O'REILLY®

3rd Edition

Version Control with Git

Powerful Tools and Techniques for Collaborative Software Development

Early Release

RAW & UNEDITED

Prem Kumar Ponuthorai
& Jon Loeliger

Version Control with Git

THIRD EDITION

Powerful Tools and Techniques for Collaborative Software Development

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

Prem Kumar Ponuthorai and Jon Loeliger

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 Editor: Virginia Wilson

Production Editor: Beth Kelly

Interior Designer: David Futato

Cover Designer: Karen Montgomery

Illustrator: Kate Dullea

October 2022: Third Edition

Revision History for the Early Release

- 2021-06-22: First Release

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

978-1-492-09112-7

[FILL IN]

Chapter 1. Preface

> **A NOTE FOR EARLY RELEASE READERS**
>
> With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.
>
> This will be the preface of the final book.
>
> If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at vwilson@oreilly.com.

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, and 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 routine 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's

Prior experience with any version control system, their aims and goals will be a helpful foundation to understand how Git works and to build upon as you read this book. You should have some familiarity 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. General understanding of programming concepts is also a plus.

We developed the examples on Mac OS X and Ubuntu Linux environments. The examples should work under other platforms such as Debian, Solaris or Windows Operating System (using git installed command-line tools, eg. Git for Windows), but you can expect slight variations.

When following examples, some exercises may require system level operations which 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 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 visual 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 which 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 each other 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 time in the future.

If you have picked up the book amidst juggling other responsibilities and are wondering what would be the best order to hit the ground running, fret not. The matrix below will help guide you towards the chapters we feel will help you gain the most in the least amount of time.

Table 1-1. Categories Matrix

	Introduction to Git	Thinking in Git	Fundamentals of Git	Intermediate G Commands

Software Engineering	x	x	x	x
Data Scientist		x	x	
Graphic Designers		x	x	
Academia	x	x		
Content Authors	x	x		

> **NOTE**
>
> The categories matrix is provided as a rough guideline

Installing Git

To reinforce the learnings taught in the book, we highly encourage you to practice the example code snippets on your development machine. In order 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 systems, we've covered instructions on how to install Git in *Appendix A* 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 up highly upon ourselves. As a start, we will be using 'main' to indicate the default branch name.

Omissions

Git has evolved over the years. Even as we write this edition, another new

version of Git was published for commercial use, version 2.31.1 to be precise. Due to its active community base, Git is constantly evolving. 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 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 to cover.

Despite these omissions, we feel confident that this book will equip you with a strong foundation and prepare you to dive in deeper in the realms of Git if the need arises.

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.

TIP

This icon signifies a useful hint or a tip.

WARNING

This icon indicates a warning or caution.

NOTE

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 make a copy of a file
$ cp file.txt copy-of-file.txt

# Command to create a new directory
$ mkdir newdirectory

# Command to remove a file
$ rm file

# Command to remove a directory
$ rmdir somedir

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

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

Commands root permissions, commands requiring that need to be executed with root permissions appear as a sudo operation `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, I'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 index.html
```

Using Code Examples

This book is here to help you get your job done. In general, you may use the code in this book in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing a CD-ROM of examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*Version Control with Git* by Jon Loeliger and Matthew McCullough. Copyright 2012 Jon Loeliger, 978-1-449-31638-9."

If you feel your use of code examples falls outside fair use or the permission given previously, feel free to contact us atpermissions@oreilly.com.

Safari® Books Online

NOTE

Safari Books Online (www.safaribooksonline.com) is an on-demand digital library that delivers expertcontent in both book and video form from the world's leading authors in

technology and business.

Technology professionals, software developers, web designers, and business and creative professionals use Safari Books Online as their primary resource for research, problem solving, learning, and certification training.

Safari Books Online offers a range ofproduct mixes and pricing programs fororganizations,government agencies, andindividuals. Subscribers have access to thousands of books, training videos, and prepublication manuscripts in one fully searchable database from publishers like O'Reilly Media, Prentice Hall Professional, Addison-Wesley Professional, Microsoft Press, Sams, Que, Peachpit Press, Focal Press, Cisco Press, John Wiley & Sons, Syngress, Morgan Kaufmann, IBM Redbooks, Packt, Adobe Press, FT Press, Apress, Manning, New Riders, McGraw-Hill, Jones & Bartlett, Course Technology, and dozensmore. For more information about Safari Books Online, please visit usonline.

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:

To comment or ask technical questions about this book, send email to:

bookquestions@oreilly.com

For more information about our books, courses, conferences, and news, see our website at *http://www.oreilly.com*.

Find us on Facebook: *http://facebook.com/oreilly*

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

Watch us on YouTube: *http://www.youtube.com/oreillymedia*

Acknowledgments

Authors, Editors, Tech Reviewers.

Attributions

Chapter 2. Chapter 1: Introduction to Git

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 1st chapter of the final book.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at vwilson@oreilly.com.

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 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 provides 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, it's characteristics, basic git commands and some quick guidance on creating and adding changes to a repository.

We highly recommend you take time to grasp the important concepts explained here. These topics are the building blocks of Git and will help you more easily understand the intermediate and advanced techniques to manage a git repository as part of your daily work routine. These foundational concepts will also help you ramp up your learning when we dissect and break down the inner workings of Git in chapters grouped in Part 2: Fundamentals

of Git, Part 3: Intermediate Skills and Part 4: Advanced Skills.

Git Components

Before we dive into the world of git commands, let’s take a step back and visualize the overview of components that make up the Git ecosystem. Fig. 1-1 below shows how each component works together.

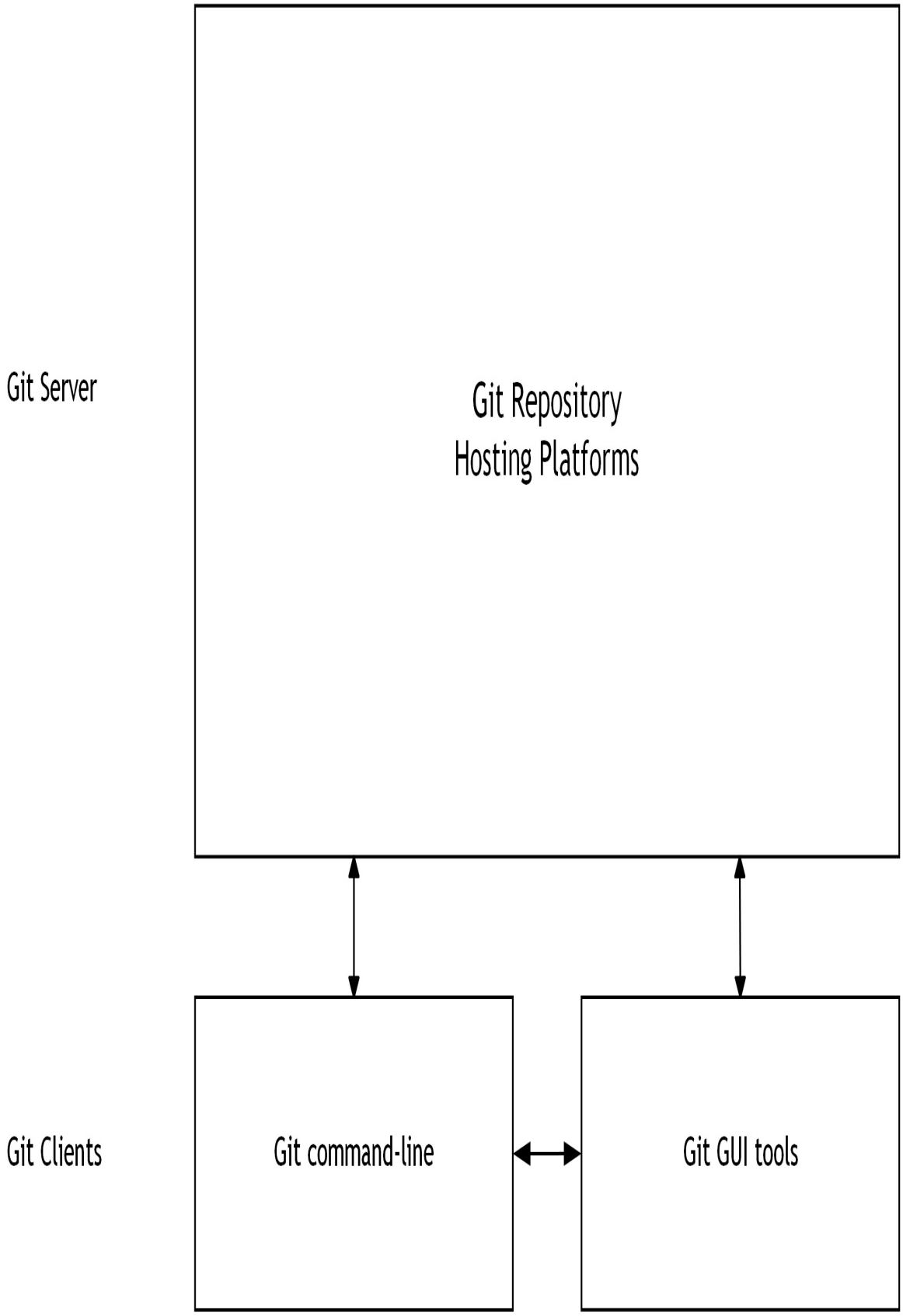

Git Server
Git Repository
Hosting Platforms
Git Clients
Git command-line
Git GUI tools

Figure 2-1. Overview of Git Components

Git GUI tools act as a front end for the git command line and some tools have extensions that integrate with popular Git Hosting Platforms.

When you are working with Git, a typical setup includes a Git Server and a Git Client. 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: Remote Repositories.

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 practise 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 forgo the overhead and opt to host your git repositories on a reliable third party hosting site.

Git Clients

> Git clients are of two types: the git command-line and 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 it, 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 sub-set 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 which may not be part of Git's standard commands. An example would be a tool with an option called "sync", which masks underlying chaining of two or more git commands to achieve a desired outcome. If for some reason you were to type in the "sync" subcommand in 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
```

> **NOTE**
>
> “git sync” is not a valid git subcommand. To ensure 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 is a plethora of tooling available at your disposal. Some Git GUI tools are fancy and extensible via a plugin 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 it builds a good foundational knowledge towards git dexterity.

Git Characteristics

Now that we have covered an overview of the Git Components, let’s learn about the characteristics of Git. When you understand these distinct traits of Git, it 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”.

Stores revision changes as Snapshots

> The very first concept to unlearn is the way git stores multiple revisions on 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 Delta’s; Instead it takes a snapshot of changes to the

state of your repository at a specific point in time. In Git terminology this is known as “commits”. Think of this as capturing a moment in time as a photograph.

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, 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. It waits for you to carry out instructions on what to implement and when to execute 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 revision to your local repo history. Every action requires your explicit command or instruction to tell git what is required, ranging from 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 auto-add new files to be version controlled.

Designed to bolster non linear Development

Git allows you to ideate and experiment with variant implementation of

features for viable solutions to your project by enabling you to diverge and work in parallel along the main stable code base of your project. This methodology, called “Branching”, is a very common practice and ensures the integrity of the main development line from 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 with ease 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 which we will discuss later in Chapter: Merges.

