections to get progressively more advanced to incrementally build your proficiency with Git, we designed the chapters within each section so that you can leverage the content either as standalone topics or as a series of topics building on one another sequentially.

We strove to apply a consistent structure and a consistent approach to teaching concepts in every chapter. We encourage you to take a moment to internalize this format. This will help you leverage and navigate the book as a handy reference at any point in the future.

If you have picked up the book amid juggling other responsibilities and are wondering what would be the best order to hit the ground running, fret not. Table P-1 will help guide you toward the chapters we feel will help you gain the most knowledge in the least amount of time.

Table P-1. Categories matrix

	Thinking in Git	Fundamentals of Git	Intermediate skills	Advanced skills	Tips and tricks
Software engineering	x	x	x	x	x
Data scientist	x	x	x		x
Graphic designers	x	x			x
Academia	x	x			x
Content authors	x	x			x

Installing Git

To reinforce the lessons taught in the book, we highly encourage you to practice the example code snippets on your development machine. To follow along with the examples, you will need Git installed on your platform of choice. Because the steps to install Git vary according to the version of your operating system, we've provided instructions on how to install Git in Appendix B accordingly.

A Note on Inclusive Language

Another important point we would like to highlight about the examples is that we feel strongly about diversity and inclusion in tech, and raising awareness is a responsibility we take seriously. As a start, we will be using the word *main* to indicate the default branch name.

Omissions

Due to its active community base, Git is constantly evolving. Even as we write this edition, another new version of Git was published for commercial use: version 2.37.1, to be precise. It was not our intention to leave information out of this book; it's simply the inevitable reality when writing about an ever-changing technology.

We deliberately chose not to cover all of Git's own core commands and options so that we could instead focus on common and frequently used commands. We also do not cover every Git-related tool available, simply because there are too many.

Despite these omissions, we feel confident that this book will equip you with a strong foundation and prepare you to dive deeper into the realms of Git if the need arises. For a detailed list of release changes, you can look up the Release Notes documentation for Git.

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic
: Indicates new terms, URLs, email addresses, filenames, and file extensions.

`Constant width`
: Used for program listings as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

`Constant width bold`
: Shows commands or other text that should be typed literally by the user.

`Constant width italic`
: Shows text that should be replaced with user-supplied values or by values determined by context.

This icon signifies a useful hint or a tip.

This icon indicates a warning or caution.

This icon indicates a general note.

Furthermore, you should be familiar with basic shell commands to manipulate files and directories. Many examples will contain commands such as these to add or remove directories, copy files, or create simple files:

```
# Command to create a new directory
$ mkdir newdirectory


# Command to write content into a file
$ echo "Test line" &gt; file.txt

# Command to append content at the end of a file
$ echo "Another line" &gt;&gt; file.txt
```

```
# Command to make a copy of a file
$ cp file.txt copy-of-file.txt


# Command to remove a file
$ rm newdirectory/file

# Command to remove a file
$ rmdir newdirectory
```

Commands, root permissions, and commands that need to be executed with root permissions appear as a `sudo` operation:

```
# Install the Git core package
$ sudo apt-get install git-core
```

How you edit files or effect changes within your working directory is pretty much up to you. You should be familiar with a text editor. In this book, we'll denote the process of editing a file by either a direct comment or a pseudocommand:

```
# edit file.c to have some new text
$ edit file.c
```

O'Reilly Online Learning

For more than 40 years, *O'Reilly Media* has provided technology and business training, knowledge, and insight to help companies succeed.

Our unique network of experts and innovators share their knowledge and expertise through books, articles, and our online learning platform. O'Reilly's online learning platform gives you on-demand access to live training courses, in-depth learning paths, interactive coding environments, and a vast collection of text and video from O'Reilly and 200+ other publishers. For more information, visit *https://oreilly.com*.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (in the United States or Canada)
707-829-0515 (international or local)
707-829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at *https://oreil.ly/VCG3e*.

Email *bookquestions@oreilly.com* to comment or ask technical questions about this book.

For news and information about our books and courses, visit *https://oreilly.com*.

Find us on LinkedIn: *https://linkedin.com/company/oreilly-media*.

Follow us on Twitter: *https://twitter.com/oreillymedia*.

Watch us on YouTube: *https://youtube.com/oreillymedia*.

Acknowledgments

The collective sum of the knowledge of many outweighs the sum of one. In light of that, this project is made possible with the help of many people. I personally owe a huge thank-you to Matthew McCullough, who provided me the opportunity to continue to teach the world about Git and GitHub in this edition of the book. I also thank Jon Loeliger, the main author of the book, who's work provided a great reference to dig deep into the weeds of Git knowledge in earlier editions of the book.

I would like to thank my tech reviewers, Jeff King, Jess Males, Aaron Sumner, Donald Ellis, and Mislav Marohnić, who had to read through the raw writings of the book and provided immense feedback to ensure that the chapters took shape in a way that was comprehensible for everyone.

Also, I'd like to thank Taylor Blau, who provided early guidance and valuable feedback in earlier chapters of the book, which helped me approach the overall structure for later parts of the book. Lars Schneider developed the idea and concept for the section "Git LFS" on page 426, which was based on a talk he prepared; for this, I thank him deeply. The works of Elijah Newren, Derrick Stolee, and Vincent Driessen are referenced with permission, and I am grateful for their contribution. I would also like to thank Peter Murray; our talks about the structure of the book provided me assurance and guidance that I was on the right track with the changes being introduced for this third edition.

To my editors and to the staff at O'Reilly—especially Shira Evans, Virginia Wilson, Beth Kelly, Audrey Doyle, Kim Sandoval, Sue Klefstad, David Futato, Karen Montgomery, Kate Dullea, Suzanne Huston, and Melissa Duffield—I extend a heartfelt thank-you for your patience, motivation, and cheering to ensure that we would get this across the finish line.

Finally, I want to thank my wife, Tejirl, and my daughter, Temyksciraa, for providing unconditional moral support and patience; for sacrificing family dinners, date nights, and holidays; and most of all, for believing in me. Thanks also to my parents, Ponuthorai and Jayaletchmy, who taught me perseverance. And a special thank-you to

Raksha, my sweet White Swiss Shepherd, who patiently waited by my side throughout the entire writing process.

Attributions

The Git Trademark Policy is available online.

Linux® is the registered trademark of Linus Torvalds in the United States and other countries.

PowerPC® is a trademark of International Business Machines Corporation in the United States, other countries, or both.

Unix® is a registered trademark of The Open Group in the United States and other countries.

PART I
Thinking in Git

The chapters in this first part of the book introduce you to Git and its foundations. We start by providing you with an overarching view of the Git ecosystem, followed by a look at Git's characteristics so that you can learn what makes Git unique and how it operates. Following this, we share examples of what it looks like to work with Git via the command-line interface and some code examples.

In the second part of the book, we dive into the technicalities of how Git works internally. We start by explaining repositories before discussing the Git object store, further solidifying the notion that Git under the hood is really a content-addressable database. We made the decision to discuss Git internals early in the book because we believe it will help you prepare for later chapters.

In a nutshell, the first two chapters of Part I aim to put you in a Git mindset. Git is perceived to have a steep learning curve, but it really has to do with the way you understand how files and projects are being version-controlled, especially if you are transitioning from a traditional centralized version control system to Git, which is distributed in its implementation. It is this transition that may make understanding Git complex early on. You can rest assured that once you learn to think in Git, the rest of the technical skill will flow.

CHAPTER 1

Introduction to Git

Simply put, Git is a content tracker. Given that notion, Git shares common principles of most version control systems. However, the distinct feature that makes Git unique among the variety of tools available today is that it is a distributed version control system. This distinction means Git is fast and scalable, has a rich collection of command sets that provide access to both high-level and low-level operations, and is optimized for local operations.

In this chapter you will learn the fundamental principles of Git, its characteristics, and basic `git` commands, and you'll receive some quick guidance on creating and adding changes to a repository.

We highly recommend that you take time to grasp the important concepts explained here. These topics are the building blocks of Git and will help you understand the intermediate and advanced techniques for managing a Git repository as part of your daily work. These foundational concepts will also help you ramp up your learning when we break down the inner workings of Git in chapters grouped in Part II, "Fundamentals of Git", Part III, "Intermediate Skills", and Part IV, "Advanced Skills".

