Outline of *Professional Git*

Brent Laster

Table of Contents

Introduction

Professional Git is intended to help you understand and use Git to get your job done. In the process, it will also make Git part of your professional comfort zone.

This section will explain how this book is unique from other books about Git, the intended audience, the book's overall structure and content, and some of the value it offers you.

How This Book is Unique

Git in terms of Known Concepts

Most books are aimed at providing the technical usage as their major and singular goal. This book provides that, but it will aso provide you with an understanding of Git in terms of *concepts* that you probably already know.

Connected Labs for Actual Examples

Most books do not provide practical ways to integrate the concepts they describe. Learning is most effective when you have *actual examples* to work through so you can internalize the concepts and gain proficiency. This book provides **Connected Labs** that you can work through to absorb what you've read.

Illustrations for Key Ideas and Workflows

This book also provides clear *illustrations* to help you visualize key ideas and workflows. These sections provide additional explanations of how to use some lesser-known features of Git as well as how to go beyond the standard Git features to gain extra valua.

Git Model and Workflow

To be most effective, you need to comprehend the Git *model* and *workflow*. You should also know what to watch out for as you make the transition and why it's important to consider not only the commands and workflow, but also the structure and scope of its underlying repositories.

Target Audience

There is only one assumption in this book: that the reader has experience with at least one source management system. It doesn't matter which one: CVS, Subversion, Mercury—any will do. The reader is assumed to have a basic awareness of what a source management system does as well as fundamental concepts such as checking in and checking out code and branching. Even if you have significant experience with Git or another system, the reader will find something of benefit here.

Git requires a mind shift. In fact, requires a series of mind shifts. However, each shift is easy to understand once you can relate it to something you already know. Understanding each of these shifts will, in turn, allow you to be more productive and to harness the features of this powerful tool—that's what this book is about.

Structure and Content

This book is organized as a series of chapters that present Git from the ground up, teaching what is needed to know and build on to become proficient before adding new concepts.

Part One: Foundational Git

The first three chapters cover the foundational concepts of Git: how it is different from other systems, the ecosystem that's been built around it, its advantages and challenges, and the model that allows you to understand its workflow and manage content effectively with it. This section will provide you with a basic understanding of the ideas, goals and essential terminology of Git.

Part Two: Usage and Features of Git

The remaining chapters of the book cover the usage and features of Git, from performing basic operations to create repositories and commit changes into them, to creating branches, doing merges, and working with content in public repositories.

Reader Value

You will find:

- examples and guidance on the commands and workflows needed to be productive with Git;
- ways to relate concepts to what you already know and understand;
- many illustrations to help you understand concepts visually;
- a feature that allows you to get hands-on experience with Git, via **Connected Labs**, insterspersed throught the chapters. These labs are designed to reinforce the concepts presented in the text of the prceding chapters and to get you actively involved in the learning process.

To get the most out of the book, you should take the time to complete each lab. As well, talk a look at the Advanced Topics sections.

For the later labs, custom Git repositories with example content are provided for the user at `http://hithub.com/professional-git`. In addition, downlable copies of the code for the hooks from the last chapter are availabe in `http://github.com/professional-git/hooks`.

UNDERSTANDING GIT CONCEPTS

1 What Is Git

Chapter Goals

- A brief introduction to Git and its history
- The different ways to find and access Git
- Types of applications that incorporate Git
- The advantages of using Git
- The challenges of using Git

Here you will be introduced to Git and will learn about it from a product perspective—what it is, why it's used, the different kinds of interfaces you can use with it, and the good parts and challenging parts of working with it.

One Paragraph Summary

> Git is a popular and widely used source management system that greatly simplifies the development cycle. It enables uers to create, use, and switch between branches for content development as easily as people create and switch between files in their daily workflow. It is implemented using a fast, efficient architecture that allows for ease of experimentation and refinement of local changes in an isolated environment before sharing with with others. In shortj, it allows everyday users to focus on getting the content right instead of worrying aboust source management, while providing more advanced users with the ability to record, edit, and share changes at any level of detail.

