PART I
Understanding Git Concepts

- **CHAPTER 1:** What Is Git?
- **CHAPTER 2:** Key Concepts
- **CHAPTER 3:** The Git Promotion Model

1

What Is Git?

WHAT'S IN THIS CHAPTER?

- 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

In this chapter, you'll 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. This will provide an important foundation for understanding the technical details that follow in the subsequent chapters.

If I were to summarize what Git is in one paragraph, it would go something like this:

Git is a popular and widely used source management system that greatly simplifies the development cycle. It enables users 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 them with others. In short, it allows everyday users to focus on getting the content right instead of worrying about source management, while providing more advanced users with the ability to record, edit, and share changes at any level of detail.

In short, Git is different—really. When you're experienced with using Git and understand it, this will make you feel empowered and productive. When you're new to Git, and trying to understand it, you will encounter a model that will lead you to think differently about managing content in source control.

To illustrate, there's an old saying that "when all you have is a hammer, everything looks like a nail." When all you have is a traditional centralized source management system, everything looks like a file-by-file change that is expensive to branch.

Not so with Git. Git is one of those nice tools that actually allows users to focus on developing content and simplifying workflows. It's not just another tool in the toolbox, it is the toolbox. It contains all of the tools you need to manage tracking anything from a few files for a single user to projects spanning hundreds of users and a huge scope, such as the Linux kernel. Today, many large companies use Git. It's free, it's powerful, it scales, and its model works when used as designed.

Git also has a certain "feel" that's appealing to many people. Git is structured more like a series of individual utilities that you can run against your content, similar to how users work with operating systems. However, it doesn't try to be the system; it gives users ultimate control over their content, even to the point of being able to update history if needed. Git manages basic units that equate to directory structures rather than individual files, so content that extends across file and directory boundaries can be managed together. Git simplifies branching, to a point where creating, merging, or deleting branches becomes nearly as quick and easy as creating, merging, or deleting files. It also provides a local environment with full source management control that can be updated independently of the shared, public environment.

Given that it is different from other source code management (SCM) systems, it's useful to understand how Git originated. The following section includes some of its history.

HISTORY OF GIT

Git has its roots in the development environment for the Linux kernel. In the early 2000s, the team working on the kernel began using a proprietary distributed source control system called BitKeeper (sometimes abbreviated as BK). The team was initially allowed to use this system for free. Over time, differences of opinion developed around the use of BK to the point that the owner of that system revoked the free use of the product. At that time (in 2005), Linus Torvalds, the creator of Linux, set out to create a new system that maintained the distributed ideal, but also incorporated several additional concepts he had been working with. Perhaps most importantly, he wanted it to provide the fast performance that a project on the scope of the Linux kernel would need. Thus the motivation and ideas for what became Git came into being.

Development began in early April of 2005, and an initial release was ready by July. Originally, there was an idea of purposing Git as a toolkit that could have other systems implemented on top of it. However, over time, it has been made into a full-fledged SCM in its own right.

If you're wondering about the name, there are multiple definitions for the word *Git,* but all of them imply a negative connotation about a person. Git was given its name by its creator. Linus jokingly stated that he named all his projects after himself.

For those interested in learning more about this phase of Git development, detailed historical information is available on the Internet.

INDUSTRY-STANDARD TOOLING

From these early beginnings, Git has grown to become an industry-standard tool. Of course, *industry standard* is a relative term. Nevertheless, based on nearly any criteria, Git fits. It is used across all levels of industry. Huge projects, such as the Linux kernel, are managed in it, and also mandate

its use (see the following list). It is a key component of many continuous integration/continuous delivery pipelines. Demand for knowledge about it is ever increasing. Commercial and open-source projects and applications recognize that if they require source management services, they have to integrate with Git. Projects and companies using Git include

- Google
- Facebook
- Microsoft
- Twitter
- LinkedIn
- Netflix
- O'Reilly
- PostgreSQL
- Android
- Linux
- Eclipse

As with any sufficiently successful open-source technology, an entire ecosystem has sprung up around Git. This point is worth discussing for a moment. 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—most named with some wordplay based on *git*. If you start discussing Git with someone, you may hear such names as GitHub, Gitolite, Easy Git, Git Extensions, EGit, and so on. To the uninitiated, it can be challenging to understand how each one of these names relates to the original Git tooling. To help clarify some of the confusion, I'll give you an overview of how the different offerings are categorized.

THE GIT ECOSYSTEM

Broadly, 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, and Git libraries.