Git Components

Before we dive into the world of `git` commands, let's take a step back and visualize an overview of the components that make up the Git ecosystem. Figure 1-1 shows how the components work together.

Git GUI tools act as a frontend for the Git command line, and some tools have extensions that integrate with popular Git hosting platforms. The Git client tools mostly work on the local copy of your repository.

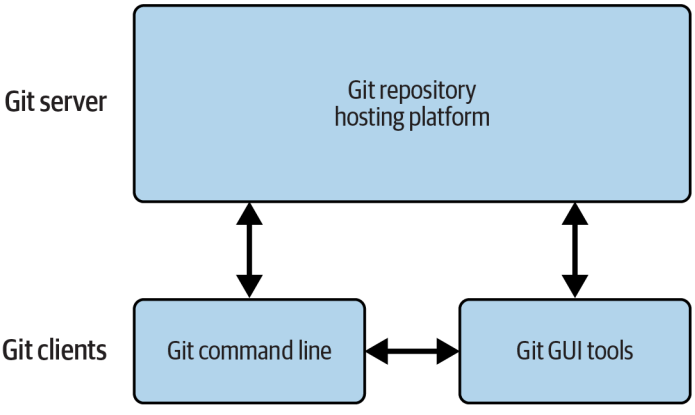

Figure 1-1. Overview of Git components

When you are working with Git, a typical setup includes a Git server and Git clients. You can possibly forgo a server, but that would add complexity to how you maintain and manage repositories when sharing revision changes in a collaborative setup and would make consistency more difficult (we will revisit this in Chapter 11). The Git server and clients work as follows:

Git server
: A Git server enables you to collaborate more easily because it ensures the availability of a central and reliable source of truth for the repositories you will be working on. A Git server is also where your remote Git repositories are stored; as common practice goes, the repository has the most up-to-date and stable source of your projects. You have the option to install and configure your own Git server, or you can forgo the overhead and opt to host your Git repositories on reliable third-party hosting sites such as GitHub, GitLab, and Bitbucket.

Git clients
: Git clients interact with your local repositories, and you are able to interact with Git clients via the Git command line or the Git GUI tools. When you install and configure a Git client, you will be able to access the remote repositories, work on a local copy of the repository, and push changes back to the Git server. If you are new to Git, we recommend starting out using the Git command line; familiarize yourself with the common subset of `git` commands required for your day-to-day operations and then progress to a Git GUI tool of your choice.

The reason for this approach is that to some extent, Git GUI tools tend to provide terminologies that represent a desired outcome that may not be part of Git's standard commands. An example would be a tool with an option called `sync`, which masks the underlying chaining of two or more `git` commands to achieve a desired outcome. If

for some reason you were to enter the `sync` subcommand on the command line, you might get this confusing output:

```
$ git sync

git: 'sync' is not a git command. See 'git --help'.

The most similar command is
       svn
```

`git sync` is not a valid `git` subcommand. To ensure that your local working copy of the repository is in sync with changes from the remote Git repository, you will need to run a combination of these commands: `git fetch`, `git merge`, `git pull`, or `git push`.

There are a plethora of tools available at your disposal. Some Git GUI tools are fancy and extensible via a plug-in model that provides you the option to connect and leverage features made available on popular third-party Git hosting sites. As convenient as it may be to learn Git via a GUI tool, we will be focusing on the Git command-line tool for examples and code discussions, since this builds a good foundational knowledge that will lead to Git dexterity.

Git Characteristics

Now that we have given an overview of the Git components, let's learn about the characteristics of Git. Understanding these distinct traits of Git enables you to effortlessly switch from a centralized version control mindset to a distributed version control mentality. We like to refer to this as "Thinking in Git":

Git stores revision changes as snapshots
: The very first concept to unlearn is the way Git stores multiple revisions of a file that you are working on. Unlike other version control systems, Git does not track revision changes as a series of modifications, commonly known as *deltas*; instead, it takes a snapshot of changes made to the state of your repository at a specific point in time. In Git terminology this is known as a *commit*. Think of this as capturing a moment in time, as through a photograph.

Git is enhanced for local development
: In Git, you work on a copy of the repository on your local development machine. This is known as a *local repository*, or a clone of the remote repository on a Git server. Your local repository will have the resources and the snapshots of the revision changes made on those resources all in one location. Git terms these collections of linked snapshots *repository commit history*, or *repo history* for short. This allows you to work in a disconnected environment since Git does not need a constant connection to the Git server to version-control your changes. As a

natural consequence, you are able to work on large, complex projects across distributed teams without compromising efficiency and performance for version control operations.

Git is definitive
: *Definitive* means the `git` commands are explicit. Git waits for you to provide instructions on what to do and when to do it. For example, Git does not automatically sync changes from your local repository to the remote repository, nor does it automatically save a snapshot of a revision to your local repo history. Every action requires your explicit command or instruction to tell Git what is required, including adding new commits, fixing existing commits, pushing changes from your local repository to the remote repository, and even retrieving new changes from the remote repository. In short, you need to be intentional with your actions. This also includes letting Git know which files you intend to track, since Git does not automatically add new files to be version-controlled.

Git is designed to bolster nonlinear development
: Git allows you to ideate and experiment with various implementations of features for viable solutions to your project by enabling you to diverge and work in parallel along the main, stable codebase of your project. This methodology, called *branching*, is a very common practice and ensures the integrity of the main development line, preventiing any accidental changes that may break it.

In Git, the concept of branching is considered lightweight and inexpensive because a branch in Git is just a pointer to the latest commit in a series of linked commits. For every branch you create, Git keeps track of the series of commits for that branch. You can switch between branches locally. Git then restores the state of the project to the most recent moment when the snapshot of the specified branch was created. When you decide to merge the changes from any branch into the main development line, Git is able to combine those series of commits by applying techniques that we will discuss in Chapter 6.

Since Git offers many novelties, keep in mind that the concepts and practices of other version control systems may work differently or may not be applicable at all in Git.

The Git Command Line

Git's command-line interface is simple to use. It is designed to put full control of your repository into your hands. As such, there are many ways to do the same thing. By focusing on which commands are important for your day-to-day work, we can simplify and learn them in more depth.

As a starting point, just type **`git version`** or **`git --version`** to determine whether your machine has already been preloaded with Git. You should see output similar to the following:

```
$ git --version
git version 2.37.0
```

If you do not have Git installed on your machine, please refer to Appendix B to learn how you can install Git according to your operating system platform before continuing with the next section.

Upon installation, type **`git`** without any arguments. Git will then list its options and the most common subcommands:

```
$ git

usage: git [-v | --version] [-h | --help] [-C <path>] [-c <name>=<value>]
           [--exec-path[=<path>]] [--html-path] [--man-path] [--info-path]
           [-p | --paginate | -P | --no-pager] [--no-replace-objects] [--bare]
           [--git-dir=<path>] [--work-tree=<path>] [--namespace=<name>]
           [--super-prefix=<path>] [--config-env=<name>=<envvar>]
           <command> [<args>]

These are common Git commands used in various situations:

start a working area (see also: git help tutorial)
   clone     Clone a repository into a new directory
   init      Create an empty Git repository or reinitialize an existing one

work on the current change (see also: git help everyday)
   add       Add file contents to the index
   mv        Move or rename a file, a directory, or a symlink
   restore   Restore working tree files
   rm        Remove files from the working tree and from the index

examine the history and state (see also: git help revisions)
   bisect    Use binary search to find the commit that introduced a bug
   diff      Show changes between commits, commit and working tree, etc
   grep      Print lines matching a pattern
   log       Show commit logs
   show      Show various types of objects
   status    Show the working tree status

grow, mark and tweak your common history
   branch    List, create, or delete branches
   commit    Record changes to the repository
   merge     Join two or more development histories together
   rebase    Reapply commits on top of another base tip
   reset     Reset current HEAD to the specified state
   switch    Switch branches
   tag       Create, list, delete or verify a tag object signed with GPG

collaborate (see also: git help workflows)
   fetch     Download objects and refs from another repository
   pull      Fetch from and integrate with another repository or a local branch
   push      Update remote refs along with associated objects

'git help -a' and 'git help -g' list available subcommands and some(((("git help command")))
concept guides. See 'git help <command>' or 'git help <concept>'
to read about a specific subcommand or concept.
See 'git help git' for an overview of the system.
```

For a complete list of `git` subcommands, type **`git help --all`**.