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

The Git Command Line

Git command line is simple to use. Just type `git version` or `git --version`, to know if your machine has already been preloaded with git. You should see an output similar to the following:

```
$ git --version
git version 2.31.1
```

If you do not have git installed on your machine, please refer to *Appendix: Installing Git* 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

git [--version] [--exec-path[=GIT_EXEC_PATH]]
    [-p|--paginate|--no-pager] [--bare] [--git-dir=GIT_DIR]
    [--work-tree=GIT_WORK_TREE] [--help] COMMAND [ARGS]

The most commonly used git commands are:
   add        Add file contents to the index
   bisect     Find the change that introduced a bug by binary search
   branch     List, create, or delete branches
   checkout   Checkout and switch to a branch
   clone      Clone a repository into a new directory
   commit     Record changes to the repository
   diff       Show changes between commits, the commit and working trees, etc
   fetch      Download objects and refs from another repository
   grep       Print lines matching a pattern
   init       Create an empty git repository or reinitialize an existing one
   log        Show commit logs
   merge      Join two or more development histories
   mv         Move or rename a file, a directory, or a symlink
   pull       Fetch from and merge with another repository or a local branch
   push       Update remote refs along with associated objects
   rebase     Forward-port local commits to the updated upstream head
   reset      Reset current HEAD to the specified state
   rm         Remove files from the working tree and from the index
   show       Show various types of objects
   status     Show the working tree status
   tag        Create, list, delete, or verify a tag object signed with GPG
```

TIP

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 2.31.1
```

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>`.

> **NOTE**
>
> You can visit *http://www.kernel.org/pub/software/scm/git/docs/* to read the complete Git documentation online.

Example 2-1.

Historically, Git was provided as a suite of many simple, distinct, standalone commands developed according to the “Unix toolkit” 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.

Git commands understand both “short” and “long” options. For example, the `git commit` command treats the following examples as equivalents.

```
$ git commit -m "Fixed a typo."
```

```
$ git commit --message="Fixed a typo."
```

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

> **TIP**
>
> You can create a commit summary and detailed message for the summary by using the `-m` option consecutively:
>
> ```
> $ 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 file names if they might otherwise be 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 get different behavior:

```
# 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. It is much like you setting up the correct date, timezone and language on a new camera before taking your first snapshot.

Configuring the Commit Author

At a bare minimum , Git requires your name and email address before you make a first commit in your repository. The identity you supply then shows as the commit *author*, baked in together 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 following argument `--author` at the end of the command:

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

Keep in mind 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 machines. We will start simple and work our way towards 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`:

```
$ git init -b main

Initialized empty Git repository in .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 .git/

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

TIP

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 is stored in this hidden single top level `.git` folder.

Git considers the *~/my_website* as 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 named as `Index` and `Local History` to illustrate the concept of initializing a new Git repository. We will discuss the `index` and `Local History` in Chapters *File Management and the Index* and *Commits* respectively.

Figure 1-2 will help you visualize what we have just explained:

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

~/my_website

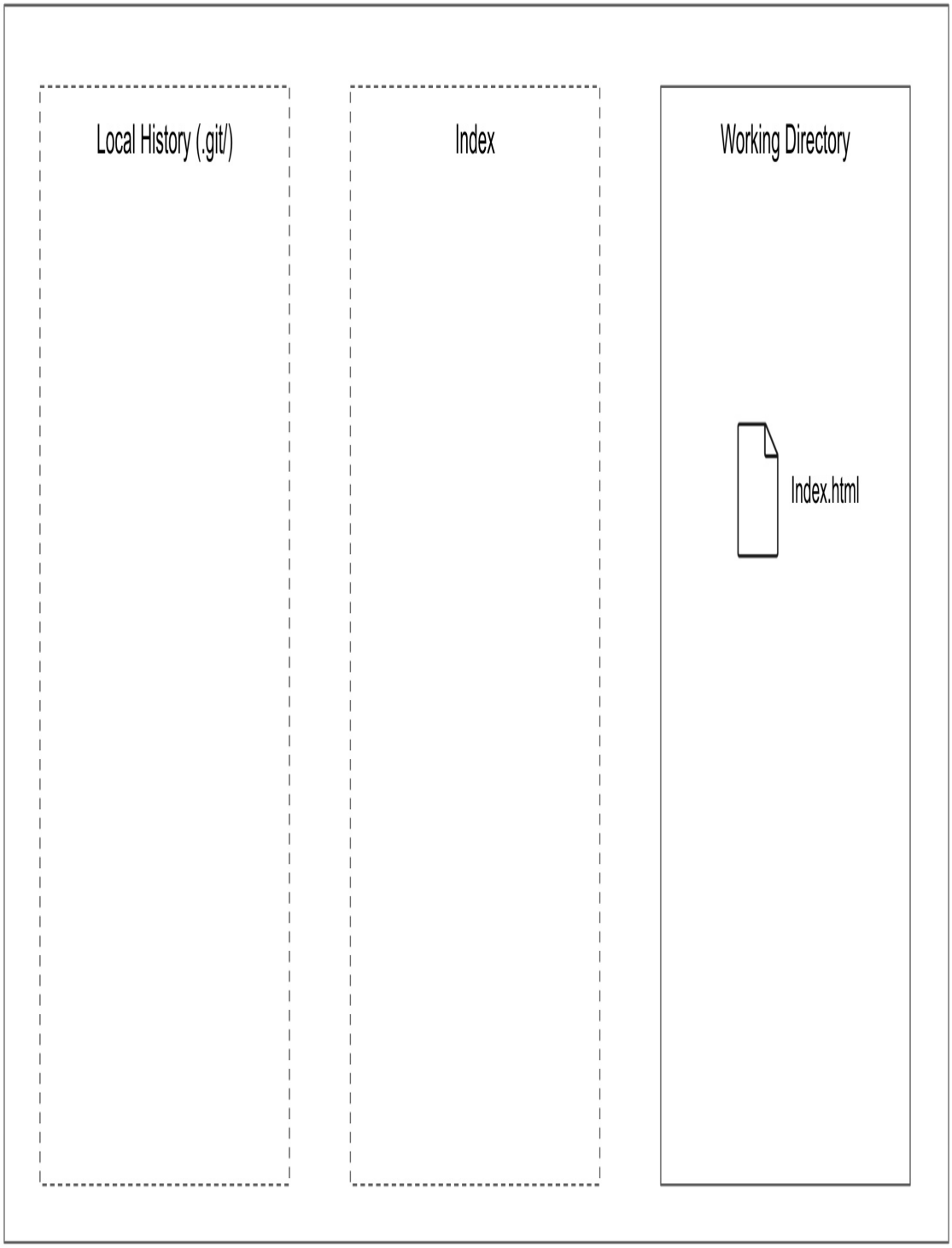


Figure 2-2. Initial Repository Visual

The dotted lines for the `Index` and `Local History` 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 to the git repository, you need to explicitly do so by using the `git add _file_` command:

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

> **WARNING**
>
> Although you can let Git add all the files in the directory and all subdirectories using the `git add .` command, this can be a dangerous habit since it could lead to sensitive information or unwanted files being included when commits are made. To avoid including such information, you can use the `.gitignore` file which is covered in section *The .gitignore File* in *Chapter: File Management and the Index.*
>
> 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 the *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. 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*', keeping the

repository in a stable, consistent state. This method also allows for us to craft a narrative of why we are changing the code. In *Chapter: Commits* we will dive deeper on this concept.

We recommend that you strive to group logical changes as a *'batch*' before making a commit. This is called an `atomic` commit and this will help you along the way in situations where you need to do some advanced git operations discussed in later chapters.

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

```
$ git status

# On branch main
#
# Initial commit
#
# 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*. Everytime you make a commit, git records several other 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 through 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 *Chapter: Commits* and *Chapter: Altering Commits Rewriting History.*

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

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

Created initial commit 9da581d: Initial contents of my_website
 1 files changed, 1 insertions(+), 0 deletions(-)
 create mode 100644 index.html
```

> **NOTE**
>
> 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 favourite editor during a `git commit`(leave out the *'-m*' argument); Set the GIT_EDITOR environment variable as follows:

```
# In tcsh
$ setenv GIT_EDITOR emacs

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

> **NOTE**
>
> As a default, Git will honour the default text editor configured in the shell environment variables VISUAL or EDITOR. If neither are configured, it falls back to use the `vi` editor

After you commit the *index.html* 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 directory 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:

~/my_website

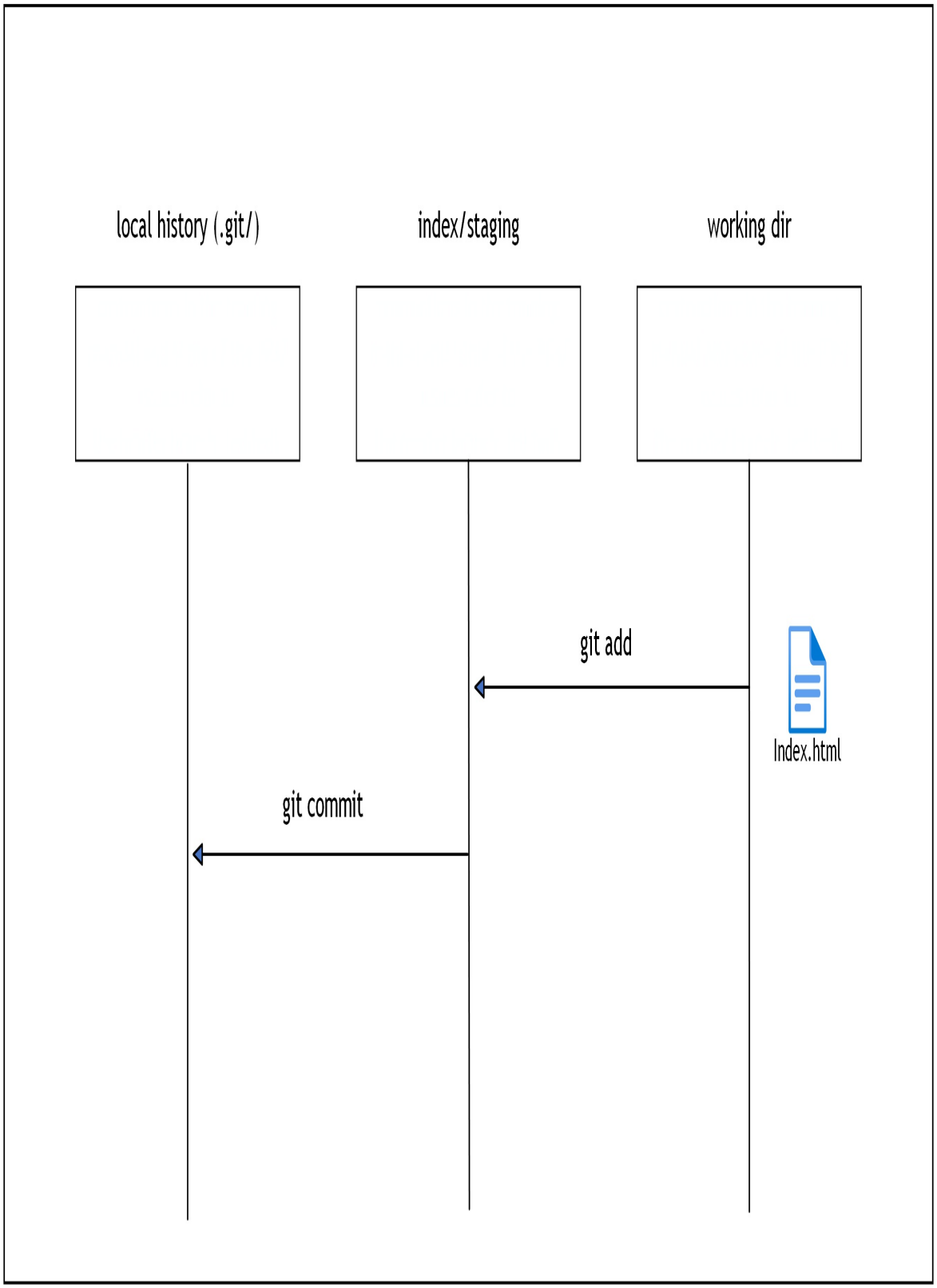

local history (.git/)
index/staging
working dir
git add
Index.html
git commit

Figure 2-3. Staging and adding a file to a repository

The Difference between the `git add` and `git commit` is much like you organizing a group of school children in a prefered order to get the perfect classroom photograph. `git add` does the organizing, whereas `git commit`takes the snapshot.

Making Another Commit

Next, let's make a few modifications to the *index.html* and create a *repo history* within the repository. Convert the *index.html* in to a proper HTML and commit the alteration to it:

