A Working Introduction

Git

Pocket Guide

O'REILLY®

Richard E. Silverman

Software Engineering

O'REILLY®

Git Pocket Guide

This pocket guide is the perfect on-the-job companion to Git, the distributed version control system. It provides a compact, readable introduction to Git for new users, as well as a reference to common commands and procedures for those of you with Git experience.

Written for Git version 1.8.2, this handy task-oriented guide is organized around the basic version control functions you need, such as making commits, fixing mistakes, merging, and searching history.

- Examine the state of your project at earlier points in time
- Learn the basics of creating and making changes to a repository
- Create branches so many people can work on a project simultaneously
- Merge branches and reconcile the changes among them
- Clone an existing repository and share changes with push/pull commands
- Examine and change your repository's commit history
- Access remote repositories, using different network protocols
- Get recipes for accomplishing a variety of common tasks

Richard E. Silverman is a co-author of *SSH, The Secure Shell: The Definitive Guide, Second Edition* and the *Linux Security Cookbook*. He has worked in the fields of networking, software development, formal methods, security, and Unix systems administration.

oreilly.com
twitter.com/oreillymedia
facebook.com/oreilly

US $14.99 CAN $15.99

ISBN: 978-1-449-32586-2

Git Pocket Guide

Richard E. Silverman

O'REILLY®
Beijing · Cambridge · Farnham · Köln · Sebastopol · Tokyo

Git Pocket Guide

by Richard E. Silverman

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://my.safaribooksonline.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Editors: Mike Loukides and Meghan Blanchette
Production Editor: Melanie Yarbrough
Copyeditor: Kiel Van Horn
Proofreader: Linley Dolby
Indexer: Judith McConville
Cover Designer: Randy Comer
Interior Designer: David Futato
Illustrator: Rebecca Demarest

June 2013: First Edition

Revision History for the First Edition:

2013-06-24: First release
2013-07-10: Second release

See *http://oreilly.com/catalog/errata.csp?isbn=9781449325862* for release details.

ISBN: 978-1-449-32586-2

[M]

Table of Contents

Preface

What Is Git?

Git is a tool for tracking changes made to a set of files over time, a task traditionally known as "version control." Although it is most often used by programmers to coordinate changes to software source code, and it is especially good at that, you can use Git to track any kind of content at all. Any body of related files evolving over time, which we'll call a "project," is a candidate for using Git. With Git, you can:

- Examine the state of your project at earlier points in time
- Show the differences among various states of the project
- Split the project development into multiple independent lines, called "branches," which can evolve separately
- Periodically recombine branches in a process called "merging," reconciling the changes made in two or more branches
- Allow many people to work on a project simultaneously, sharing and combining their work as needed

…and much more.

There have been many different version control systems developed in the computing world, including SCCS, RCS, CVS,

Subversion, BitKeeper, Mercurial, Bazaar, Darcs, and others. Some particular strengths of Git are:

- Git is a member of the newer generation of *distributed* version control systems. Older systems such as CVS and Subversion are *centralized*, meaning that there is a single, central copy of the project content and history to which all users must refer. Typically accessed over a network, if the central copy is unavailable for some reason, all users are stuck; they cannot use version control until the central copy is working again. Distributed systems such as Git, on the other hand, have no inherent central copy. Each user has a complete, independent copy of the entire project history, called a "repository," and full access to all version control facilities. Network access is only needed occasionally, to share sets of changes among people working on the same project.
- In some systems, notably CVS and Subversion, branches are slow and difficult to use in practice, which discourages their use. Branches in Git, on the other hand, are very fast and easy to use. Effective branching and merging allows more people to work on a project in parallel, relying on Git to combine their separate contributions.
- Applying changes to a repository is a two-step process: you add the changes to a staging area called the "index," then commit those changes to the repository. The extra step allows you to easily apply just some of the changes in your current working files (including a subset of changes to a single file), rather than being forced to apply them all at once, or undoing some of those changes yourself before committing and then redoing them by hand. This encourages splitting changes up into better organized, more coherent and reusable sets.
- Git's distributed nature and flexibility allow for many different styles of use, or "workflows." Individuals can share work directly between their personal repositories. Groups can coordinate their work through a single central

repository. Hybrid schemes permit several people to organize the contributions of others to different areas of a project, and then collaborate among themselves to maintain the overall project state.