As you can see from the usage hint, a small handful of options apply to `git`. Most options, shown as `[ARGS]` in the hint, apply to specific subcommands.

For example, the option `--version` affects the `git` command and produces a version number:

```
$ git --version
git version 2.37.0
```

In contrast, `--amend` is an example of an option specific to the `git` subcommand `commit`:

```
$ git commit --amend
```

Some invocations require both forms of options (here, the extra spaces in the command line merely serve to visually separate the subcommand from the base command and are not required):

```
$ git --git-dir=project.git    repack -d
```

For convenience, documentation for each `git` subcommand is available using `git help subcommand`, `git --help subcommand`, `git subcommand --help`, or `man git-`*`subcommand`*.

The complete Git documentation is online.

Historically, Git was provided as a suite of many simple, distinct, standalone commands developed according to the Unix philosophy: build small, interoperable tools. Each command sported a hyphenated name, such as `git-commit` and `git-log`. However, modern Git installations no longer support the hyphenated command forms and instead use a single `git` executable with a subcommand.

The `git` commands understand both "short" and "long" options. For example, the `git commit` command treats the following examples equivalently:

```
$ git commit -m "Fix a typo."
$ git commit --message="Fix a typo."
```

The short form, `-m`, uses one hyphen, whereas the long form, `--message`, uses two. (This is consistent with the GNU long options extension.) Some options exist in only one form.

You can create a commit summary and detailed message for the summary by using the `-m` option multiple times:

```
$ git commit -m "Summary" -m "Detail of Summary"
```

Finally, you can separate options from a list of arguments via the *bare double dash* convention. For instance, use the double dash to contrast the control portion of the command line from a list of operands, such as filenames:

```
$ git diff -w main origin -- tools/Makefile
```

You may need to use the double dash to separate and explicitly identify filenames so that they are not mistaken for another part of the command. For example, if you happened to have both a file and a tag named *main.c*, then you will need to be intentional with your operations:

```
# Checkout the tag named "main.c"
$ git checkout main.c

# Checkout the file named "main.c"
$ git checkout -- main.c
```

Quick Introduction to Using Git

To see Git in action, you can create a new repository, add some content, and track a few revisions. You can create a repository in two ways: either create a repository from scratch and populate it with some content, or work with an existing repository by *cloning* it from a remote Git server.

Preparing to Work with Git

Whether you are creating a new repository or working with an existing repository, there are basic prerequisite configurations that you need to complete after installing Git on your local development machine. This is akin to setting up the correct date, time zone, and language on a new camera before taking your first snapshot.

At a bare minimum, Git requires your name and email address before you make your first commit in your repository. The identity you supply then shows as the commit *author*, baked in with other snapshot metadata. You can save your identity in a configuration file using the `git config` command:

```
$ git config user.name "Jon Loeliger"
$ git config user.email "jdl@example.com"
```

If you decide not to include your identity in a configuration file, you will have to specify your identity for every `git commit` subcommand by appending the argument `--author` at the end of the command:

```
$ git commit -m "log message" --author="Jon Loeliger <jdl@example.com>"
```

Keep in mind that this is the hard way, and it can quickly become tedious.

You can also specify your identity by supplying your name and email address to the `GIT_AUTHOR_NAME` and `GIT_AUTHOR_EMAIL` environment variables, respectively. If set, these variables will override all configuration settings. However, for specifications set on the command line, Git will override the values supplied in the configuration file and environment variable.

Working with a Local Repository

Now that you have configured your identity, you are ready to start working with a repository. Start by creating a new empty repository on your local development machine. We will start simple and work our way toward techniques for working with a shared repository on a Git server.

Creating an initial repository

We will model a typical situation by creating a repository for your personal website. Let's assume you're starting from scratch and you are going to add content for your project in the local directory *~/my_website*, which you place in a Git repository.

Type in the following commands to create the directory, and place some basic content in a file called *index.html*:

```
$ mkdir ~/my_website
$ cd ~/my_website
$ echo 'My awesome website!' > index.html
```

To convert *~/my_website* into a Git repository, run `git init`. Here we provide the option `-b` followed by a default branch named `main`:

```
$ git init -b main

Initialized empty Git repository in ../my_website/.git/
```

If you prefer to initialize an empty Git repository first and then add files to it, you can do so by running the following commands:

```
$ git init -b main ~/my_website

Initialized empty Git repository in ../my_website/.git/

$ cd ~/my_website
$ echo 'My awesome website!' > index.html
```

You can initialize a completely empty directory or an existing directory full of files. In either case, the process of converting the directory into a Git repository is the same.

The `git init` command creates a hidden directory called *.git* at the root level of your project. All revision information along with supporting metadata and Git extensions are stored in this top-level, hidden *.git* folder.

Git considers *~/my_website* to be the *working directory*. This directory contains the current version of files for your website. When you make changes to existing files or add new files to your project, Git records those changes in the hidden *.git* folder.

For the purpose of learning, we will reference two virtual directories that we call *Local History* and *Index* to illustrate the concept of initializing a new Git repository. We will discuss the *Local History* and *Index* in Chapters 4 and 5, respectively.

Figure 1-2 depicts what we have just explained:

```
.
└── my_website
    ├── .git/
    │   └── Hidden git objects
    └── index.html
```

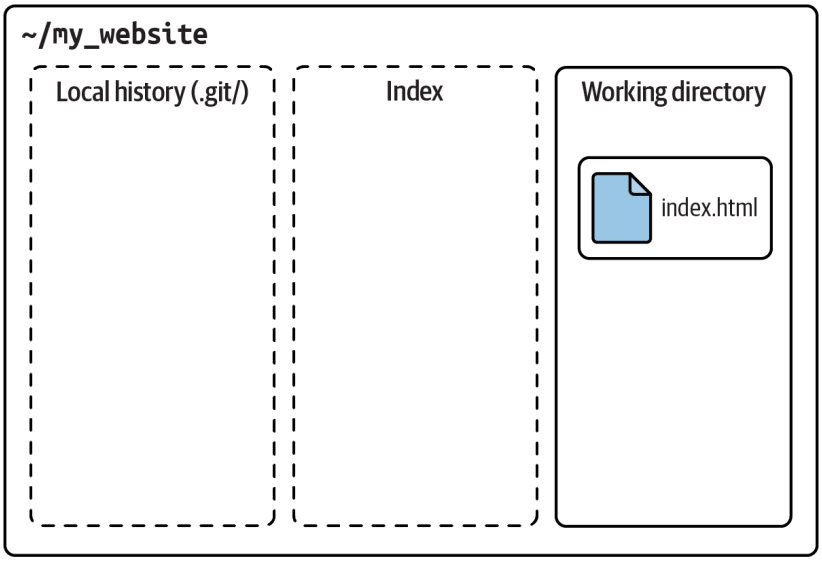

Figure 1-2. Initial repository

The dotted lines surrounding the *Local History* and *Index* represent the hidden directories within the *.git* folder.

Adding a file to your repository

Up to this point, you have only *created* a new Git repository. In other words, this Git repository is empty. Although the file *index.html* exists in the directory *~/my_website*, to Git, this is the *working directory*, a representation of a scratch pad or directory where you frequently alter your files.

When you have finalized changes to the files and want to deposit those changes into the Git repository, you need to explicitly do so by using the `git add` *`file`* command:

```
$ git add index.html
```

Although you can let Git add all the files in the directory and all subdirectories using the `git add .` command, this stages everything, and we advise you to be intentional with what you are planning to stage, mainly to prevent sensitive information or unwanted files from being included when commits are made. To avoid including such information, you can use the *.gitignore* file, which is covered in Chapter 5.

The argument `.`, the single period or *dot* in Unix parlance, is shorthand for the current directory.

With the `git add` command, Git understands that you intend to include the final iteration of the modification on *index.html* as a revision in the repository. However, so far Git has merely staged the file, an interim step before taking a snapshot via a commit.

Git separates the `add` and `commit` steps to avoid volatility while providing flexibility and granularity in how you record changes. Imagine how disruptive, confusing, and time-consuming it would be to update the repository each time you add, remove, or change a file. Instead, multiple provisional and related steps, such as an add, can be *batched*, thereby keeping the repository in a stable, consistent state. This method also allows us to craft a narrative of why we are changing the code. In Chapter 4 we will dive deeper into this concept.

We recommend that you strive to group logical change batches before making a commit. This is called an *atomic* commit and will help you in situations where you'll need to do some advanced Git operations in later chapters.