```
$ cd ~/my_website

# edit the index.html file.

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

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

If you are already familiar with Git, you may be tempted to wonder 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

git commit [-a | --interactive | --patch] [-s] [-v] [-u<mode>] [--amend]
           [--dry-run] [(-c | -C | --fixup | --squash) <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]]
           [-S[<keyid>]] [--] [<pathspec>...]
```

...

> **TIP**
>
> Detailed explanation of the various commit methods are also explained in `git commit --help` manual pages

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

~/my_website

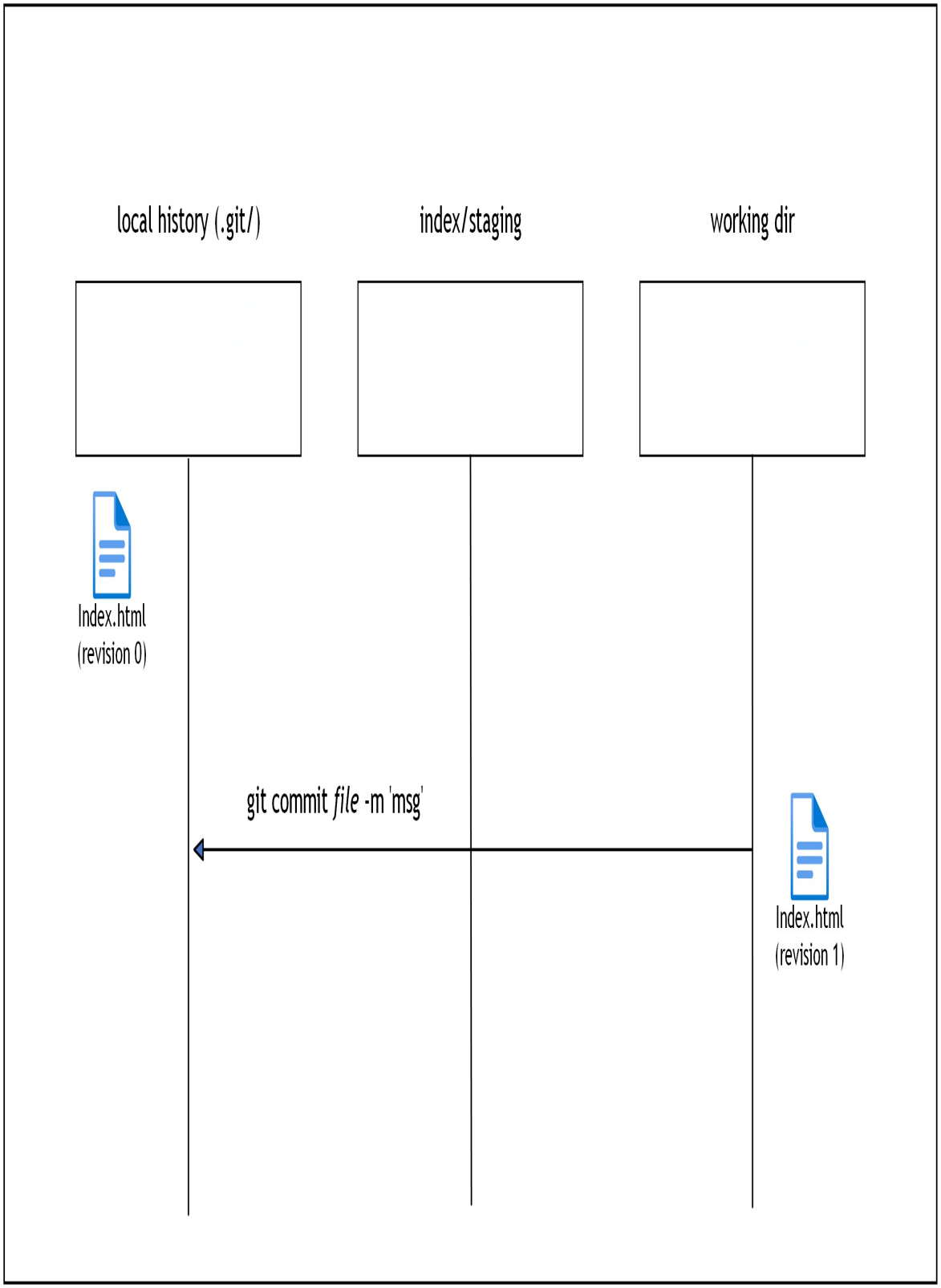

local history (.git/)
index/staging
working dir
Index.html
(revision 0)
git commit *file* -m 'msg'
Index.html
(revision 1)

Figure 2-4. Staging and adding a file to a repository

Note that our usage of `git commit index.html -m 'Convert to HTML'` does not skip the staging of the file, it's just that 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 ec232cddfb94e0dfd5b5855af8ded7f5eb5c90d6
Author: Jon Loeliger <jdl@example.com>
Date:   Wed Apr 2 16:47:42 2021 -0500

Convert to HTML

commit 9da581d910c9c4ac93557ca4859e767f5caf5169
Author: Jon Loeliger <jdl@example.com>
Date:   Thu Mar 13 22:38:13 2021 -0500

Initial contents of my_website
```

In the above output, the `git log` prints out detailed log information for every commit in the repository. At this point you only have two commits in your *repo history* which makes it easier to read the output easily. For repositories with many commit histories, this standard view may not help you to 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

ec232cd (HEAD -> main) Convert to HTML
```

```
9da581d 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 section Content Addressable Names of the Basic Concepts chapter, and we will discuss more about commits in the Commits chapter.

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

```
$ git show 9da581d910c9c4ac93557ca4859e767f5caf5169

commit 9da581d910c9c4ac93557ca4859e767f5caf5169
Author: Jon Loeliger <jdl@example.com>
Date:   Thu Mar 13 22:38:13 2021 -0500

Initial contents of public_html

diff --git a/index.html b/index.html
new file mode 100644
index 0000000..34217e9
--- /dev/null
+++ b/index.html
@@ -0,0 +1 @@
+My web site is awesome!
```

TIP

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` 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 more 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: Branches*.

Viewing Commit Differences

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

```
$ git diff 9da581d910c9c4ac93557ca4859e767f5caf5169 \
            ec232cddfb94e0dfd5b5855af8ded7f5eb5c90d6

diff --git a/index.html b/index.html
index 34217e9..8638631 100644
--- a/index.html
+++ b/index.html
@@ -1 +1,5 @@
-My awesome website!
+<html>
+<body>
 My web site is awesome!
+</body>
+</html>
```

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

> **TIP**
>
> Do not be intimidated by the long hex numbers. Git provides many shorter, easier alternative ways to run similar commands so 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 is sufficient. We elaborate more on this in section *Content Addressable Database* of *Chatper: Basic Git Concepts.*

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"
Created commit 364a708: Remove adverts html
 0 files 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 the `git rm` which also *stages* the file. Realize the change in the repository with a `git commit`.

You can rename a file indirectly by using a combination of `git rm` and `git add` command, or you can rename it 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 lest `git rm` permanently delete 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 be committed subsequently:

```
$ git commit -m "Moved foo to bar"
Created commit 8805821: Moved foo to bar
 1 files changed, 0 insertions(+), 0 deletions(-)
 rename foo.html => bar.html (100%)
```