- Git is the technology behind the enormously popular "social coding" website GitHub, which includes many well-known open source projects. In learning Git, you will open up a whole world of collaboration on small and large scales.

Goals of This Book

There are already several good books available on Git, including Scott Chacon's *Pro Git*, and the full-size *Version Control with Git* by Jon Loeliger (O'Reilly). In addition, the Git software documentation ("man pages" on Unix) is generally well written and complete. So, why a *Git Pocket Guide*? The primary goal of this book is to provide a compact, readable introduction to Git for the new user, as well as a reference to common commands and procedures that will continue to be useful once you've already gotten some Git under your belt. The man pages are extensive and very detailed; sometimes, it's difficult to peruse them for just the information you need for simple operations, and you may need to refer to several different sections to pull together the pieces you need. The two books mentioned are similarly weighty tomes with a wealth of detail. This Pocket Guide is task oriented, organized around the basic functions you need from version control: making commits, fixing mistakes, merging, searching history, and so on. It also contains a streamlined technical introduction whose aim is to make sense of Git generally and facilitate understanding of the operations discussed, rather than completeness or depth for its own sake. The intent is to help you become productive with Git quickly and easily.

Since this book does not aim to be a complete reference to all of Git's capabilities, there are Git commands and functions that we do not discuss. We often mention these omissions explicitly, but some are tacit. Several more advanced features are just mentioned

and described briefly so that you're aware of their existence, with a pointer to the relevant documentation. Also, the sections that cover specific commands usually do not list every possible option or mode of operation, but rather the most common or useful ones that fit into the discussion at hand. The goal is simplicity and economy of explanation, rather than exhaustive detail. We do provide frequent references to various portions of the Git documentation, where you can find more complete information on the current topic. This book should be taken as an introduction, an aid to understanding, and a complement to the full documentation, rather than as a replacement for it.

At the time of this writing in early 2013, Git is undergoing rapid development; new versions appear regularly with new features and changes to existing ones, so expect that by the time you read this, some alterations will already have occurred; that's just the nature of technical writing. This book describes Git as of version 1.8.2.

Conventions Used in This Book

Here are a few general remarks and conventions to keep in mind while reading this book.

Unix

Git was created in the Unix environment, originally in fact both for and by people working on the core of the Linux operating system. Though it has been ported to other platforms, it is still most popular on Unix variants, and its commands, design, and terminology all strongly reflect its origin. Especially in a Pocket Guide format, it would be distracting to have constant asides on minor differences with other platforms, so for simplicity and uniformity, this book assumes Unix generally in its descriptions and choice of examples.

Shell

All command-line examples are given using the *bash* shell syntax. Git uses characters that are special to *bash* and other shells as well, such as `*`, `~`, and `?`. Remember that you will need to quote these in order to prevent the shell from expanding them before Git sees them. For example, to see a log of changes pertaining to all C source files, you need something like this:

```
$ git log -- '*.c'
```

and not this:

```
$ git log -- *.c
```

The latter is unpredictable, as the shell will try to expand `*.c` in the current context. It might do any number of things; few of them are likely to be what you want.

The examples given in the book use such quoting as necessary.

Command Syntax

We employ common Unix conventions for indicating the syntax of commands, including:

- `--{foo,bar}` indicates the options `--foo` and `--bar`.
- Square brackets indicate an optional element that may appear or not; e.g., `--where[=`*`location`*`]` means that you may either use `--where` by itself (with some default location) or give a specific location, perhaps `--where=Boston`.

Typography

The following typographical conventions are used in this book:

Italic
: Indicates new terms; also, Git branches are normally given in italic, as opposed to other names such as tags and commit IDs, which are given in constant width. Titles to Unix manpages are also given in italics.

`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

These lines signify a tip, warning, caution, or general note.

Using Code Examples

This book is here to help you get your job done. In general, if this book includes code examples, 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: "*Git Pocket Guide* by Richard E. Silverman (O'Reilly). Copyright 2013 Richard Silverman, 978-1-449-32586-2."

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

Safari® Books Online

Safari Books Online is an on-demand digital library that delivers expert content 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 of product mixes and pricing programs for organizations, government agencies, and individuals. 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 dozens more. For more information about Safari Books Online, please visit us online.