Running the `git status` command reveals this in-between state of *index.html*:

```
$ git status

On branch main

No commits yet

Changes to be committed:
```

```
  (use "git rm --cached <file>..." to unstage)
        new file:   index.html
```

The command reports that the new file *index.html* will be added to the repository during the next commit.

After staging the file, the next logical step is to commit the file to the repository. Once you commit the file, it becomes part of the *repository commit history*; for brevity, we will refer to this as the *repo history*. Every time you make a commit, Git records several other pieces of metadata along with it, most notably the commit log message and the author of the change.

A fully qualified `git commit` command should supply a terse and meaningful log message using active language to denote the change that is being introduced by the commit. This is very helpful when you need to traverse the repo history to track down a specific change or quickly identify changes of a commit without having to dig deeper into the change details. We dive in deeper on this topic in Chapters 4 and 8.

Let's commit the staged *index.html* file for your website:

```
$ git commit -m "Initial contents of my_website"

[main (root-commit) c149e12] initial contents of my_website
 1 file changed, 1 insertion(+)
 create mode 100644 index.html
```

The details of the author who is making the commit are retrieved from the Git configuration we set up earlier.

In the code example, we supplied the `-m` argument to be able to provide the log message directly on the command line. If you prefer to provide a detailed log message via an interactive editor session, you can do so as well. You will need to configure Git to launch your favorite editor during a `git commit` (leave out the `-m` argument); if it isn't set already, you can set the `$GIT_EDITOR` environment variable as follows:

```
# In bash or zsh
$ export GIT_EDITOR=vim

# In tcsh
$ setenv GIT_EDITOR emacs
```

Git will honor the default text editor configured in the shell environment variables `VISUAL` and `EDITOR`. If neither is configured, it falls back to using the `vi` editor.

After you commit the *index.html* file into the repository, run `git status` to get an update on the current state of your repository. In our example, running `git status` should indicate that there are no outstanding changes to be committed:

```
$ git status

On branch main
nothing to commit, working tree clean
```

Git also tells you that your *working directory* is clean, which means the working directory has no new or modified files that differ from what is in the repository.

Figure 1-3 will help you visualize all the steps you just learned.

The difference between `git add` and `git commit` is much like you organizing a group of schoolchildren in a preferred order to get the perfect classroom photograph: `git add` does the organizing, whereas `git commit` takes the snapshot.

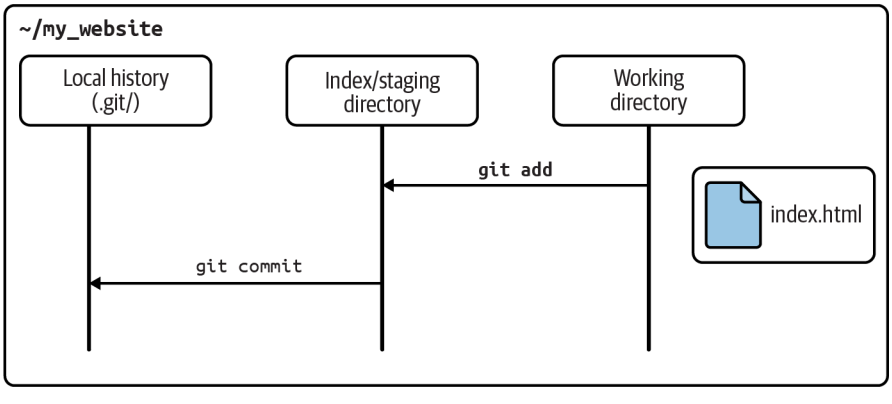

Figure 1-3. Staging and adding a file to a repository

Making another commit

Next, let's make a few modifications to *index.html* and create a repo history within the repository.

Convert *index.html* into a proper HTML file, and commit the alteration to it:

```
$ cd ~/my_website

# edit the index.html file.

$ cat index.html
<html>
<body>
My website is awesome!
</body>
</html>

$ git commit index.html -m 'Convert to HTML'
```

```
[main 521edbe] Convert to HTML
 1 file changed, 5 insertions(+), 1 deletion(-)
```

If you are already familiar with Git, you may be wondering why we skipped the `git add index.html` step before we committed the file. It is because the content to be committed may be specified in more than one way in Git.

Type **`git commit --help`** to learn more about these options:

```
$ git commit --help

NAME
    git-commit - Record changes to the repository

SYNOPSIS
    git commit [-a | --interactive | --patch] [-s] [-v] [-u<mode>] [--amend]
               [--dry-run] [(-c | -C | --squash) <commit> | --fixup [(amend|reword):]<commit>)]
               [-F <file> | -m <msg>] [--reset-author] [--allow-empty]
               [--allow-empty-message] [--no-verify] [-e] [--author=<author>]
               [--date=<date>] [--cleanup=<mode>] [--[no-]status]
               [-i | -o] [--pathspec-from-file=<file> [--pathspec-file-nul]]
               [(--trailer <token>[(=|:)<value>])...] [-S[<keyid>]]
               [--] [<pathspec>...]

...
```

Detailed explanations of the various commit methods are also explained in the `git commit --help` manual pages.

In our example, we decided to commit the *index.html* file with an additional argument, the `-m` switch, which supplied a message explaining the changes in the commit: `'Convert to HTML'`. Figure 1-4 explains the method we just discussed.

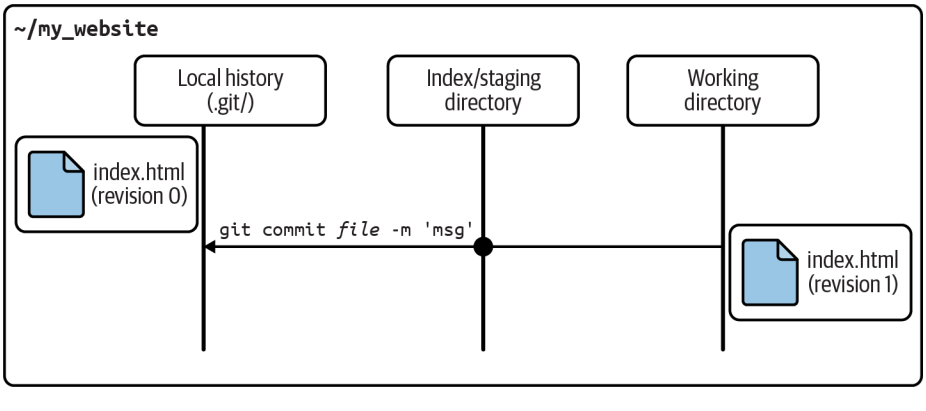

Figure 1-4. Staging and adding changes to a tracked file in a repository

Note that our usage of `git commit index.html -m 'Convert to HTML'` does not skip the staging of the file; Git handles it automatically as part of the commit action.

Viewing your commits

Now that you have more commits in the repo history, you can inspect them in a variety of ways. Some `git` commands show the sequence of individual commits, others show the summary of an individual commit, and still others show the full details of any commit you specify in the repository.

The `git log` command yields a sequential history of the individual commits within the repository:

```
$ git log

commit 521edbe1dd2ec9c6f959c504d12615a751b5218f (HEAD -> main)
Author: Jon Loeliger <jdl@example.com>
Date:   Mon Jul 4 12:01:54 2022 +0200

    Convert to HTML

commit c149e12e89a9c035b9240e057b592ebfc9c88ea4
Author: Jon Loeliger <jdl@example.com>
Date:   Mon Jul 4 11:58:36 2022 +0200

    Initial contents of my_website
```

In the preceding output, the `git log` command prints out detailed log information for every commit in the repository. At this point you have only two commits in your repo history, which makes it easier to read the output. For repositories with many commit histories, this standard view may not help you traverse a long list of detailed commit information with ease; in such situations you can provide the `--oneline` switch to list a summarized commit ID number along with the commit message:

```
$ git log --oneline

521edbe (HEAD -> main) Convert to HTML
c149e12 Initial contents of my_website
```

The commit log entries are listed, in order, from most recent to oldest[1] (the original file); each entry shows the commit author's name and email address, the date of the commit, the log message for the change, and the internal identification number of the commit. The commit ID number is explained in "Content-Addressable Database" on page 29. We will discuss commits in more detail in Chapter 4.