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

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 exclusively available to your local development machine. It 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 purpose 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 was 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:

```
$ 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 `cp -a` or `rsync`. In contrast, if you are to *clone* the same repository from a git server, the syntax will 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 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 visualizes this concept:

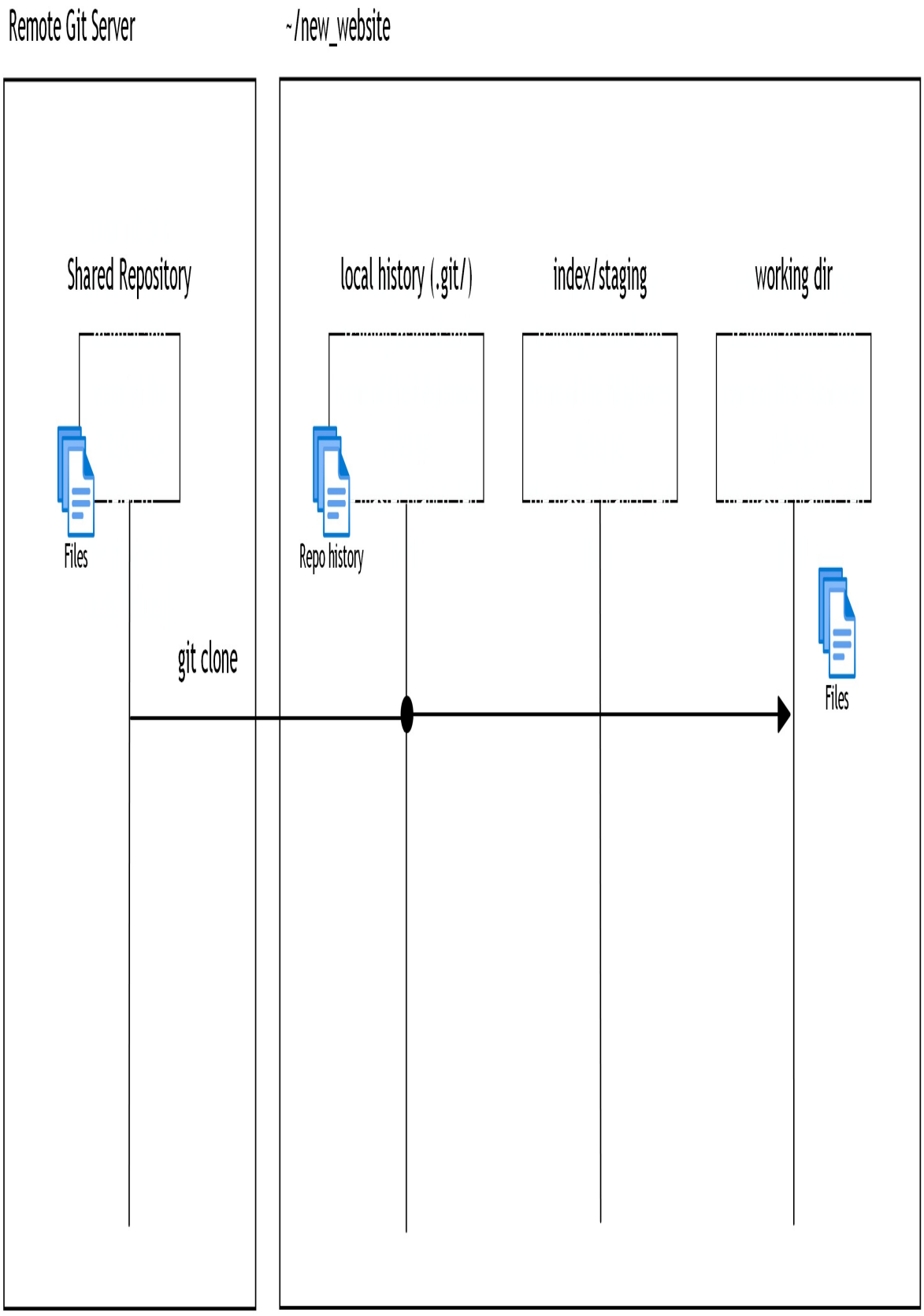

Remote Git Server
~/new_website
Shared Repository
local history (.git/)
index/staging
working dir
Files
Repo history
git clone
Files

Figure 2-5. Cloning a shared repository

Try not to be distracted with some of the terms you see on 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: Remote Repositories.*

Configuration Files

Git configuration files are all simple text files in the style of *.ini* files. The configuration files are used to store preference and settings used by multiple git commands. Some of the settings represent personal preferences (eg. should a `color.pager` be used?); others are important to a repository functioning correctly (eg. `core repositoryformatversion`); and still others tweak git command behavior a bit (eg. `gc.auto`). Like other tools, git supports a hierarchy of configuration files.

Hierarchy of Configuration files

Here are the settings, Figure 1-6 represents the git configuration files hierarchy in decreasing precedence:

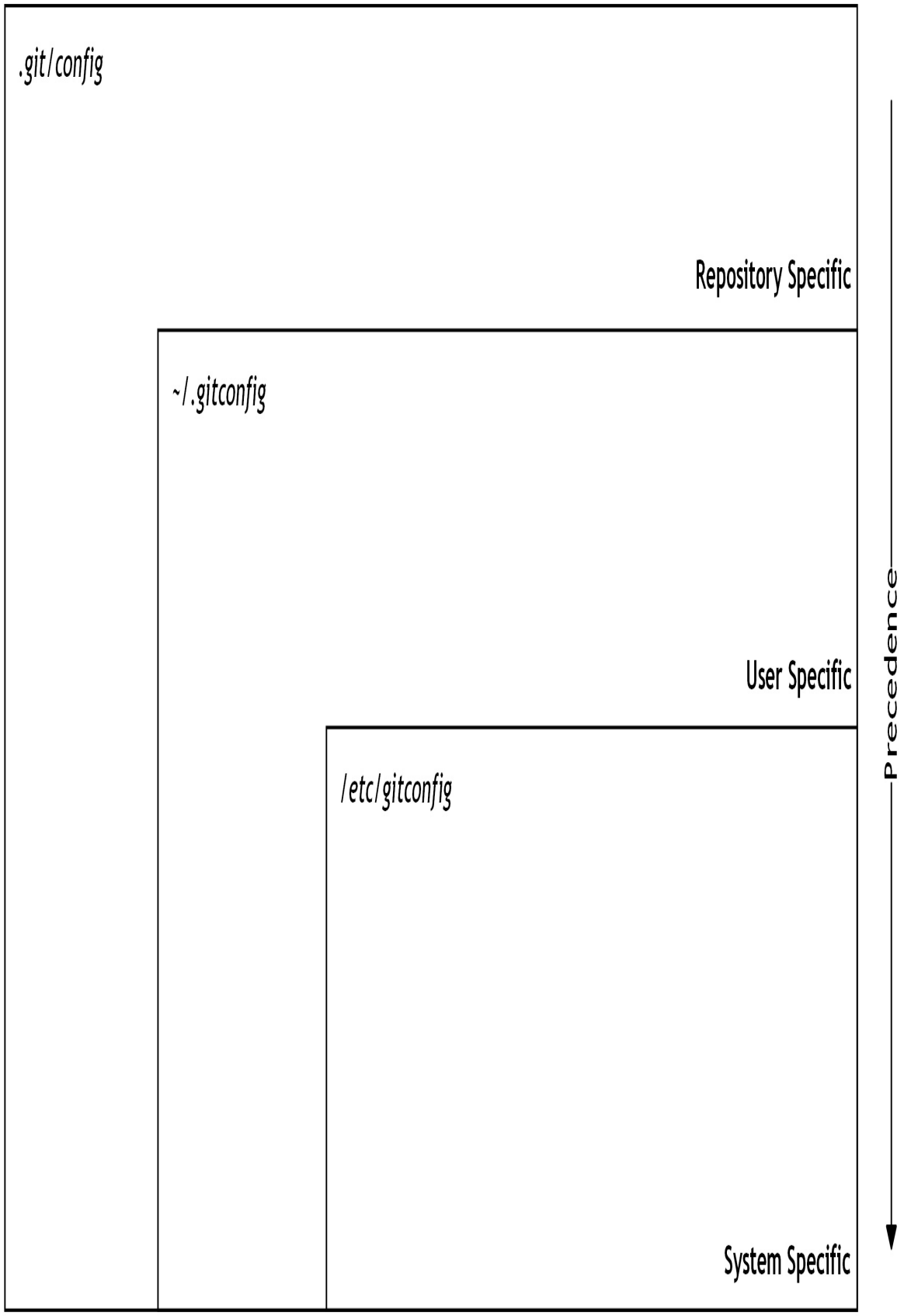

.git/config
Repository Specific
~/.gitconfig
User Specific
/etc/gitconfig
System Specific
Precedence

Figure 2-6. Git Configuration files hierarchy

.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 it. These settings have the lowest precedence. Depending on your actual installation, the system settings file might be somewhere else (perhaps in `/usr/local/etc gitconfig`), or may be entirely absent.

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 may use the `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
```

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

NOTE

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 file system characteristics.

In case 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 may set a value for the editor to be used when composing a commit log message. Based on configuration, invocation follows these steps, in order:

- `GIT_EDITOR` environment variable
- `core.editor` configuration option
- `VISUAL` environment variable
- `EDITOR` environment variable
- 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.

> **TIP**
>
> The following website: *https://git-scm.com/docs* also contains reference to a complete list of all git commands on the internet as an alternative source of reference manual.

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 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've made up the `show-graph` alias and made it available for use in any repository we create. When we use the command `git show-graph`, it is going to give an output, just like when we had 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 on the operations you will be commonly using on your projects. The answer to your questions will be discussed in detail in chapters under *Part 2: Fundamentals of Git.*

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

---

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

Chapter 3. Foundational Concepts

> **A NOTE FOR EARLY RELEASE READERS**
>
> With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.
>
> This will be the 2nd chapter of the final book.
>
> If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at vwilson@oreilly.com.

In the previous chapter you learned the foundations of Git, its characteristics and typical application of version controlling your projects. It probably sparked your curiosity and left you with a good number of questions. For instance, how does git keep track revisions of the same file at every commit on your local development machine? What are the contents of the hidden *.git* directory and its significance? How is a commit ID generated and why does it look 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 last chapter likely 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 will be 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 terminologies, the relation between git objects and how they are utilized, all through the lens of a single repository. The fundamentals you learn in this chapter apply just the same, when you will work 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 not only provides 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 helps illustrate the explanation:

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

> **NOTE**
>
> 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 have already worked with some of these in the previous chapter, in specific 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: the 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, path names, 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.

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 an arbitrary yet presumably human readable name to a specific object, usually a commit. Although `9da581d910c9c4ac93557ca4859e767f5caf5169` refers to an exact and well-defined commit, a more familiar tag name like `Ver-1.0-Alpha` might make more sense!

> **NOTE**
>
> These four objects are immutable. Note that only an *annotated tag* is immutable, a *lightweight tag* is not

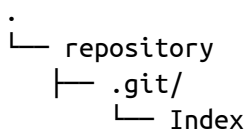
Over time, all the information in the object store changes and grows, tracking and modeling your project edits, additions, and deletions. To use disk space and network bandwidth efficiently, Git compresses and stores the objects in packfiles, which are also placed in the object store.

Index

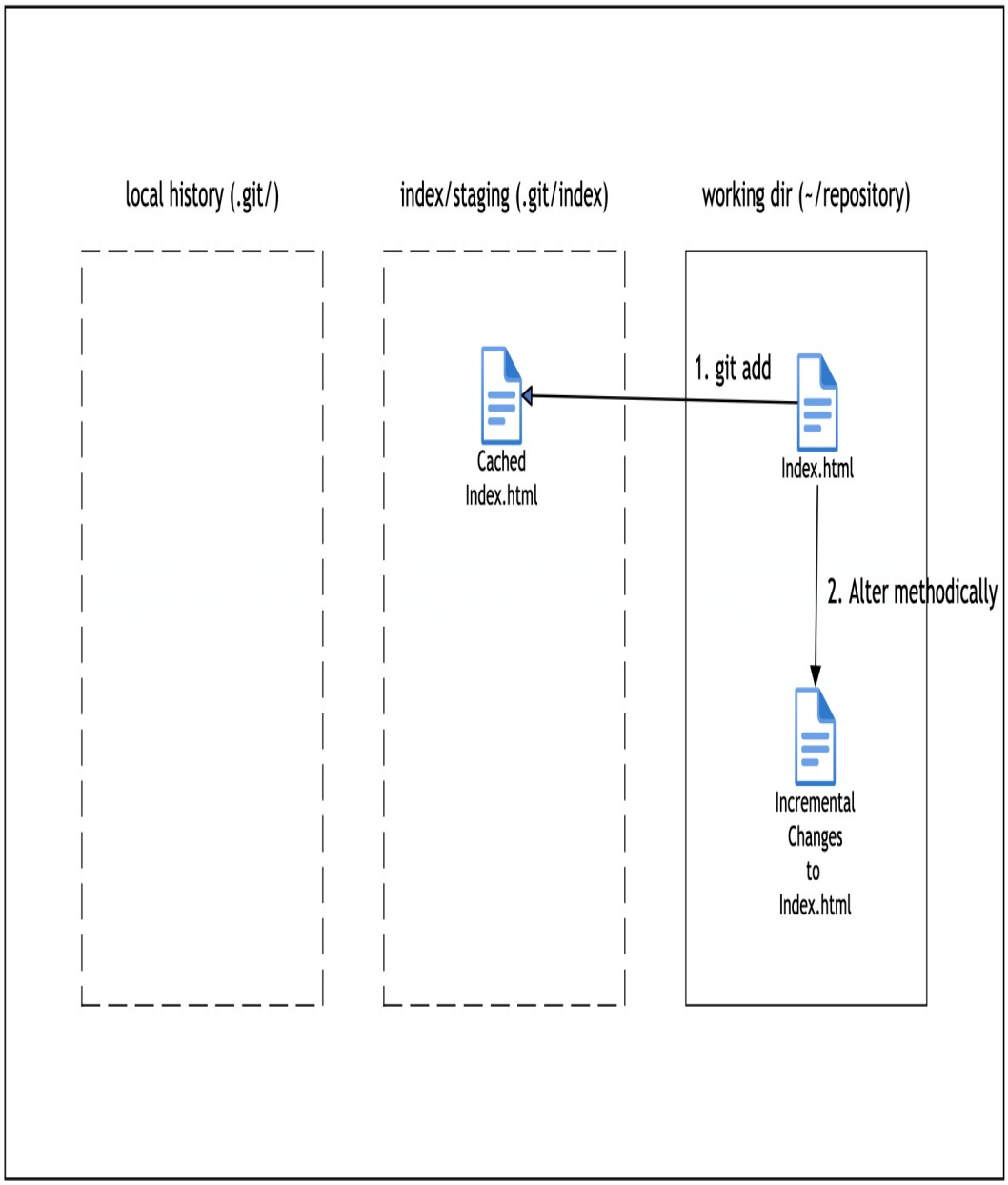
The index stores binary data and is private to your repository. The content of the index is temporary and describes the structure of the entire repository at a specific moment in time. More specifically, it provides a cached representation of all the blob objects which reflects the current state of the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

The information in the index is transitory, meaning it’s a dynamic stage between your project’s working directory (file system) and the repository’s object store (repository commit history). As such the index is also labeled as the “Staging Directory” interchangeably.