Core Git

In the core Git category, you have the basic Git executables, configuration files, and repository management tooling that you can install and use through the command line interface. (These can be installed from https://git-scm.com/downloads.) In addition to the basic pieces, the distributions usually include some supporting tools such as a simple GUI (git gui), a history visualization tool (gitk), and in some cases, an alternate interface such as a Bash shell that runs on Windows. The distribution for Windows is now called Git for Windows. Similarly there is a ported version of Git for OS/X. This version can be installed directly from the git-scm.com site, or via the Homebrew package manager or built via the MacPorts application.

When installing on Linux systems, the recommended method is to use the preferred package manager for your distribution. Example commands are shown in the following list.

- Debian/Ubuntu
  - $ apt-get install git
- Fedora (up to 21)
  - $ yum install git
- Fedora (22 and beyond)
  - $ dnf install git
- FreeBSD
  - $ cd/usr/ports/devel/git
  - $ make install
- Gentoo
  - $ emerge --ask --verbose dev-vcs/git
- OpenBSD
  - $ pkg_add git
- Solaris 11 Express
  - $ pkg install developer/versioning/git

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 requesting that the original user review and incorporate the changes; this is sometimes known as the *fork and pull* model. (This model is explained in more detail in Chapter 13.)

For hosting, there is a pricing model that depends on the level of access, number of users, number of repositories, or features needed. For example, if a repository is intended to be public—with open access to anyone—it may be hosted for free. If access to a repository needs to be limited or it needs a higher level of service, then there may also be a charge. In addition, the hosting site may offer services such as consulting or training to generate revenue.

Examples of these types of sites include GitHub and Bitbucket. Figure 1-1 shows an example of a GitHub repository page.

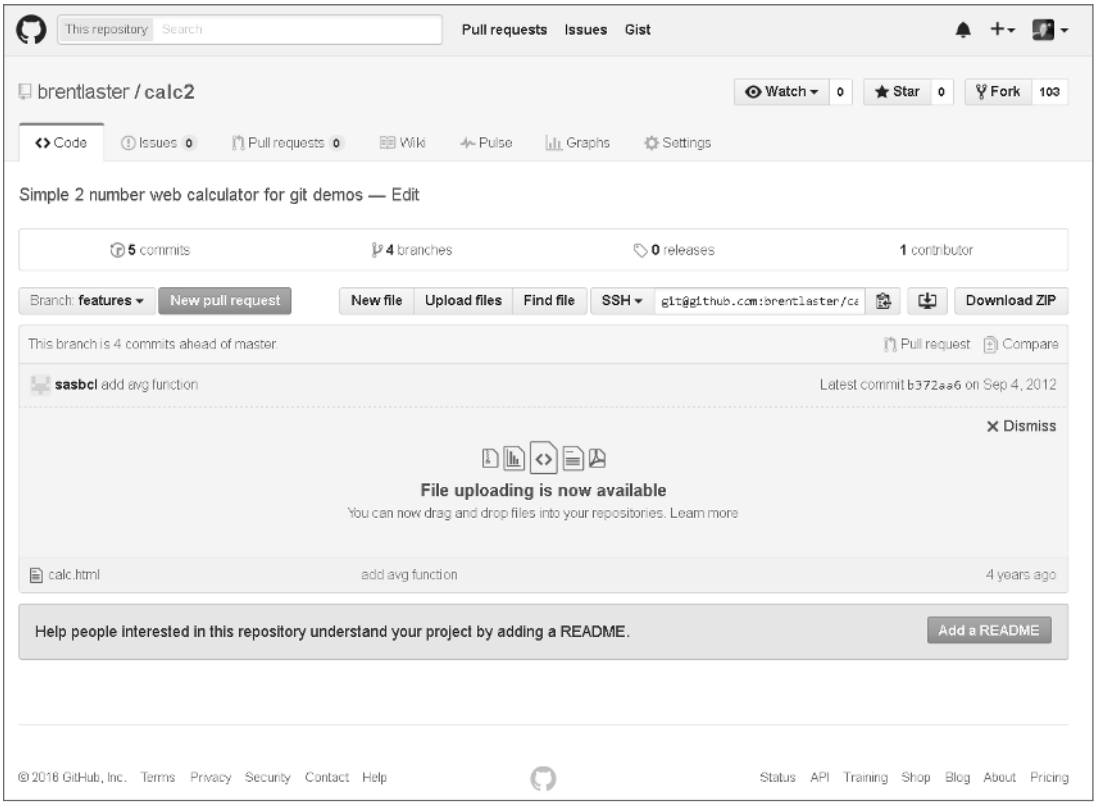

FIGURE 1-1: Example GitHub page

Self-Hosting Packages

Based on the success of the model and usage of the 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 don't 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. The cost structure usually depends on factors relating to the scale of use, such as the number of users or repositories. Figure 1-2 shows an example of a GitLab project screen.