If you want to see more detail about a particular commit, use the `git show` command with a commit ID number:

1 Strictly speaking, they are not in *chronological* order but rather are a *topological* sort of the commits.

```
$ git show c149e12e89a9c035b9240e057b592ebfc9c88ea4

commit c149e12e89a9c035b9240e057b592ebfc9c88ea4
Author: Jon Loeliger <jdl@example.com>
Date:   Mon Jul 4 11:58:36 2022 +0200

Initial contents of my_website

diff --git a/index.html b/index.html
new file mode 100644
index 0000000..6331c71
--- /dev/null
+++ b/index.html
@@ -0,0 +1 @@
+My awesome website!
```

If you run `git show` without an explicit commit number, it simply shows the details of the `HEAD` commit, in our case, the most recent one.

The `git log` command shows the commit logs for how changes for each commit are included in the repo history. If you want to see concise, one-line summaries for the current development branch without supplying additional filter options to the `git log --oneline` command, an alternative approach is to use the `git show-branch` command:

```
$ git show-branch --more=10

[main] Convert to HTML
[main^] Initial contents of my_website
```

The phrase `--more=10` reveals up to an additional 10 versions, but only two exist so far and so both are shown. (The default in this case would list only the most recent commit.) The name `main` is the default branch name.

We will discuss branches and revisit the `git show-branch` command in more detail in Chapter 3.

Viewing commit differences

With the repo history in place from the addition of commits, you can now see the differences between the two revisions of *index.html*. You will need to recall the commit ID numbers and run the `git diff` command:

```
$ git diff c149e12e89a9c035b9240e057b592ebfc9c88ea4 \
           521edbe1dd2ec9c6f959c504d12615a751b5218f

diff --git a/index.html b/index.html
index 6331c71..8cfcb90 100644
--- a/index.html
+++ b/index.html
@@ -1 +1,5 @@
-My awesome website!
```

```
+<html>
+<body>
 My website is awesome!
+</body>
+</html>
```

The output resembles what the `git diff` command produces. As per convention, the first revision commit, `9da581d910c9c4ac93557ca4859e767f5caf5169`, is the earlier of the content for *index.html*, and the second revision commit, `ec232cddfb94e0dfd5b5855af8ded7f5eb5c90d6`, is the latest content of *index.html*. Thus, a plus sign (+) precedes each line of new content after the minus sign (–), which indicates removed content.

Do not be intimidated by the long hex numbers. Git provides many shorter, easier ways to run similar commands so that you can avoid large, complicated commit IDs. Usually the first seven characters of the hex numbers, as shown in the `git log --oneline` example earlier, are sufficient. We elaborate on this in "Content-Addressable Database" on page 29.

Removing and renaming files in your repository

Now that you have learned how to add files to a Git repository, let's look at how to remove a file from one. Removing a file from a Git repository is analogous to adding a file but uses the `git rm` command. Suppose you have the file *adverts.html* in your website content and plan to remove it. You can do so as follows:

```
$ cd ~/my_website
$ ls
index.html  adverts.html

$ git rm adverts.html
rm  'adverts.html'

$ git commit -m "Remove adverts html"
[main 97ff70a] Remove adverts html
 1 file changed, 0 insertions(+), 0 deletions(-)
 delete mode 100644 adverts.html
```

Similar to an addition, a deletion also requires two steps: express your intent to remove the file using `git rm`, which also *stages* the file you intend to remove. Realize the change in the repository with a `git commit`.

Just as with `git add`, with `git rm` we are not directly deleting the file; instead, we are changing what is tracked: the deletion or addition of a file.

You can rename a file indirectly by using a combination of the `git rm` and `git add` commands, or you can rename the file more quickly and directly with the command `git mv`. Here's an example of the former:

```
$ mv foo.html bar.html
$ git rm foo.html
rm 'foo.html'

$ git add bar.html
```

In this sequence, you must execute `mv foo.html bar.html` at the onset to prevent `git rm` from permanently deleting the *foo.html* file from the filesystem.

Here's the same operation performed with `git mv`:

```
$ git mv foo.html bar.html
```

In either case, the staged changes must subsequently be committed:

```
$ git commit -m "Moved foo to bar"
[main d1e37c8] Moved foo to bar
 1 file changed, 0 insertions(+), 0 deletions(-)
 rename foo.html => bar.html (100%)
```

Git handles file move operations differently than most similar systems, employing a mechanism based on the similarity of the content between two file versions. The specifics are described in Chapter 5.

Working with a Shared Repository

By now you have initialized a new repository and have been making changes to it. All the changes are only available to your local development machine. This is a good example of how you can manage a project that is only available to you. But how can you work collaboratively on a repository that is hosted on a Git server? Let's discuss how you can achieve this.

Making a local copy of the repository

You can create a complete copy, or a *clone*, of a repository using the `git clone` command. This is how you collaborate with other people, making changes on the same files and keeping in sync with changes from other versions of the same repository.

For the purposes of this tutorial, let's start simple by creating a copy of your existing repository; then we can contrast the same example as if it were on a remote Git server:

```
$ cd ~
$ git clone my_website new_website
```

Although these two Git repositories now contain exactly the same objects, files, and directories, there are some subtle differences. You may want to explore those differences with commands such as the following:

```
$ ls -lsa my_website new_website
...
$ diff -r my_website new_website
...
```

On a local filesystem like this, using `git clone` to make a copy of a repository is quite similar to using `cp -a` or `rsync`. In contrast, if you were to *clone* the same repository from a Git server, the syntax would be as follows:

```
$ cd ~

$ git clone https://git-hosted-server.com/some-dir/my_website.git new_website
Cloning into 'new_website'...
remote: Enumerating objects: 2, done.
remote: Counting objects: 100% (2/2), done.
remote: Compressing objects: 100% (103/103), done.
remote: Total 125 (delta 45), reused 65 (delta 18), pack-reused 0
Receiving objects: 100% (125/125), 1.67 MiB | 4.03 MiB/s, done.
Resolving deltas: 100% (45/45), done.
```

Once you clone a repository, you can modify the cloned version, make new commits, inspect its logs and history, and so on. It is a complete repository with a full history. Remember that the changes you make to the cloned repository will not be automatically pushed to the original copy on the repository.

Figure 1-5 depicts this concept.

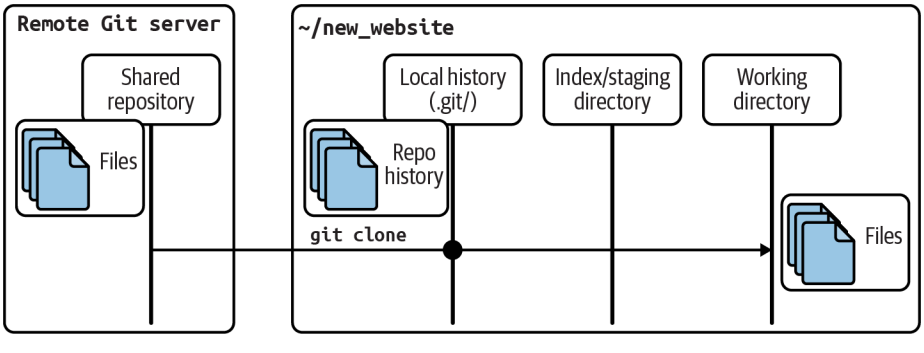

Figure 1-5. Cloning a shared repository

Try not to be distracted by some of the terms you see in the output. Git supports a richer set of repository sources, including network names, for naming the repository to be cloned. We will explain these forms and usage in Chapter 11.

Configuration Files

Git configuration files are all simple text files in the style of *.ini* files. The configuration files are used to store preferences and settings used by multiple `git` commands. Some of the settings represent personal preferences (e.g., should a `color.pager` be used?), others are important for a repository to function correctly (e.g., `core`

`repositoryformatversion`), and still others tweak `git` command behavior a bit (e.g., `gc.auto`). Like other tools, Git supports a hierarchy of configuration files.

Hierarchy of configuration files

Figure 1-6 represents the Git configuration files hierarchy in decreasing precedence:

.git/config
: Repository-specific configuration settings manipulated with the `--file` option or by default. You can also write to this file with the `--local` option. These settings have the highest precedence.

~/.gitconfig
: User-specific configuration settings manipulated with the `--global` option.

/etc/gitconfig
: System-wide configuration settings manipulated with the `--system` option if you have proper Unix file write permissions on the *gitconfig* file. These settings have the lowest precedence. Depending on your installation, the system settings file might be somewhere else (perhaps in */usr/local/etc gitconfig*) or may be absent entirely.