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



~/repository


local history (.git/)
index/staging (.git/index)
working dir (~/repository)
1. git add
Cached
Index.html
Index.html
2. Alter methodically
Incremental
Changes
to
Index.html

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: Merges*, 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 speak of the term SHA1, hash code, 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.

Git Tracks Content

Git is much more than a Version Control System, based on what we learnt in the earlier section, it will help to understand the inner mechanics of Git when 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 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[1] version control systems.

- 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 only stores a single copy of that content as a blob within the object store. Git computes the hash code of each file according solely 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 nor 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 it 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 file name and set of differences directly. It may seem odd, but this feature allows Git to perform certain tasks with ease.

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

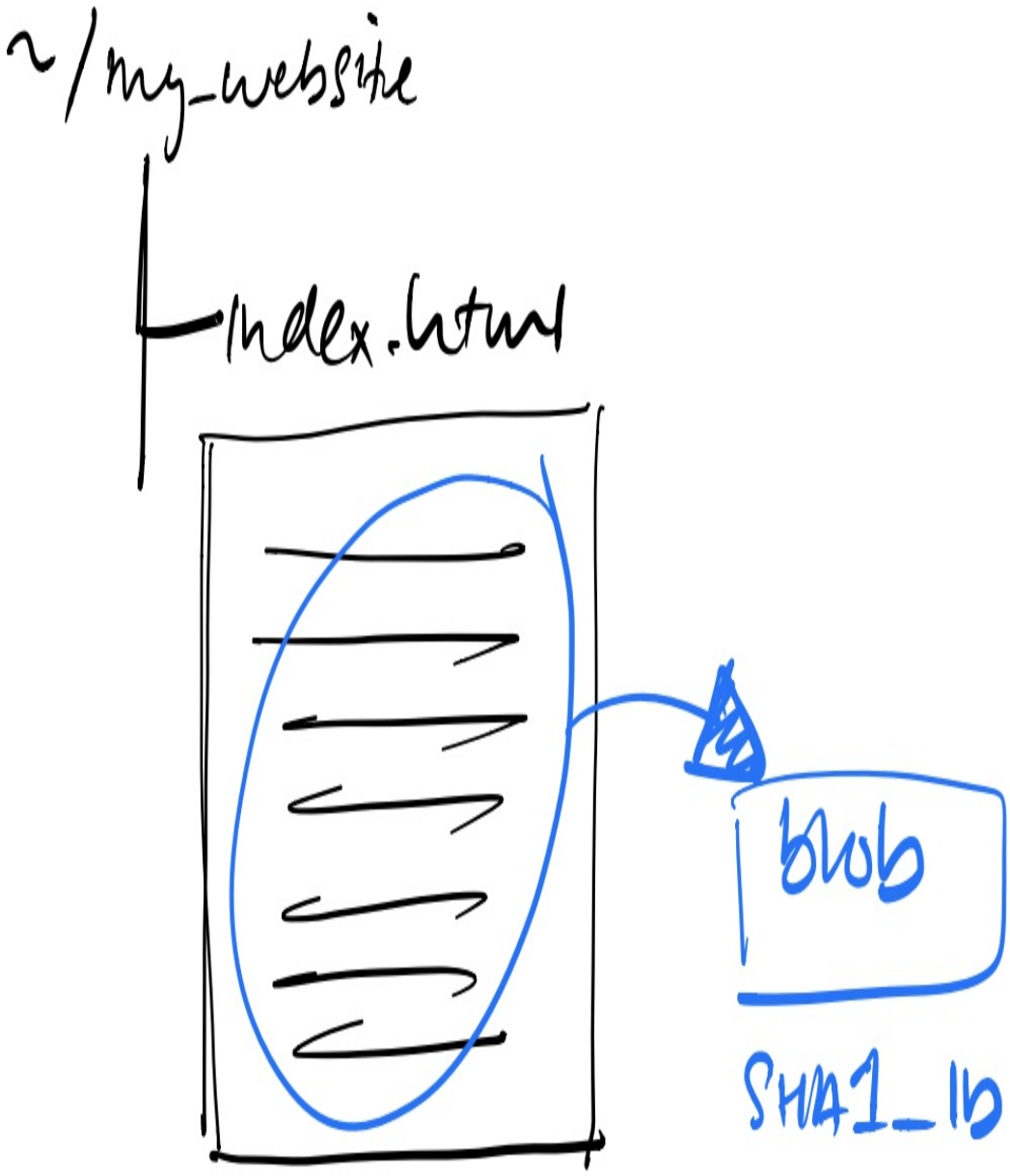


Figure 3-1. Blob Object

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 file names. 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 1.Table 3-1, which roughly compares Git to other familiar systems.

Table 3-1. Database comparison

System	Index mechanism	Data store
Relational database	Indexed Sequential Access Method (ISAM)	Data records
Unix file system	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 internal 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:

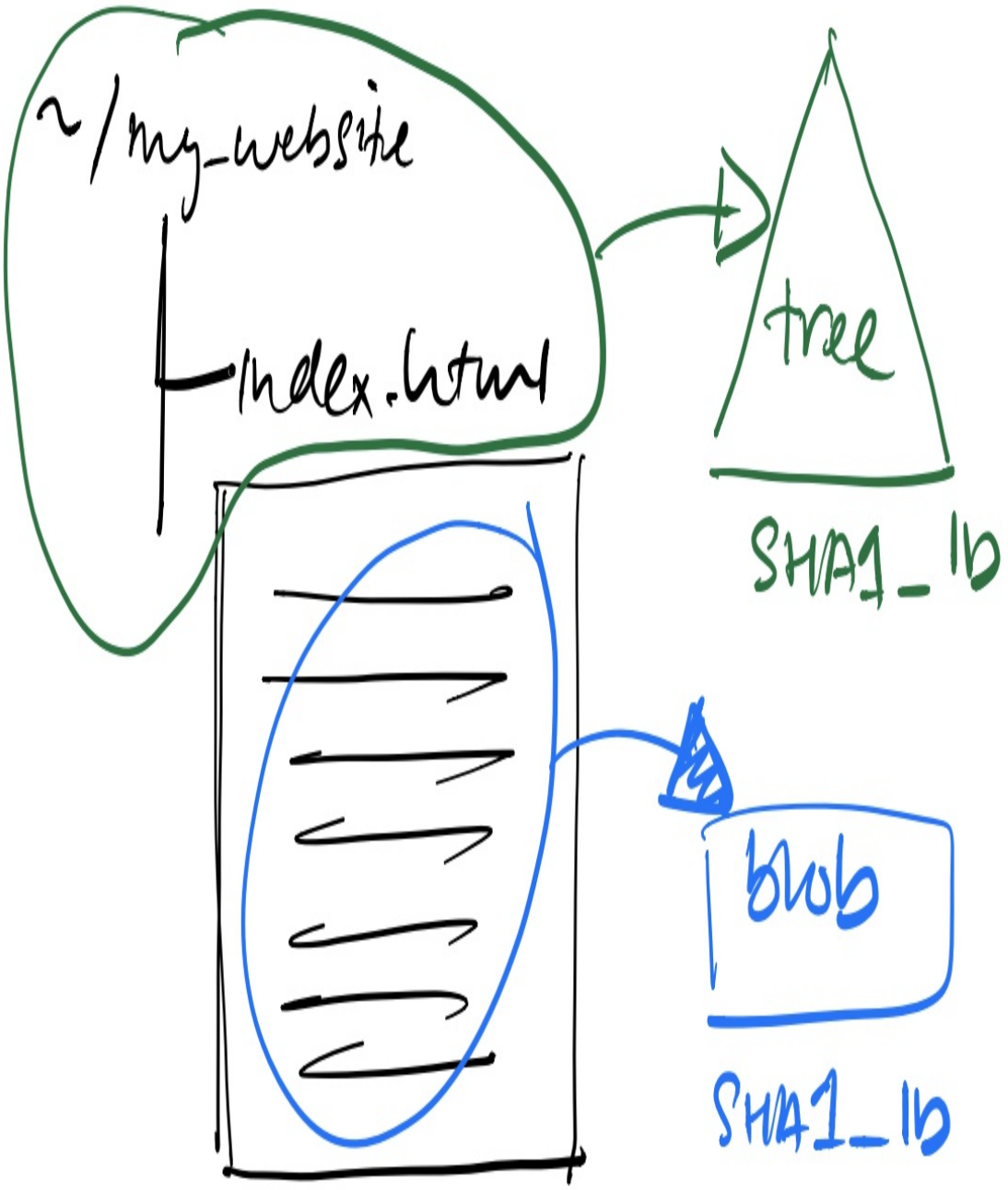


Figure 3-2. Tree Object

Packfiles

Next, let's look at how Git stores the *blob* and *tree* objects in it's object store. Also 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 it's 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 it's database. Instead, Git uses a more efficient storage mechanism called *packfiles*. Git uses zlib[2] a free software which implements the *DEFLATE* algorithm[3] to compress each object prior to storing it in it's object store. We will be diving deeper into *packfiles* in *Chapter: Remote Repositories.*

> **TIP**
>
> For efficiency, Git's algorithm by design generates deltas against larger object to be mindful of the space it takes up 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 Git Object Store

Now that we know how Git efficiently stores it's objects, let's visualize 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 as 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.
- 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 rounded rectangle.