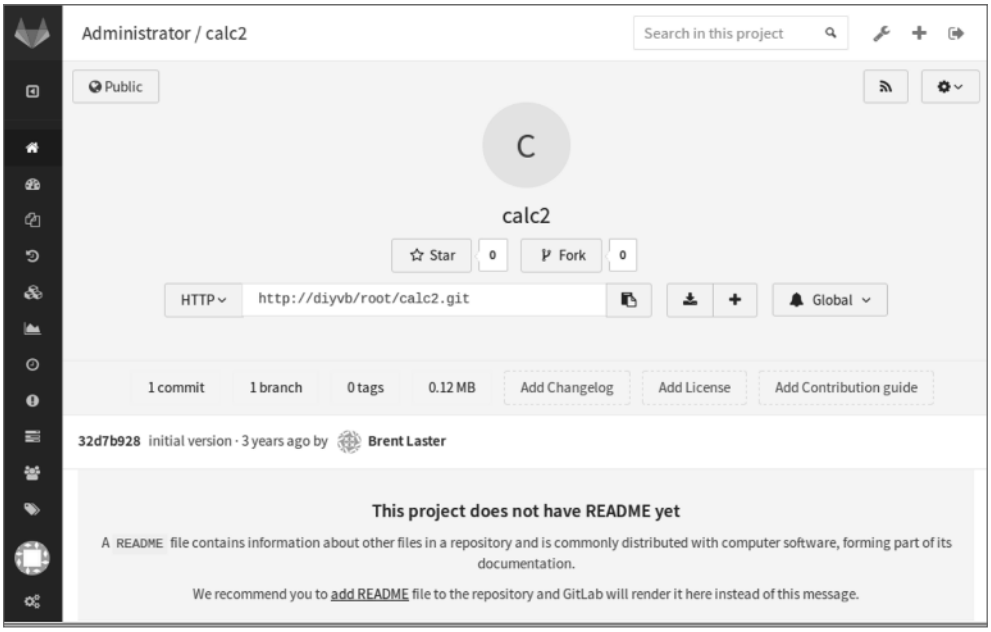

FIGURE 1-2: GitLab project screen

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. In the same way, they often provide graphical tools for labor-intensive operations such as merging.

Examples include SourceTree, SmartGit, TortoiseGit, and 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`.

Figure 1-3 shows some examples of available packages.

CHOOSING AN INTERFACE

One of the questions that frequently comes up when using Git is which stand-alone interface is best. There is no right answer here, but as a good default, the command line provides the most value for a number of reasons.

Although a large number and variety of GUIs are available to use with Git, there is no accepted standard. GUIs come and go, and vary highly in their degree of functionality, completeness, and utility. The command line is consistent and universally applicable.

Not all functionality is exposed through any one GUI for Git. However, all functionality available to users is exposed through the command line. If you need to do something that isn't available through a GUI, you can always drop back to the command line to accomplish it. In addition, Git includes man pages for all command line usage, so help is readily available for that interface.

If you understand the command line operations and options, it's generally easy to translate and map them to the corresponding items in a GUI.

Once you understand the basic command line operation, you'll have more insight into what you want and need to do with a GUI interface. You'll also be in a better position to choose one if desired.

As a side note, one of the main advantages of having a graphical interface with Git is having a graphical merge tool. Git also allows you to configure using a third-party tool for merges from the command line interface. We'll explore configuring merge tools in Chapter 9.