1.1 History of Git

Git was born from within the environment of the Linux kernel. In around 2005, Linus Torvalds, the creator of Linux, set out to create a new system that maintained the distributed deal, but also incorporated several additional concepts he had been working with. He wanted it to provide the fast performance that a project on the scope of the Linux kernel would need. Development began in early April 2005, and an initial release was ready by July.

1.2 Industry-Standard Tooling

Git has grown to become an industry-standard tool.

- It is used across all levels of industry;
- Huge projects, such as the Linux kernel, are managed in it, and also mandate its use;
- It is a key component of many continuous integration/continuous delivery pipelines;
- Demand for knowledge about it is every increasing;
- Commercial and open-source projects and applications recognize that if they require soure management services, they have to integrate witih Git;

An entire ecosystem has sprung up around Git. The basic tool that is Git has given rise to a seemingly endless number of applications to further help users who want to work with it.

- GitHub
- Gitolite
- Easy Git
- Git Extensions
- EGit

1.3 The Git Ecosystem

You can break down the Git-based offerings into a few categories:

- core Git
- Git-hosting sites
- self-hosting packages
- ease-of-use packages
- plug-ins
- tools that incorporate Git
- Git libraries

1.3.1 Core Git

- Core Git `https://git-scm.com/downloads`
  - basic Git executables
  - configuration files
  - repository management tooling
- Supporting tools
  - simple GUI (guit gui)
  - history visualization tool (gitk)
  - alternate interface (Bash shell on Windows)
  - ported version of Mac OS X
  - Linux package management installs

1.3.2 Git-Hosting Sites

Git-hosting sites are websites that provide hosting services for Git repositories, both for personal and shared projects. Customers may be individuals, open-source collaborators, or businesses. Many open-source projects have their Git repositories hosted on these sites.

In addition to the basic hosting services, these sites offer added value in the form of custom browsing features, easy web interfaces to Git commands, integrated bug-tracking, and the ability to easily set up and share access among teams or groups of individuals.

These sites typically provide a workflow intended to allow users to contribute back to projects on the site. At a high level, this usually involves getting a copy of another user's repository, making changes in the copy, and then requesrting that the original user review and incorporate the changes; this is sometimes know as the *fork and pull* model.

For hosting, there is a pricing model that depends on the level of access, number of users, number of repositories, or features needed.

Examples of these sites include

- GitHub
- Bitbucket

1.3.3 Self-Hosting Packages

Based on the success of the model and usage of hosting sites, several packages have been developed to provide a similar functionality and experience for users and groups without having to rely on an external service. For some, this is their primary target market (GitLab), while others are stand-alone (also known as *on-premise*) versions of the popular web-hosting sites (such as GitHub Enterprise).

These packages are more palatable to businesses that do not want to host their code externally (on someone else's servers), but still want the collaborative features and control that are provided with the model.

1.3.4 Ease-of-Use Packages

The ease-of-use category encompasses applications that sit on top of the basic Git tooling with the intention of simplifying user interaction with Git. Typically this means they provide GUI interfaces for working with repositories and may support GUI-based conventions such as drag-and-drop to move content between levels. They often provide graphical tools for labor-intensive operations, such as merging.

Examples include:

- SourceTree
- SmartGit
- TortoiseGit
- Git Extensions

Typically these packages are free for non-commercial use. You can see a more comprehensive list at `https://git-scm.com/downloads/guis`.

1.3.5 Plug-Ins

Plug-ins are software components that add interfaces for working with Git to existing applications. Common plug-ins that users may deal with are those for popular IDEs such as:

- Eclipse
- IntelliJ
- Visual Studio

or those that integrate with workflow tools such as:

- Jenkins
- TeamCity

It is now becoming more common for applications to include a Git plug-in by default, or to just build it in directly.

1.3.6 Tools That Incorporate Git

Tooling has emerged that directly incorporates and uses Git as part of its modfel. One example is Gerrit, a tool designed primarily to do code reviews on changes targeted for Git remote repositories. At its core, Gerrit manages Git repositories and inserts itself into the Git workflow. It wraps Git repositories in a project structure with access control, a code review workflow and tooling, and the ability to configure otgher validations and checks on the code.