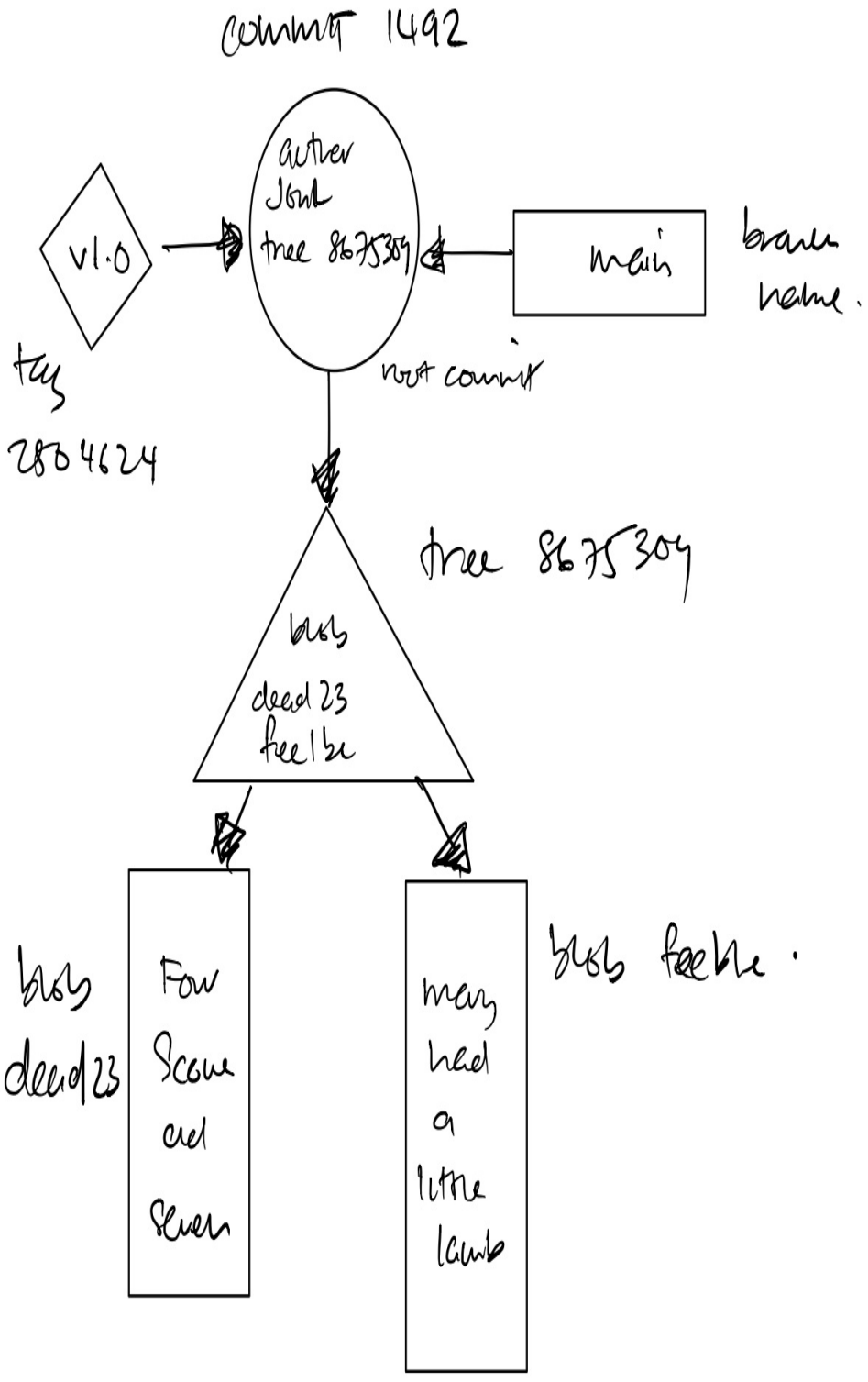

commit 1492
author
Josh
tree 8675309
v1.0
main
branch name.
tag
2804624
root commit
tree 8675309
blob
deed 23
feelbe
blob
deed 23
Four
Score
and
Seven
many
head
a
little
lamb
blob feelbe.

Figure 3-3. Git objects

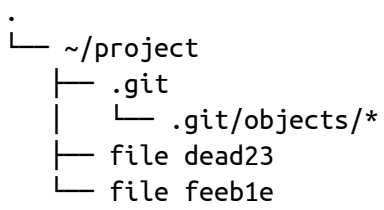
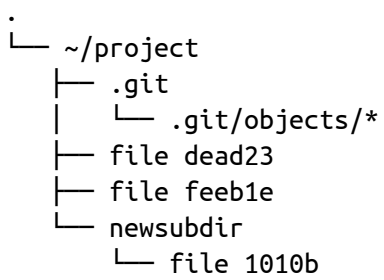
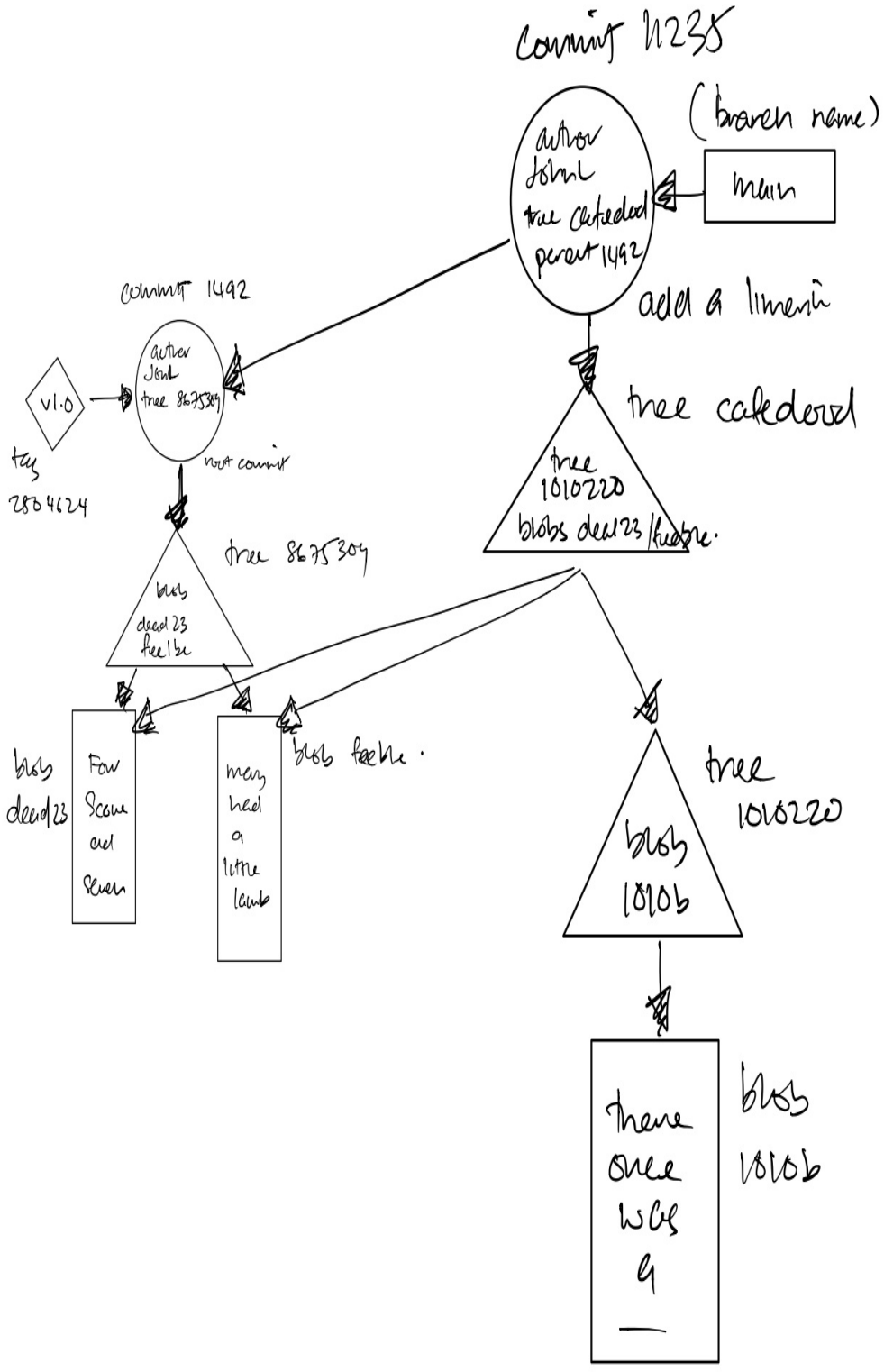


Figure 2-5 captures how all the pieces fit together. This diagram shows the state of a repository after a single, initial commit added two files. Both files are in the top-level directory. Both the `master` branch and a tag named `V1.0` point to the commit with ID `1492`.

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 below.




Commit 1235
(braven name)
author
commit
tree cafedead
parent 1492
main
add a line in
commit 1492
author
commit
tree 8675309
v1.0
tag
2804624
next commit
tree cafedead
tree
1010220
blobs dead23 feedbe.
tree 8675309
blob
dead23
feedbe
blob dead23
Fow
Scow
ed
Seven
many
had
a
little
lamb
blob feedbe.
tree
1010220
blob
1010b
there
once
was
a
—
blob
1010b

As in the previous picture, the new commit has added one associated tree object to represent the total state of 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. Git realizes that the IDs haven't changed and thus can be directly referenced and shared by the new `cafed00d` tree.

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 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 right most commit in *DAG diagram* represent the latest state of a repository.

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

Git Internals: Concepts at Work

With some tenets out of the way, let's peek under the hood and see how all these concepts fit together in a git repository. We will start by creating a new repository and inspect the internal files and object store in much greater detail. We do this by starting at the bottom of Git's data structure and work our way up in the object store.

Before we go any further, it is important to know that Git has a few categories of commands to implement it's inner mechanics. To get a detailed categorized list of all the commands, type in `git help -a` in your terminal.

Git commands are categorized as follows :

- Main Porcelain Commands (High level commands for routine Git operations)
- Ancillary Commands (Commands that help query Git's internal data store)
- Low-level Commands (Plumbing Commands for internal Git Operations)
- External Commands (Commands that extent the standard Git Operations)
- Commands to act as a bridge with selected version control tool (Interacting with Others Commands)
- Command Aliases (Custom aliases created by users to mask complex Git commands)

Typically, for our daily use and interaction with Git, we will mostly use a subset of the main porcelain commands. In this section, we will be using some low-level or plumbing commands to better understand Git Internals.

Inside the .git Directory

To begin, initialize an empty repository using `git init` and then run the `tree .git` command to reveal what's created.

```
$ mkdir /tmp/hello
$ cd /tmp/hello
$ git init
Initialized empty Git repository in /tmp/hello/.git/

# List all the files in the current directory
$ tree .git
     .git
 ├── HEAD
 ├── config
 ├── description
 ├── hooks
```

```
│   ├── applypatch-msg.sample
│   ├── commit-msg.sample
│   ├── fsmonitor-watchman.sample
│   ├── post-update.sample
│   ├── pre-applypatch.sample
│   ├── pre-commit.sample
│   ├── pre-merge-commit.sample
│   ├── pre-push.sample
│   ├── pre-rebase.sample
│   ├── pre-receive.sample
│   ├── prepare-commit-msg.sample
│   ├── push-to-checkout.sample
│   └── update.sample
├── info
│   └── exclude
├── objects
│   ├── info
│   └── pack
└── refs
    ├── heads
    └── tags
```

As you can see, *.git* contains a lot of stuff. The files are displayed based on a template directory that you can adjust if desired by passing in the `--template=<template_directory>` option. For example, if you prefer to create a new repository which implements custom git-hooks, you may point to a template which is already preconfigured with custom directory structure and git-hook files to begin with. We will discuss more on git-hooks in *Chapter: Hooks.*

> **NOTE**
>
> Depending on the version of Git you are using, your actual manifest may look a little different. For example, older versions of Git do not use a *.sample* suffix on the *.git/hooks* files. You can learn more about the command by running `man git-init` in the command-line.

In general, you don't have to view or manipulate the files in *.git* directory. These "hidden" files are considered part of Git's plumbing or configuration

commands.

Initially, the *.git/objects* directory (the directory for all of Git's objects) is empty, except for a few placeholders.

```
$ find .git/objects

.git/objects
.git/objects/pack
.git/objects/info
```

Let's now carefully create a simple object:

```
$ echo "hello world" > hello.txt
$ git add hello.txt
```

If you typed "hello world" exactly as it appears here (with no changes to spacing or capitalization), then your objects directory should now look like this:

```
$ find .git/objects
.git/objects
.git/objects/3b
.git/objects/3b/18e512dba79e4c8300dd08aeb37f8e728b8dad
.git/objects/pack
.git/objects/info
```

Note that there is only one object at this point in time, the blob object with a SHA1 ID generated based on content of the file hello.txt. All this looks pretty mysterious. But it's not, as the following sections explain.

Blob Objects and Hashes

When we created the file *hello.txt* and staged it in the *index* directory using `git add`, git internally created a *blob* object. At this point, Git doesn't care that the filename is *hello.txt*. Git cares only about what's inside the file: the sequence of 12 bytes that represent "hello world" and the terminating newline (the same blob created earlier). Git performs a few operations on this blob,

calculates its SHA1 hash, and enters it into the object store as a file named after the hexadecimal representation of the hash.

The hash in this case is `3b18e512dba79e4c8300dd08aeb37f8e728b8dad`. The 160 bits of an SHA1 hash correspond to 20 bytes, which takes 40 bytes of hexadecimal to display, so the content is stored as *.git/objects/3b/18e512dba79e4c8300dd08aeb37f8e728b8dad*.