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 *http://oreil.ly/git_pocket_guide*.

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

I gratefully acknowledge the support and patience of everyone at O'Reilly involved in creating this book, especially my editors Meghan Blanchette and Mike Loukides, during a book-writing process with a few unexpected challenges along the way. I would also like to thank my technical reviewers: Robert G. Byrnes, Max Caceres, Robert P. J. Day, Bart Massey, and Lukas Toth. Their attention to detail and thoughtful criticism have made this a much better book than it would otherwise have been. All errors that survived their combined assault are mine and mine alone.

I dedicate this book to the memory of my grandmother, Eleanor Gorsuch Jefferies (19 May 1920–18 March 2012).

Richard E. Silverman
New York City, 15 April 2013

CHAPTER 1

Understanding Git

In this initial chapter, we discuss how Git operates, defining important terms and concepts you should understand in order to use Git effectively.

Some tools and technologies lend themselves to a "black-box" approach, in which new users don't pay too much attention to how a tool works under the hood. You concentrate first on learning to manipulate the tool; the "why" and "how" can come later. Git's particular design, however, is better served by the opposite approach, in that a number of fundamental internal design decisions are reflected directly in how you use it. By understanding up front and in reasonable detail several key points about its operation, you will be able to come up to speed with Git more quickly and confidently, and be better prepared to continue learning on your own.

Thus, I encourage you to take the time to read this chapter first, rather than just jump over it to the more tutorial, hands-on chapters that follow (most of which assume a basic grasp of the material presented here, in any case). You will probably find that your understanding and command of Git will grow more easily if you do.

Overview

We start by introducing some basic terms and ideas, the general notion of branching, and the usual mechanism by which you share your work with others in Git.

Terminology

A Git project is represented by a "repository," which contains the complete history of the project from its inception. A repository in turn consists of a set of individual snapshots of project content —collections of files and directories—called "commits." A single commit comprises the following:

A project content snapshot, called a "tree"
: A structure of nested files and directories representing a complete state of the project

The "author" identification
: Name, email address, and date/time (or "timestamp") indicating who made the changes that resulted in this project state and when

The "committer" identification
: The same information about the person who added this commit to the repository (which may be different from the author)

A "commit message"
: Text used to comment on the changes made by this commit

A list of zero or more "parent commits"
: References to other commits in the same repository, indicating immediately preceding states of the project content

The set of all commits in a repository, connected by lines indicating their parent commits, forms a picture called the repository "commit graph," shown in Figure 1-1.

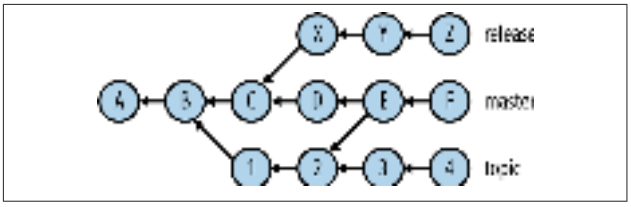

Figure 1-1. The repository "commit graph"

The letters and numbers here represent commits, and arrows point from a commit to its parents. Commit A has no parents and is called a "root commit"; it was the initial commit in this repository's history. Most commits have a single parent, indicating that they evolved in a straightforward way from a single previous state of the project, usually incorporating a set of related changes made by one person. Some commits, here just the one labeled E, have multiple parents and are called "merge commits." This indicates that the commit reconciles the changes made on distinct branches of the commit graph, often combining contributions made separately by different people.

Since it is normally clear from context in which direction the history proceeds—usually, as here, parent commits appear to the left of their children—we will omit the arrow heads in such diagrams from now on.

Branches

The labels on the right side of this picture—*master*, *topic*, and *release*—denote "branches." The branch name refers to the latest commit on that branch; here, commits F, 4, and Z, respectively, are called the "tip" of the branch. The branch itself is defined as the collection of all commits in the graph that are reachable from the tip by following the parent arrows backward along the history. Here, the branches are:

- *release* = {A, B, C, X, Y, Z}
- *master* = {A, B, C, D, E, F, 1, 2}

- *topic* = {A, B, 1, 2, 3, 4}

Note that branches can overlap; here, commits 1 and 2 are on both the *master* and *topic* branches, and commits A and B are on all three branches. Usually, you are "on" a branch, looking at the content corresponding to the tip commit on that branch. When you change some files and add a new commit containing the changes (called "committing to the repository"), the branch name advances to the new commit, which in turn points to the old commit as its sole parent; this is the way branches move forward. From time to time, you will tell Git to "merge" several branches (most often two, but there can be more), tying them together as at commit E in Figure 1-1. The same branches can be merged repeatedly over time, showing that they continued to progress separately while you periodically combined their contents.