1.3.7 Git Libraries

For interfacing with some programming languages, developers have implemented libraries that wrap those languages or re-implement the Git functionality. One of the best-known examples of this is JGit. JGit is a Java library that re-implements Git and is used by a number of applications such as Gerrit. These implementations make interfacing with Git programmatically much more direct.

1.4 Git's Advantages and Challenges

Everyone has opinions. These lists are subjective, but themes seem to consistently emerge.

1.4.1 The Advantages

There are some things Git does better (faster, easier) than other source management systems, and some things it takes a totally different approach on. Learning about and leveraging the aspects outlined here will allow you to get the most out of this tool.

1.4.1.1 Disconnected Development

The Git model provides a local environment where you can work with a local copy of a server-side environment (known as the *remote* in Git terminology). This copy resides within your workspace. When you are satisfied with your changes in this local repository, you then sync the local repository's contents up with the remote side. There's no need to connect to the remote repository until you are ready to sync content. This means you can work *disconnected* from the remote and even disconnected from a network. This is what *disconnected development* means.

1.4.1.2 Fast Performance

Git stores a lot of information. However, it is efficient both in the way it stores content and in the way it retrieves it. Internally, Git packs together similar objects. Externally, it uses a good compression model to send significant amounts of data efficiently through a network.

For changes in the local environment, Git is as fast as its commands can be executed on your disk. Because it only has to interact with a local repository, the performance is equivalent to operating system commands.

It is designed to manage multiple smaller repositories—rather than larger aggregate ones that may be present in traditional source control systems. Thsi granularity contributes to the smaller amount of content that has to be moved around in Git, and thus to a faster operation.

Branching is extremely fast in Git, essentially as fast as you can create a file on your OS. This means there is no more waiting for extended periods while the source management system branches your content. Deleting branches is just as quick. Merging is generally quick as well, assuming there are no conflict.

1.4.1.3 Ease of Use

There's a paradigm shift that is required when learning to use Git. And a prerequisite to thinking that Git is easy to use is understanding it. However, once you grasp the concepts and start to use this tool regularly, it becomes both easy to use and powerful. There are simply default forms of commands and options. As your proficiency grows, there are extended forms that can allow you to do nearly anything you need to do with your content. Almost everything about Git settings is configurable so that you can customize your working environment.

The primary mistake that most new Git users make is trying to use it in the same way that they have always used their traditional management system. A better approach is to consider what sort of source management outcome is needed (files in the repository, viewing hsitory), and then take the time to learn how that workflow is done with Git. The Connected Labs will aid this process significantly by providing hands-on experience with Git.

1.4.1.4 SHA1s

This is an acronym for Secure Hashing Algorithm 1. It is a checksum. Git computes SHA1's internally as keys for everything it stores in its repositories. This means that every change in Git has a unique identifier and that it's not possible to change content that Git manages without Git knowing about it—because the checksum would change.

1.4.1.5 Abiltity to Rewrite History

One aspect of Git that is different from most other source management systems is the ability to rewrite or *redo* previous versions of content stored in the repository—that is, its *history*. Git provides functionality that allows you to travers previous versions, edit and update them, and place the updated versions back in the same sequence of changes stored in the repository.

When content that you are working on in the local repository has not yet been sync'ed to the remote side, this is a safe operation. It can be very beneficial. Git provides an *amend* option that allows you to update or replace the last change made in the local repository.

Additional functionality makes it possible take selected changes from one branch and incorporate them directly into the line of changes in another branch. Beyond that are levels of functionality for dfoing editing throughout the history of one or more branches. An example case would be removing a hard-coded password that was accidentally introduced into the history months ago from all affected versions.

1.4.1.6 Staging Area

Git includes an intermediate level between the directly where content is created and edited, and the repository where content is committed. It provides a separate area for use in some of Git's advanced operations, such as the amend option. It also simplifies some status tracking.