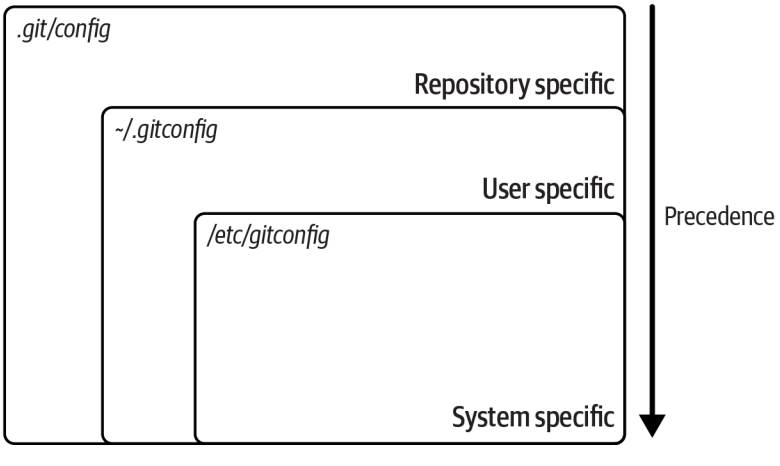

Figure 1-6. Git configuration files hierarchy

For example, to store an author name and email address that will be used on all the commits you make for all of your repositories, configure values for `user name` and `user.email` in your *$HOME/.gitconfig* file using `git config --global`:

```
$ git config --global user.name "Jon Loeliger"
$ git config --global user.email "jdl@example.com"
```

If you need to set a repository-specific name and email address that would override a `--global` setting, simply omit the `--global` flag or use the `--local` flag to be explicit:

```
$ git config user.name "Jon Loeliger"
$ git config user.email "jdl@special-project.example.org"
```

You can use `git config -l` (or the long form `--list`) to list the settings of all the variables collectively found in the complete set of configuration files:

```
# Make a brand-new, empty repository
$ mkdir /tmp/new
$ cd /tmp/new
$ git init

# Set some config values
$ git config --global user.name "Jon Loeliger"
$ git config --global user.email "jdl@example.com"
$ git config user.email "jdl@special-project.example.org"

$ git config -l
user.name=Jon Loeliger
user.email=jdl@example.com
core.repositoryformatversion=0
core.filemode=true
core.bare=false
core.logallrefupdates=true
user.email=jdl@special-project.example.org
```

When specifying the command `git config -l`, adding the options `--show-scope` and `--show-origin` will help to print the various sources for the configurations! Try this out with `git config -l --show-scope --show-origin` in your terminal.

Because the configuration files are simple text files, you can view their contents with `cat` and edit them with your favorite text editor too:

```
# Look at just the repository-specific settings

$ cat .git/config
[core]
    repositoryformatversion = 0
    filemode = true
    bare = false
    logallrefupdates = true
ignorecase = true
     precomposeunicode = true

[user]
    email = jdl@special-project.example.org
```

The content of the configuration text file may be presented with some slight differences according to your operating system type. Many of these differences allow for different filesystem characteristics.

If you need to remove a setting from the configuration files, use the `--unset` option together with the correct configuration files flag:

```
$ git config --unset --global user.email
```

Git provides you with many configuration options and environment variables that frequently exist for the same purpose. For example, you can set a value for the editor to be used when composing a commit log message. Based on the configuration, invocation follows these steps:

1. `GIT_EDITOR` environment variable
2. `core.editor` configuration option
3. `VISUAL` environment variable
4. `EDITOR` environment variable
5. The `vi` command

There are more than a few hundred configuration parameters. We will not bore you with them but will point out important ones as we go along. A more extensive (yet still incomplete) list can be found on the `git config` manual page.

A complete list of all `git` commands is online.

Configuring an alias

Git aliases allow you to substitute common but complex `git` commands that you type frequently with simple and easy-to-remember aliases. This also saves you the hassle of remembering or typing out those long commands, and it saves you from the frustration of running into typos:

```
$ git config --global alias.show-graph \
        'log --graph --abbrev-commit --pretty=oneline'
```

In this example, we created the `show-graph` alias and made it available for use in any repository we create. When we use the command `git show-graph`, it will give us the same output we got when we typed that long `git log` command with all those options.

Summary

You will surely have a lot of unanswered questions about how Git works, even after everything you've learned so far. For instance, how does Git store each version of a file? What really makes up a commit? Where did those funny commit numbers come from? Why the name "main"? And is a "branch" what I *think* it is? These are good questions. What we covered gives you a glimpse of the operations you will commonly use in your projects. The answer to your questions will be explained in detail in Part II.

The next chapter defines some terminology, introduces some Git concepts, and establishes a foundation for the lessons found in the rest of the book.

CHAPTER 2

Foundational Concepts

In the previous chapter, you learned the foundations of Git, its characteristics, and typical applications of version control. It probably sparked your curiosity and left you with a good number of questions. For instance, how does Git keep track of revisions of the same file at every commit on your local development machine? What are the contents of the hidden *.git* directory, and what is their significance? How is a commit ID generated, why does it look like gibberish, and should you take note of it?

If you have used another version control system, such as SVN or CVS, you may notice that some of the commands described in the preceding chapter seemed familiar. Although Git serves the same function and provides all the operations you expect from a modern version control system, an exception to this notion is that the inner workings and principles of Git differ in some fundamental and surprising ways.

In this chapter, we explore why and how Git differs by examining the key components of its architecture and some important concepts. We will focus on the basics, common terminology, and the relationship between Git objects and how they are utilized, all through the lens of a single repository. The fundamentals you'll learn in this chapter also apply when you're working with multiple interconnected repositories.

Repositories

A Git repository is simply a key–value pair database containing all the information needed to retain and manage the revisions and history of files in a project. A Git repository retains a complete copy of the entire project throughout its lifetime. However, unlike most other version control systems, the Git repository provides not only a complete working copy of all the files stored in the project but also a copy of the repository (key–value pair database) itself with which to work.

Figure 2-1 illustrates this explanation.

```
.
├── my_website
│   └── .git
│       └── Hidden git objects
└── index.html
```

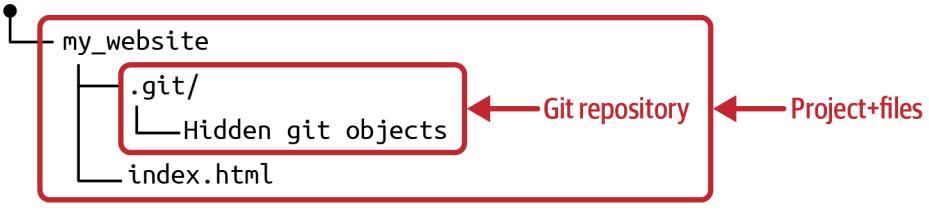

Figure 2-1. Git repository

We use the term *repository* to describe the entire project and its key–value pair database as a single unit.

Besides file data and repository metadata, Git also maintains a set of configuration values within each repository. We already worked with some of these in the previous chapter, specifically, the repository user's name and email address. These configuration settings are not propagated from one repository to another during a clone or duplicating operation. Instead, Git manages and inspects configuration and setup information on a per-environment, per-user, and per-repository basis.

Within a repository, Git maintains two primary data structures: the object store and the index. All of this repository data is stored at the root of your working directory in the hidden subdirectory named *.git*.

The object store is designed to be efficiently copied during a clone operation as part of the mechanism that supports a fully distributed version control system. The index is transitory information, is private to a repository, and can be created or modified on demand as needed.

The next two sections describe the object store and index in more detail.

Git Object Store

At the heart of Git's repository implementation is the object store. It contains your original data files and all the log messages, author information, dates, and other information required to rebuild or restore any version or branch of the project to a specific state in time.

Git places only four types of objects in the object store: blobs, trees, commits, and tags. These four atomic objects form the foundation of Git's higher-level data structures:

Blobs
: Each version of a file is represented as a blob. *Blob*, a contraction of "binary large object," is a term that's commonly used in computing to refer to some variable or file that can contain any data and whose internal structure is ignored by the program. A blob is treated as being opaque. A blob holds a file's data but does not contain any metadata about the file or even its name.