The first branch in a new repository is named *master* by default, and it's customary to use that name if there is only one branch in the repository, or for the branch that contains the main line of development (if that makes sense for your project). You are not required to do so, however, and there is nothing special about the name "master" apart from convention, and its use as a default by some commands.

Sharing Work

There are two contexts in which version control is useful: private and public. When working on your own, it's useful to commit "early and often," so that you can explore different ideas and make changes freely without worrying about recovering earlier work. Such commits are likely to be somewhat disorganized and have cryptic commit messages, which is fine because they need to be intelligible only to you, and for a short period of time. Once a portion of your work is finished and you're ready to share it with others, though, you may want to reorganize those commits, to make them well-factored with regard to reusability of the changes being made (especially with software), and to give them meaningful, well-written commit messages.

In centralized version control systems, the acts of committing a change and publishing it for others to see are one and the same: the unit of publication is the commit, and committing requires publishing (applying the change to the central repository where others can immediately see it). This makes it difficult to use version control in both private and public contexts. By separating committing and publishing, and giving you tools with which to edit and reorganize existing commits, Git encourages better use of version control overall.

With Git, sharing work between repositories happens via operations called "push" and "pull": you pull changes from a remote repository and push changes to it. To work on a project, you "clone" it from an existing repository, possibly over a network via protocols such as HTTP and SSH. Your clone is a full copy of the original, including all project history, completely functional on its own. In particular, you do not need to contact the first repository again in order to examine the history of your clone or commit to it—however, your new repository does retain a reference to the original one, called a "remote." This reference includes the state of the branches in the remote as of the last time you pulled from it; these are called "remote tracking" branches. If the original repository contains two branches named *master* and *topic*, their remote-tracking branches in your clone appear qualified with the name of the remote (by default called "origin"): *origin/master* and *origin/topic*.

Most often, the *master* branch will be automatically checked out for you when you first clone the repository; Git initially checks out whatever the current branch is in the remote repository. If you later ask to check out the *topic* branch, Git sees that there isn't yet a local branch with that name—but since there is a remote-tracking branch named *origin/topic*, it automatically creates a branch named *topic* and sets *origin/topic* as its "upstream" branch. This relationship causes the push/pull mechanism to keep the changes made to these branches in sync as they evolve in both your repository and in the remote.

When you pull, Git updates the remote-tracking branches with the current state of the origin repository; conversely, when you push, it updates the remote with any changes you've made to corresponding local branches. If these changes conflict, Git prompts you to merge the changes before accepting or sending them, so that neither side loses any history in the process.

If you're familiar with CVS or Subversion, a useful conceptual shift is to consider that a "commit" in those systems is analogous to a Git "push." You still commit in Git, of course, but that affects only your repository and is not visible to anyone else until you push those commits—and you are free to edit, reorganize, or delete your commits until you do so.

The Object Store

Now, we discuss the ideas just introduced in more detail, starting with the heart of a Git repository: its *object store*. This is a database that holds just four kinds of items: *blobs*, *trees*, *commits*, and *tags*.

Blob

A *blob* is an opaque chunk of data, a string of bytes with no further internal structure as far as Git is concerned. The content of a file under version control is represented as a blob. This does not mean the implementation of blobs is naive; Git uses sophisticated compression and transmission techniques to handle blobs efficiently.

Every version of a file in Git is represented as a whole, with its own blob containing the file's complete contents. This stands in contrast to some other systems, in which file versions are represented as a series of differences from one revision to the next, starting with a base version. Various trade-offs stem from this design point. One is that Git may use more storage space; on the other hand, it does not have to reconstruct files to retrieve them by applying layers of differences, so it can be faster. This design increases reliability by increasing redundancy: corruption of one blob affects only that file version, whereas corruption of a difference affects all versions coming after that one.