Git inserts a / after the first two digits to improve filesystem efficiency. (Some filesystems slow down if you put too many files in the same directory; making the first byte of the SHA1 into a directory is an easy way to create a fixed, 256-way partitioning of the namespace for all possible objects with an even distribution.)

You can verify that the content in the file is not changed by Git (it's still the same comforting "hello world") by using the generated hash value to extract out the content from the object store, utilizing a *low-level plumbing* command.

```
# Using the git cat-file command
$ git cat-file -p 3b18e512dba79e4c8300dd08aeb37f8e728b8dad
hello world
```

Or

```
# Using the git hash-object command
$ echo "hello world" | git hash-object --stdin
3b18e512dba79e4c8300dd08aeb37f8e728b8dad
```

TIP

Git also knows that 40 characters is a bit chancy to type by hand, so it provides a command to look up objects by a unique prefix of the object hash:

```
$ git rev-parse 3b18e512d
3b18e512dba79e4c8300dd08aeb37f8e728b8dad
```

HOW DO WE KNOW A SHA1 HASH IS UNIQUE?

There is an extremely slim chance that two different blobs yield the same SHA1 hash. When this happens, it is called a collision. However, a SHA1 collision is so unlikely that you can safely bank on it never interfering with our use of Git. But could a collision happen at random? Let's see.

With 160 bits, you have 2^{160} or about 10^{48} (1 with 48 zeros after it) possible SHA1 hashes. That number is just incomprehensibly huge. Even if you hired a trillion people to produce a trillion new unique blobs per second for a trillion years, you would still only have about 10^{43} blobs.

If you hashed 2^{80} random blobs, you might find a collision. Don't trust us, go read Bruce Schneier[4].

SHA1 is known as a "cryptographically secure hash." That is until recently, whereby security researchers were able to point out flaws in the integrity of the SHA1 hash function. They published their findings as SHAttered attack[5].

Git, starting from version 2.13.0 moved to implement a hardened SHA1 for it's computation of hash functions. The probability of such an attack vector being repeated is not something that can be guaranteed in the future, for this reason Git introduced a new repository format extension which enables the use of SHA256 instead of SHA1. It is described in detailed in the technical documentation of Git[6].

Next we move up the data structure to understand how path and filenames are stored by Git.

Tree Object and Files

Now that the "hello world" blob is safely ensconced in the object store, let's take a look at how it is associated with a filename. Git wouldn't be very useful if it couldn't find files by name.

As mentioned before, Git tracks the pathnames of files through another kind of object called a *tree*. When you use `git add`, Git creates an object for the

contents of each file you add, but it doesn't create an object for your *tree* right away. Instead, it updates the index. The index is found in *.git/index* and keeps track of file pathnames and corresponding blobs. Each time you run commands such as `git add`, `git rm`, or `git mv`, Git updates the index with the new pathname and blob information.

Whenever you want, you can create a *tree* object from your current index by capturing a snapshot of its current information with the low-level `git write-tree` command. An action which you will rarely execute in your typical daily git rendezvous.

At the moment, the index contains exactly one file, *hello.txt*.

```
$ git ls-files -s
100644 3b18e512dba79e4c8300dd08aeb37f8e728b8dad 0       hello.txt
```

Here you can see the association of the file, *hello.txt*, and the `3b18e5`… blob.

Next, let's capture the index state and save it to a *tree* object:

```
$ git write-tree
68aba62e560c0ebc3396e8ae9335232cd93a3f60

$ find .git/objects
.git/objects
.git/objects/68
.git/objects/68/aba62e560c0ebc3396e8ae9335232cd93a3f60
.git/objects/pack
.git/objects/3b
.git/objects/3b/18e512dba79e4c8300dd08aeb37f8e728b8dad
.git/objects/info
```

Now there are two objects: the "hello world" *blob* object at `3b18e5` and a new one, the *tree* object, at `68aba6`. As you can see, the SHA1 object name corresponds exactly to the subdirectory and filename in *.git/objects*.

But what does a *tree* look like? Because it's an object, just like the blob, you can use the same *low-level plumbing* command to view it.

```
$ git cat-file -p 68aba6
100644 blob 3b18e512dba79e4c8300dd08aeb37f8e728b8dad    hello.txt
```

The contents of the object should be easy to interpret. The first number, `100644`, represents the file attributes of the object in octal, which should be familiar to anyone who has used the Unix `chmod` command. Here, `3b18e5` is the object name of the *hello world* blob, and *hello.txt* is the name associated with that blob.

It is now easy to see that the *tree* object has captured the information that was in the index when you ran `git ls-files -s`.

A Note on Git's Use of SHA1

Before inspecting the contents of the *tree* object in more detail, let's re-emphasize an important feature of SHA1 hashes:

```
$ git write-tree
68aba62e560c0ebc3396e8ae9335232cd93a3f60

$ git write-tree
68aba62e560c0ebc3396e8ae9335232cd93a3f60

$ git write-tree
68aba62e560c0ebc3396e8ae9335232cd93a3f60
```

From the example above, every time you compute another *tree* object for the same index (no adding or removing of files), the SHA1 hash remains exactly the same. Git doesn't need to recreate a new *tree* object. If you're following these steps at the computer, you should be seeing *exactly the same SHA1 hashes* as the ones published in this book.

In this sense, the hash function is a true function in the mathematical sense: For a given input, it always produces the same output. Such a hash function is sometimes called a digest to emphasize that it serves as a sort of summary of the hashed object. This is also true for any hash function, even the lowly parity bit, has this property.

For example, if you create the exact same content as another developer, regardless of where or when or how both of you work, an identical hash is

proof enough that the full content is identical, too. In fact, Git treats them as identical and this notion is extremely important.

But hold on a second—aren't SHA1 hashes unique? What happened to the trillions of people with trillions of blobs per second who never produce a single collision? This is a common source of confusion among new Git users. So read on carefully, because if you can understand this distinction, then everything else in this chapter is easy.

Identical SHA1 hashes in this case *do not count as a collision*. It would be a collision only if two *different* objects produced the same hash. Here, you created two separate instances of the very same content, and the same content always has the same hash.

Git depends on another consequence of the SHA1 hash function: it doesn't matter *how* you got a *tree* called `68aba62e560c0ebc3396e8ae9335232cd93a3f60`. If you have it, you can be extremely confident it is the same *tree* object that, say, another reader of this book has. Consider the following:

- Scenario 1 - Bob might have created the *tree* by combining commits A and B from Jennie and commit C from Sergey on a shared repository
- Scenario 2 - You on the other hand, working in that same shared repository, might have created the same *tree* but via a different path, you might have got commit A from Sue and an update from Lakshmi that combines commits B and C.

The results are the same for the generated *tree* object in both scenarios, this facilitates distributed development with Git.

If you are asked to look for object `68aba62e560c0ebc3396e8ae9335232cd93a3f60` and can find such an object, then, because SHA1 is a cryptographic hash, you can be confident that you are looking at precisely the same data from which the hash was created.

The converse is also true: If you don't find an object with a specific hash in

your object store, then you can be confident that you do not hold a copy of that exact object. In sum, you can determine whether your object store does or does not have a particular object even though you know nothing about its (potentially very large) contents. The hash thus serves as a reliable label or name for the object.

Tree Hierarchies

In our examples from the previous section, we only have information regarding a single file, but in actuality projects contain complex, deeply nested directories that are refactored and moved around over time. In this section we will be creating a new subdirectory that contains an identical copy of the *hello.txt* file in order to see how Git handles this scenario:

```
$ pwd
/tmp/hello
$ mkdir subdir
$ cp hello.txt subdir/
$ git add subdir/hello.txt
$ git write-tree
492413269336d21fac079d4a4672e55d5d2147ac

$ git cat-file -p 4924132693
100644 blob 3b18e512dba79e4c8300dd08aeb37f8e728b8dad    hello.txt
040000 tree 68aba62e560c0ebc3396e8ae9335232cd93a3f60    subdir
```

The new top-level tree contains two items: the original *hello.txt* file as well as the new *subdir* directory, which is of type *tree* instead of *blob*.

Look closer at the object name of *subdir*, do you notice anything unusual? Indeed It's none other than our old friend, the SHA1 `68aba62e560c0ebc3396e8ae9335232cd93a3f60`!

How can this be you ask? Well, the new tree for *subdir* contains only one file, *hello.txt*, and that file contains the same old "hello world" content. So the `*subdir*` tree is exactly the same as the older, top-level tree! And yes, you are correct to point out that it is for this reason alone, it has the same SHA1 object name as before: traits of a *true function* in the mathematical sense.

Let’s look at the *.git/objects* directory and see what this most recent change affected:

```
$ find .git/objects
.git/objects
.git/objects/49
.git/objects/49/2413269336d21fac079d4a4672e55d5d2147ac
.git/objects/68
.git/objects/68/aba62e560c0ebc3396e8ae9335232cd93a3f60
.git/objects/pack
.git/objects/3b
.git/objects/3b/18e512dba79e4c8300dd08aeb37f8e728b8dad
.git/objects/info
```

There are still only three *unique* objects: a blob containing “hello world”; a tree containing *hello.txt*, which contains the text “hello world” plus a new line; and a second tree that contains *another* reference to *hello.txt* along with the first tree.

Figure 2-7 illustrates this concept:

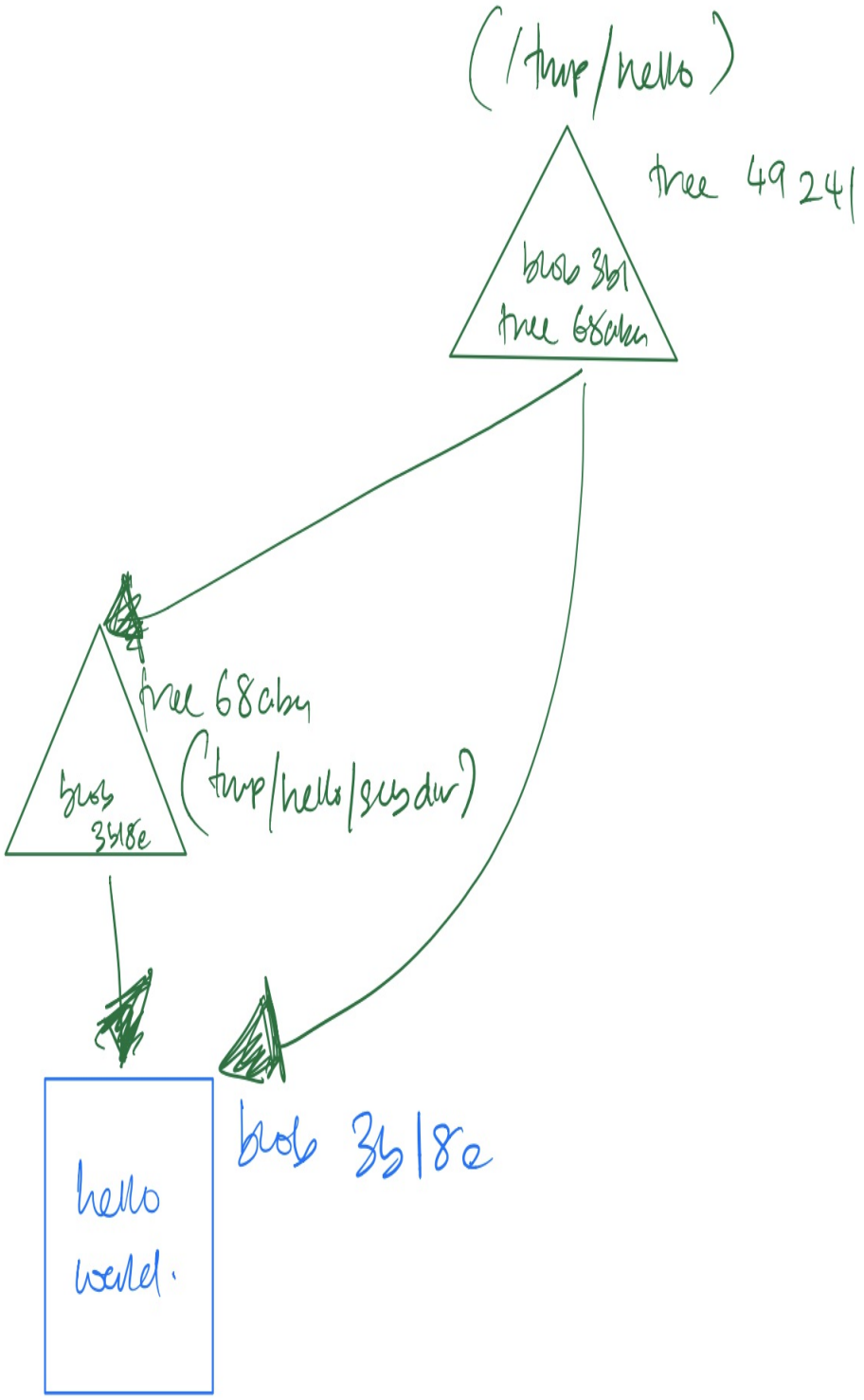

(/tmp/hello)
tree 4924
blob 361
tree 68cbm
tree 68cbm
(tmp/hello/sysdir)
blob 3518e
blob 3518e
hello
world.