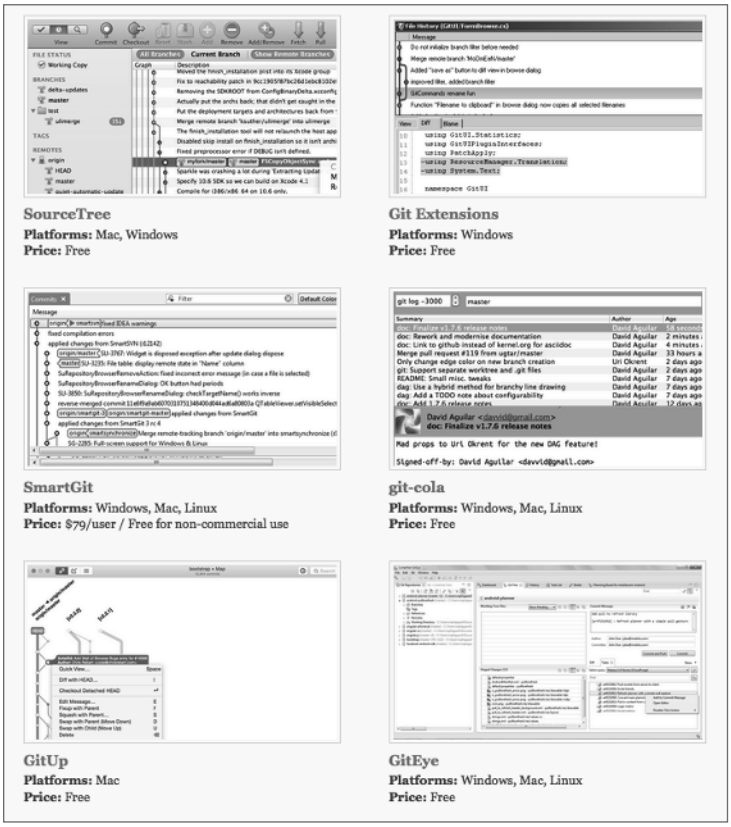

FIGURE 1-3: Examples of GUIs available for Git (from git-scm.org)

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, or Visual Studio, or those that integrate with workflow tools such as Jenkins or TeamCity. It is now becoming more common for applications to include a Git plug-in by default, or, in some cases, to just build it in directly.

Tools That Incorporate Git

Over the past few years, tooling has emerged that directly incorporates and uses Git as part of its model. 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 controls, a code review workflow and tooling, and the ability to configure other validations and checks on the code. Figure 1-4 shows an example of a Gerrit screen.

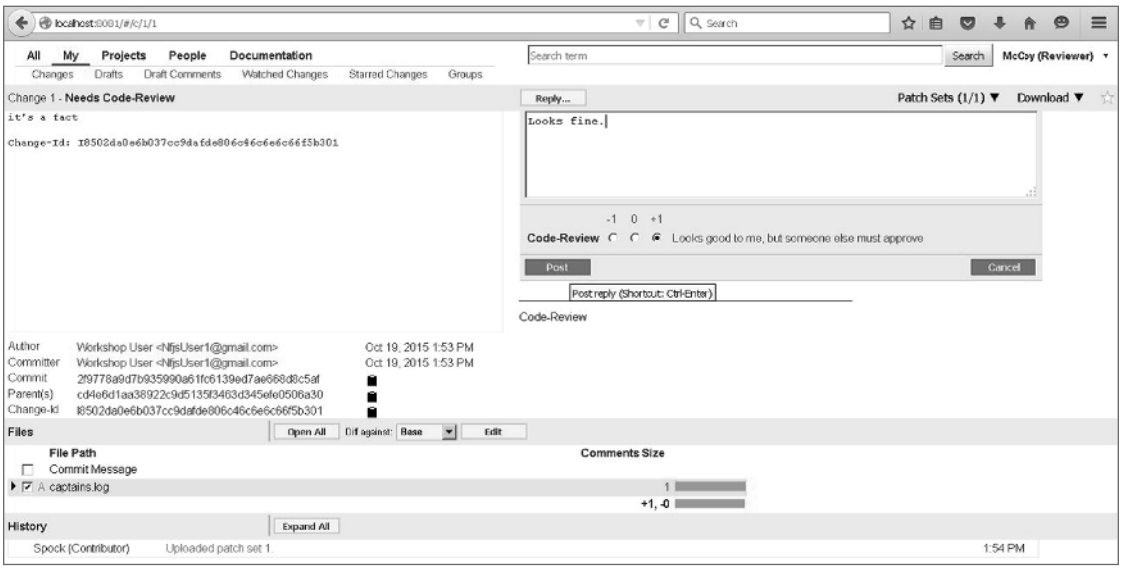

FIGURE 1-4: Example Gerrit screen

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 (mentioned in the previous section). These implementations make interfacing with Git programmatically much more direct. However, there is sometimes a cost in terms of waiting, when new

features or bug fixes that are implemented in the core Git tooling have to be re-implemented in these libraries.

GIT'S ADVANTAGES AND CHALLENGES

Everyone has opinions, and anyone who's tried Git has an opinion about it. These usually vary from believing it's the greatest thing since sliced bread to wondering how they could ever effectively use it. In this section, you'll look at some of the advantages and challenges that Git offers (in no particular order). Granted, these lists are subjective, but themes in each area seem to consistently emerge.