1.4.1.7 Strong Support for Branching

Using branches is a core concept of Git. Git provides capabilities for changing branch points and reproducing changes from one branch onto another branch—a feature known as *rebasing*. This ease in working with and manipulating branches forms the basis for a development model with Git. In this model, branches are managed as easily as files are in some other systems.

1.4.1.8 One Working Area—Many Branches

When products are managed via a continuous delivery process, in a user's local environment, there are typically multiple changes underway, for new features, bug fixes, and so on. In Git, this is a single-step process. Git allows you to work in one workspace for a repository, regardless of how many branches you may have or need to use. It manages updating the content in the workspace to ensure it is consistent with whichever branch is active. You never need to leave the workspace. While working in one branch, you still have the expected access to view, merge, or create other branches.

1.4.2 The Challenges

1.4.2.1 Very Different Model from Some Traditional Systems

1.4.2.2 Different Commands for Moving Content

1.4.2.3 Staging Area

1.4.2.4 Mind Shift and Learning Curve

1.4.2.5 Limited Support for Binary Files

Git does not have strong support for binary files. There are two aspects of dealing with binary files that are challenging here:

1. internal format
2. size

Because of the internal format of these types of files where the bits rather than the characters are what is important, standard source management operations can be difficult to apply or may not make sense at all. An example of the former would be *diffing*. An example of the latter would be managing line endings. If the SCM does not recognize or understand that a particular file is binary and tries to execute these types of operations against it, the results can be confusing and problematic.

The size of binary files can routinely be much larger than text ones. Veryi large binary files can pose a challenge for a system like Git since they usually cannot be compressed very much, and so can impose more time and space to manage, leading to extended operation times when the system has to pass around these files such as when copying to a local system.

Git has built-in mechanisms for identifying files as binary. However, it is also possible (and a best practice) to use one of its supporting files—the Git Attributes file—to explicitly identify which types of files are binary.

Artifact repositories, such as Artificatory and Nexxus, are targeted specifically at storing and managing revisions of binary files. The Git community itself has created various applications targeted at helping with this. Currently, the best-known one is probably Git LFS (Git Large File Storage)—a solution from the Git hosting site GitHub. This application stores large files in a separate repository and stores text pointers in the traditional Git repository to those large files.

1.4.2.6 No Version Numbers

From a user perspective, SHA1s are not as convenient to remember, find, or communicate about as traditional version numbers.

1.4.2.7 Merging Scope

Any two changes by different users within the scope of a *commit* can be a conflict, even if they are in entirely different files or directories. As a result, the more people that are making changes within the scope of a repository, the more likely they are to encounter merge conflicts when trying to get their updates in. This is a factor to consider when planning how to structure your Git repositories.

1.4.2.8 Ability to Rewrite History

On the challenging side of the scale is the potential impact that uncoordinated use can have on other users. As a highly recommended guideline changes that alter history should onlhy be made in a user's local environment *before* the affected revisions are pushed across to the remote side.

1.4.2.9 Timestamps

Due to the way that remote repositories are sync'ed from local repositories, the timestamp that shows up in the remote repository is the time the update was made on the *local* environment, not the timestamp of when things were sync'ed to the remote. You can't rely on timestamps for some of the cases where they are traditionally employed with existing source control systems.

1.4.2.10 Access and Permissions

Out of the box, Git does not provide a layer to set up users or to grant or deny access. For the local environment, this doesn't matter. For shared server-side repositories, there are a few options:

- Using operating system mechanisms such as groups and umasks that limit the set of users and their direct repository permissions;
- Limited access via client-server protocols (SSH, HTTPS);
- Adding an exteranl applications layer that implements a more fine-grained permissions model and interface;

2 Key Concepts

3 The Git Promotion Model

USING GIT

4 Configuration and Setup

5 Getting Productive

6 Tracking Changes

7 Working With Changes Over Time and Using Tags

8 Working With Local Branches

9 Merging Content

10 Supporting Files in Git

11 Doing More With Git

12 Understanding Remotes—Branches andc Operations

13 Understanding Remotes—Workflows for Changes

14 Working With Trees and Modules in Git

15 Extending Git Functionality With Git Hooks

Index