Figure 3-4. Tree Hierarchies

Commit Objects

The next object to discuss is the commit. Now that *hello.txt_* has been added with the `git add` command and the tree object has been produced with `git write-tree`, we can create a commit object using *low-level plumbing* commands like this:

```
$ echo -n "Commit a file that says hello\n" | git commit-tree
492413269336d21fac079d4a4672e55d5d2147ac
3ede4622cc241bcb09683af36360e7413b9ddf6c
```

The result will look something like this:

```
$ git cat-file -p 3ede462
tree 492413269336d21fac079d4a4672e55d5d2147ac
author Jon Loeliger <jdl@example.com> 1220233277 -0500
committer Jon Loeliger <jdl@example.com> 1220233277 -0500

Commit a file that says hello
```

Figure 2-8 illustrates this concept:

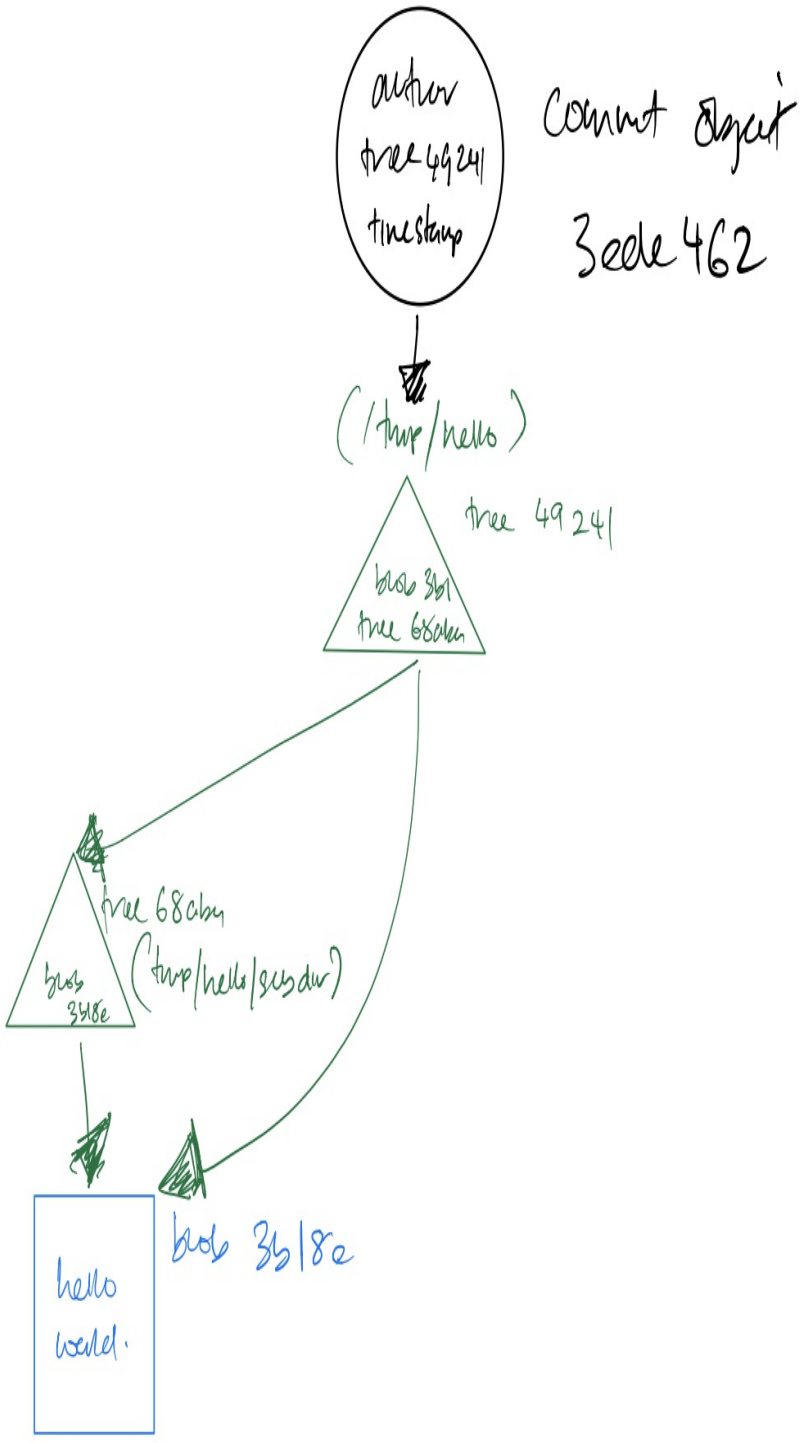

author
tree 4924
timestamp
Commit Object
3ede462
(/tmp/hello)
tree 4924
blob 361
tree 68cbm
tree 68cbm
(tmp/hello/sysdir)
blob 3418e
blob 3418e
hello
world.

Figure 3-5. Commit Object

If you're following along on your computer, you probably find that the commit object you generated does *not* have the same identical value as the one in this example. If you've understood everything so far, the reason for that should be obvious: Our commit object, it's not the same as your commit object.

Your commit object contains your name and the time you made the commit, whereas our commit object contains a different timestamp and author name, so of course it is different.

On the other hand, your commit does have the same *tree*. This is why commit objects are separate from their tree objects: different commits often refer to exactly the same tree. When that happens, Git is smart enough to transfer around only the new commit object, which is tiny, instead of the entire tree and blob objects, which are probably much larger.

In real life, you can (and should!) pass over the *low-level plumbing* commands `git write-tree` and `git commit-tree` used in the examples. You can just use the *porcelain* `git commit` command. You don't need to remember all those plumbing commands to be a perfectly happy Git user.

In essence, a basic commit object is fairly simple, and it's the last ingredient required for a real Version Control System. The commit object just shown, is the simplest possible one, containing:

- The name of a tree object that actually identifies the associated files
- The name of the person who composed the new version (the author) and the time when it was composed
- The name of the person who placed the new version into the repository (the committer) and the time when it was committed
- A description of the reason for this revision (the commit message)

By default, the author and committer are the same; there are a few situations where they're different.

> **TIP**
>
> You can use the command `git show --pretty=fuller` to see additional details about a given commit.

A more closer to home use case is when a project contains multiple commits. In such a situation, when you make a new commit in the project, you can give it one or more parent commits. Given this context, consider the most recent commit (or its associated *tree* object) in the project. Because it contains, as part of its content, the hash of its parent commits and of its *tree* and *that* in turn contains the hash of all of its subtrees and blobs recursively through the whole data structure, it follows by induction that the hash of the original commit uniquely identifies the state of the whole data structure rooted at that commit.

By following back through the chain of parents, you can discover the history of your project, thus the term *commit history*. Commit objects are also stored in a graph structure, although it's completely different from the structures used by tree objects. More details about commits and the commit graph are given in the Commits chapter.

Tag Objects

Finally, the last object Git manages is the tag. Although Git implements only one kind of tag object in it's object store, Git supports two basic tag types, usually called a lightweight and an annotated tag.

Lightweight tags are simply references to a commit object and are usually considered private to a repository. Lightweight tags are not stored as permanent objects in the object store.

An annotated tag is more substantial and creates an object. It contains a message, supplied by you, and can be digitally signed using a GnuPG key according to RFC4880.

Git treats both lightweight and annotated tag names equivalently for the

purposes of associating a commit with meaningful human readable name. However, by default, many Git commands work only on annotated tags, because they are considered “permanent immutable” objects.

> **TIP**
>
> Typical use case for an annotated tag is when you are creating a specific release version for your projects. Whereas lightweight tags are in the light of a bookmark as a temporary label attached to a commit object.

You create an annotated, unsigned tag with a message on a commit using the `git tag` command:

```
$ git tag -a V1.0 3ede462
```

Git will launch your configured default editor after the command is issued and you may provide a tag message to complete the operation.

To view the newly created tag object, you may do so via the `git cat-file -p` command, but what is the SHA1 of the tag object? To find it, use the *Tip* from “Blob Objects and Hashes”:

```
$ git rev-parse V1.0
6b608c1093943939ae78348117dd18b1ba151c6a

$ git cat-file -p 6b608c
object 3ede4622cc241bcb09683af36360e7413b9ddf6c
type commit
tag V1.0
tagger Jon Loeliger <jdl@example.com> Sun Oct 26 17:07:15 2008 -0500

Tag version 1.0
```

In addition to the log message and author information, the tag refers to the commit object `3ede462`.

Git usually tags a commit object, which points to a tree object, which encompasses the total state of the entire hierarchy of files and directories

within your repository.

Recall from Figure 3-3 that the `V1.0` tag points to the commit named `1492`, which in turn points to a tree (`8675309`) that spans multiple files. Thus, the tag simultaneously applies to all files of that tree.

This is unlike CVS, for example, which will apply a tag to each individual file and then rely on the collection of all those tagged files to reconstitute a whole tagged revision. And whereas CVS lets you move the tag on an individual file, Git requires a new commit, encompassing the file state change, onto which the tag will be moved.

Summary

We have discussed the inner workings of Git to an elaborate extent, let's now recap the key takeaways from this chapter. We started at the high level of a repository where we learnt about the various working directory Git replies upon, mainly the Index, Working directory and the local history. We continued to dive into the Git Object Store, where we analyzed each of the Immutable Git Objects: During which we also learnt how to interact with those internal objects directly using low-level git commands that you would rarely use on a daily basis. Grasping this concept should highlight the fact that Git as a concept is merely a simple content addressable database whereby its innermechnics are somewhat direct yet may at times be complex to comprehend. We've also described visually the relationship between the objects in Git's Object Store to help establish a good foundation for the next Chapters in *Part 2: Fundamentals of Git.*

---

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

2 *https://zlib.net*

3 *https://tools.ietf.org/html/rfc1951*

4 *https://www.schneier.com/blog/archives/2005/02/cryptanalysis_o.html*

5 *https://shattered.io*

6 *https://github.com/git/git/blob/master/Documentation/technical/hash-function-transition.txt* or

https://git-scm.com/docs/hash-function-transition