Trees
: A tree object represents one level of directory information. It records blob identifiers, pathnames, and a bit of metadata for all the files in one directory. It can also recursively reference other (sub)tree objects and thus build a complete hierarchy of files and subdirectories. In simple terms, a tree records the contents of a single level in the directory hierarchy. It lists files and subtrees by including their name and an identifier for the Git object they represent (either a blob OID or another tree for subdirectories).

Commits
: A commit object holds metadata for each change introduced into the repository, including the author, committer, commit date, and log message. Each commit points to a tree object that captures, in one complete snapshot, the state of the repository at the time the commit was performed. The initial commit, or root commit, has no parent. Most commits have one commit parent, although later in the book we explain how a commit can reference more than one parent.

Tags
: A tag object assigns a presumably human-readable name to a specific object, usually a commit. `9da581d910c9c4ac93557ca4859e767f5caf5169` refers to an exact, well-defined commit, but a more familiar tag name like `Ver-1.0-Alpha` might make more sense!

The four objects in the object store are immutable. Note that only an *annotated tag* is immutable. A *lightweight tag* is not an immutable object, since it can be used as a reference that we can update directly.

Over time, all the information in the object store changes and grows, tracking and modeling your project edits, additions, and deletions. To use disk space and network bandwidth efficiently, Git compresses and stores the objects in *packfiles*, which are also placed in the object store.

Index

The index stores binary data and is private to your repository. The content of the index is temporary and describes the structure of the entire repository at a specific moment in time. More specifically, it provides a cached representation of all the blob objects that reflects the current state of the project you are working on.

The information in the index is *transitory*, meaning it's a dynamic stage between your project's working directory (filesystem) and the repository's object store (repository commit history). As such, the index is also called the *staging directory*.

Figure 2-2 provides a visual representation of this concept:

```
.
└── repository
    ├── .git/
    │   └── Index
```

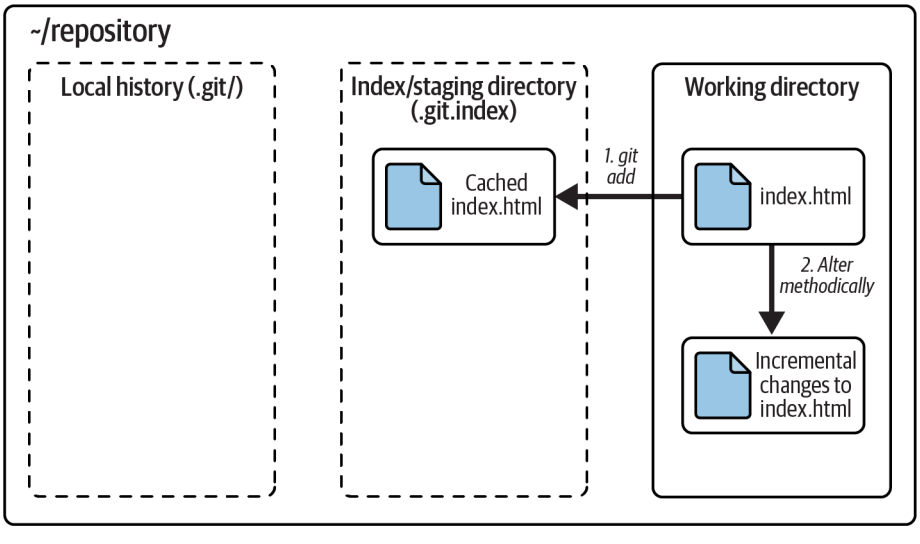

Figure 2-2. Index/staging directory

The index is one of the key distinguishing features of Git. This is because you are able to alter the content of the index methodically, allowing you to have finer control over what content will be stored in the next commit. In short, the index allows a separation between incremental development steps and the committal of those changes.

Here's how it works. As a software engineer, you usually add, delete, or edit a file or a set of files. These are changes that affect the current state of the repository. Next, you will need to execute the `git add` command to stage these changes in the index. Then the index keeps records of those changes and keeps them safe until you are ready to

commit them. Git also allows you to remove changes recorded in the index. Thus the index allows a gradual transition of the repository (curated by you) from an older version to a newer, updated version.

As you'll see in Chapter 6, the index also plays an important role in a merge operation, allowing multiple versions of the same file to be managed, inspected, and manipulated simultaneously.

Content-Addressable Database

Git is also described as a content-addressable storage system. This is because the object store is organized and implemented to store key–value pairs of each object it generates under the hood when you are version-controlling your project. Each object in the object store is associated with a unique name produced by applying SHA1 to the content of the object, yielding a SHA1 hash value.

Git uses the complete content of an object to generate the SHA1 hash value. This hash value is believed to be effectively unique to that particular content at a specific state in time, thus the SHA1 hash is used as a sufficient index or name for that object in Git's object store. Any tiny change to a file causes the SHA1 hash to change, causing the new version of the file to be indexed separately.

SHA1 values are 160-bit values that are usually represented as a 40-digit hexadecimal number, such as `9da581d910c9c4ac93557ca4859e767f5caf5169`. Sometimes, during display, SHA1 values are abbreviated to a smaller, easier-to-reference prefix. Git users use the terms *SHA1*, *hash*, and sometimes *object ID* interchangeably.

Globally Unique Identifiers

An important characteristic of the SHA1 hash computation is that it always computes the same ID for identical content, regardless of *where* that content is. In other words, the same file content in different directories and even on different machines yields the exact same SHA1 hash ID. Thus the SHA1 hash ID of a file is an effective globally unique identifier.

A powerful corollary is that files or blobs of arbitrary size can be compared for equality across the internet by merely comparing their SHA1 identifiers.

We will explore this property in a little more detail in the next section.

Git Tracks Content

Git is much more than a version control system. Based on what we learned earlier, it will help you understand the inner mechanics of Git if you think of Git as a *content tracking system*.

This distinction, however subtle, guides much of the design principle of Git and is perhaps the key reason it can perform internal data manipulations with relative ease, and without compromising performance when done right. Yet this is also perhaps one of the most difficult concepts for new users of Git to grasp, so some exposition is worthwhile.

Git's content tracking is manifested in two critical ways that differ fundamentally from almost all other version control systems:[1]

- First, Git's object store is based on the hashed computation of the *contents* of its objects, not on the file or directory names from the user's original file layout.
- Second, Git's internal database efficiently stores every version of every file, not their differences as files go from one revision to the next.

Let's explore this a little more. When Git places a file into the object store, it does so based on the hash of the data (file content) and not on the name of the file (file metadata). In fact, Git does not track file or directory names, which are associated with files in secondary ways. The data is stored as a blob object in the object store. Again, Git tracks content instead of files.

If two separate files have exactly the same content, whether in the same or different directories, Git stores only a single copy of that content as a blob within the object store. Git computes the hash code of each file only according to its content, determines that the files have the same SHA1 values and thus the same content, and places the blob object in the object store indexed by that SHA1 value. Both files in the project, regardless of where they are located in the user's directory structure, use that same object for content.

Because Git uses the hash of a file's complete content as the name for that file, it must operate on each complete copy of the file. It cannot base its work or its object store entries on only part of the file's content or on the differences between two revisions of that file. Using the earlier example of two separate files having exactly the same content, if one of those files changes, Git computes a new SHA1 for it, determines that it is now a different blob object, and adds the new blob to the object store. The original blob remains in the object store for the unchanged file to use.

For this reason, your typical view of a file that has revisions and appears to progress from one revision to another revision is simply an artifact. Git computes this history as a set of changes between different blobs with varying hashes, rather than storing a filename and a set of differences directly. It may seem odd, but this feature allows Git to perform certain tasks with ease.

1 Monotone, Mercurial, OpenCMS, and Venti are notable exceptions here.

Figure 2-3 provides a visual representation of this concept.

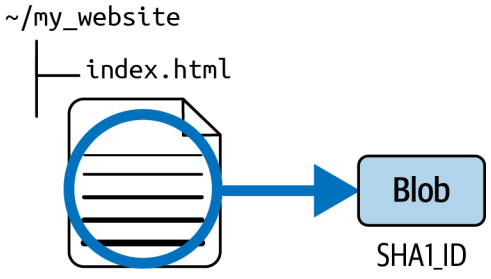

Figure 2-3. Blob object

Pathname Versus Content

As with many other version control systems, Git needs to maintain an explicit list of files that form the content of the repository. However, this need not require that Git's manifest be based on filenames. Indeed, Git treats the name of a file as a piece of data that is distinct from the contents of that file. In this way, it separates index from data in the traditional database sense. It may help to look at Table 2-1, which roughly compares Git to other familiar systems.

Table 2-1. Database comparison

System	Index mechanism	Data store
Relational database	Indexed Sequential Access Method (ISAM)	Data records
Unix filesystem	Directories (*/path/to/file*)	Blocks of data
Git	*.git/objects/`hash`*, tree object contents	Blob objects, tree objects

The names of files and directories come from the underlying filesystem, but Git does not really care about the names. Git merely records each pathname and makes sure it can accurately reproduce the files and directories from its content, which is indexed by a hash value. This set of information is stored in the Git object store as the *tree* object.

Git's physical data layout isn't modeled after the user's file directory structure. Instead, it has a completely different structure that can nonetheless reproduce the user's original file and directory layout in a project. Git's internal structure is a more efficient data structure for its own operations and storage considerations.

When Git needs to create a working directory, it says to the filesystem, "Hey! I have this big blob of data that is supposed to be placed at pathname *path/to/directory/file*. Does that make sense to you?" The filesystem is responsible for saying, "Ah, yes, I recognize that string as a set of subdirectory names, and I know where to place your blob of data! Thanks!"

Figure 2-4 provides a visual representation of this concept.

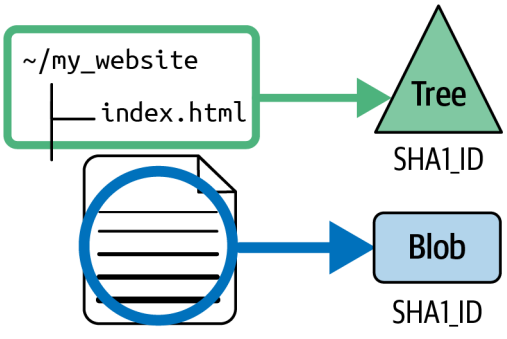

Figure 2-4. Tree object

Packfiles

Next, let's look at how Git stores the blob and tree objects in its object store. If you're following closely, you might think that Git is implementing an inefficient method to store the complete content of every version of every file directly in its object store. Even if Git compresses the files, it is still inefficient to have the complete content of different versions of the same file. For instance, what if we only add, say, one line to a file? Git will still store the complete content of both versions.

Luckily, that's not how Git internally stores the objects in its database. Instead, Git uses a more efficient storage mechanism called *packfiles*. Git uses zlib, a free software library that implements the DEFLATE algorithm to compress each object prior to storing it in its object store. We will dive deeper into packfiles in Chapter 11.

For efficiency, Git's algorithm by design generates deltas against larger objects to be mindful of the space required to save a compressed file. This size optimization is also true for many other delta algorithms because removing data is considered cheaper than adding data in a delta object.

Take note that packfiles are stored in the object store alongside the other objects. They are also used for efficient data transfer of repositories across a network.

Visualizing the Git Object Store

Now that we know how Git efficiently stores its objects, let's discuss how Git objects fit and work together to form a complete system:

- The blob object is at the "bottom" of the data structure; it references no other Git objects and is referenced only by tree objects. It can be considered a leaf node in relation to the tree object. In the figures that follow, each blob is represented by a rectangle.
- Tree objects point to blobs and possibly to other trees as well. Any given tree object might be pointed at by many different commit objects. Each tree is represented by a triangle. In Chapter 15, we will learn how a tree object can also point to a commit object, but for now, we will keep it simple.
- A circle represents a commit. A commit points to one particular tree that is introduced into the repository by the commit.
- Each tag is represented by a parallelogram. Each tag can point to, at most, one commit.

The branch is not a fundamental Git object, yet it plays a crucial role in naming commits. Each branch is pictured as a rectangle:

```
.
└── ~/project
    ├── .git
    │   └── .git/objects/*
    ├── file dead23
    └── file feeb1e
```

Figure 2-5 captures how all the pieces fit together. This diagram shows the state of a repository after a single, initial commit added two files. Both files are in the top-level directory. Both the `main` branch and a tag named `V1.0` point to the commit with ID `1492`.

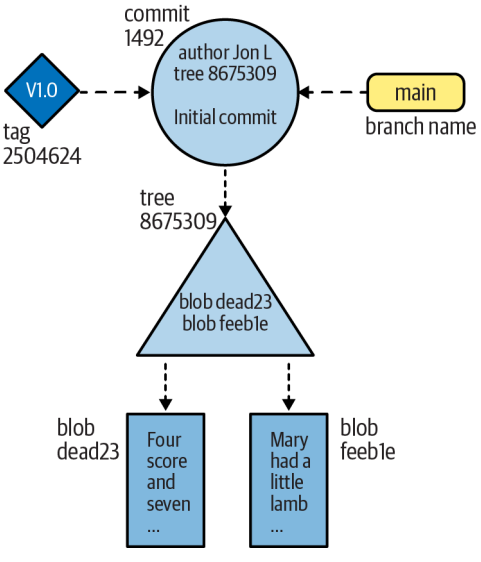

Figure 2-5. Git objects

Now let's make things a bit more complicated. Let's leave the original two files as is, adding a new subdirectory with one file in it. The resulting object store looks like Figure 2-6:

```
.
└── ~/project
    ├── .git
    │   └── .git/objects/*
    ├── file dead23
    ├── file feeb1e
    └── newsubdir
        └── file 1010b
```

As in Figure 2-5, the new commit has added one associated tree object to represent the total state of the directory and file structure. Because the top-level directory is changed by the addition of the new subdirectory, the *content* of the top-level tree object has changed as well, so Git introduces a new tree, `cafed00d`.

However, the blobs `dead23` and `feeb1e` didn't change from the first commit to the second commit. Git realizes that the IDs haven't changed and thus they can be directly referenced and shared by the new `cafed00d` tree.

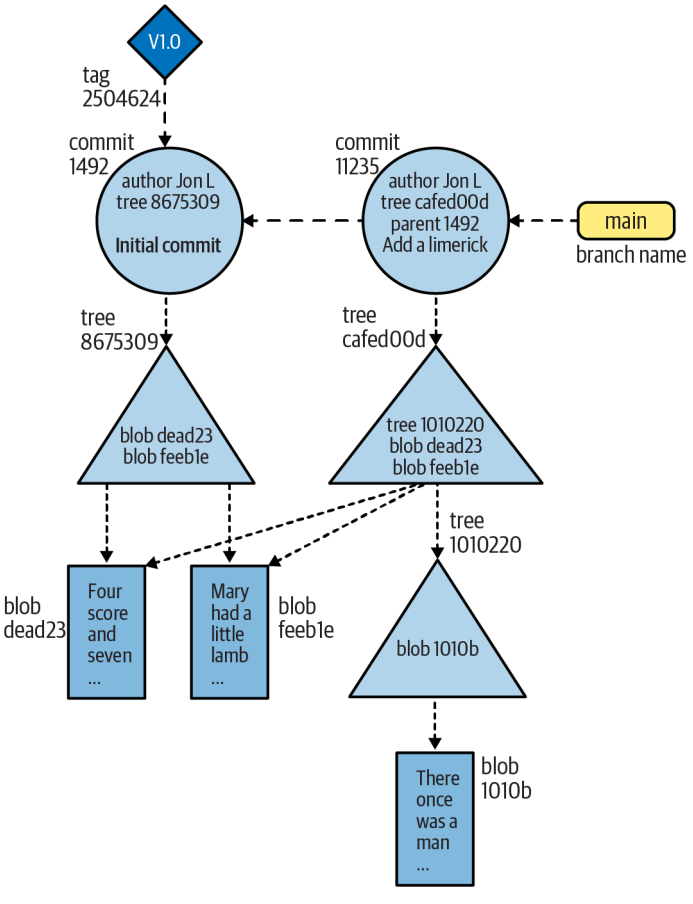

Figure 2-6. Git objects after a second commit

Pay attention to the direction of the arrows between commits. The parent commit or commits come earlier in time. Think of it as a *DAG diagram*: a directed acyclic graph where each node is directed from an earlier node in a single direction to form its topological ordering of the graph.

Therefore, in Git's implementation, each commit points back to its parent or parents. Many people get confused by this because the state of a repository is conventionally portrayed in the opposite direction: as a dataflow *from* the parent commit *to* child commits. In other words, ordered from left to right, the rightmost commit in a DAG diagram represents the latest state of a repository.

In Chapter 4, we extend these pictures to show how the history of a repository is built up and manipulated by various commands.