The Advantages

Git is popular for many reasons. There are some things it just does better (faster, easier) than other source management systems and some things that 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.

Disconnected Development

The Git model provides a local environment where you can work with a local copy of a server-side repository (this server-side repository is 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.

All of the source management commands that you need to make changes can be run in this local environment. There's no need to access the remote repository until you're ready to sync content. Because of this, you do not need a connection to the remote repository to conduct source management. You just work against the local copy.

Because you can perform source management tasks in your local environment without needing a connection to the remote-server side, you can work *disconnected* from the remote and even disconnected from a network. This is what *disconnected development* means.

One important factor to keep in mind is that until you sync up with the remote, all of your changes and data are only in the local environment on your system. This is usually the local disk on your machine.

Fast Performance

Git stores a lot of information. (I'll describe its internal storage model in the next chapter.) 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. Of course, this network performance may be mitigated by limiting factors such as network latency, but as a general rule, wait times for Git operations from the server are not a factor.

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 (in most cases not going across a network connection), the performance is equivalent to operating system commands.

Another factor that aids Git's performance is that it is designed to manage multiple smaller repositories—rather than larger aggregate ones that may be present in traditional source control systems. For example, consider how you might store the source code for a large Java project. In a traditional source control management (SCM) system, you might have a single large Java repository with all of the source code in subdirectories for the different JARs. However, in Git you would typically have a separate repository for the source code for each JAR. This granularity contributes to the smaller amount of content that has to be moved around in Git, and thus to a faster operation.

Finally, branching is extremely fast in Git. I'll explain why in Chapter 8, but essentially, as fast as you can create a file on your OS, you can create a branch in Git. 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 conflicts.

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 simple 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. In addition, almost everything about Git settings is configurable so that you can customize your working environment. (Git configuration is discussed in detail in Chapter 4.)

The primary mistake that most new Git users make is trying to use it in the same way that they've always used their traditional source management system. Usually this means that they are trying to map commands and workflow concepts from the previous system to Git's commands. However, trying to adhere too strictly to this approach with Git will actually make the learning curve steeper. A better approach is to consider what sort of source management outcome is needed (files in the repository, viewing history, and so on), and then take the time to learn how that workflow is done with Git. (The Connected Labs included throughout this book will aid this process significantly by providing hands-on experience with Git.)

SHA1s

The strange-looking name SHA1 is an acronym for Secure Hashing Algorithm 1. In short, it's a checksum. (It has its roots in the MD5 implementation if you're familiar with that.) Git computes SHA1s 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. In Git, SHA1s represent a direct way to identify and specify the exact change that you want to work with.

Ability 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, *history*. Git provides

functionality that allows you to traverse previous versions, edit and update them, and place the updated versions back in the same sequence of changes stored in the repository. This is a powerful feature of the tool, but it can also be dangerous (see the section, "The Challenges: Ability to Rewrite History," later in this chapter).

When content that you're working on in your local environment hasn't yet been synched to the remote side, this is a safe operation. And when you need it, it can be very beneficial. For example, consider a case where you forget to include a file with a change, or even just need to do something as simple as modify the message associated with the change. 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 to 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 doing 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.

Staging Area

Git includes an intermediate level between the directory where content is created and edited, and the repository where content is committed. New users typically don't see this extra level as a positive, due to the perceived inconvenience of having to move content through another level. However, it does provide a separate area for use in some of Git's advanced operations, such as the amend option discussed previously. It also simplifies some status tracking. I'll cover the staging area in detail in Chapter 3.

Strong Support for Branching

Using branches is a core concept of Git. Earlier, I mentioned the speed with which users can create, delete, and manipulate branches. However, beyond that, Git provides capabilities for changing branch points and reproducing changes from one branch onto another branch—a feature referred to 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. Later in the book, I devote entire chapters to branching concepts.

One Working Area, Many Branches

It is rare these days for source management users to only be concerned with one release of content. Even 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. Traditionally, the best way to develop these multiple changes in parallel has been in separate workspaces, and, depending on the scope and ease of use of the source management application, in separate branches. With legacy SCM systems, maintaining these multiple workspaces, switching contexts between them, and ensuring they are up to date with the correct source code is a multi-step process that requires tracking and coordination by the user.

In Git, this is a single-step process managed by Git. 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 that workspace. Also, while working in one branch, you still have the expected access to view, merge, or create other branches.

> **WORKING IN MULTIPLE BRANCHES SIMULTANEOUSLY WITH GIT**
>
> If you do find yourself needing to work in multiple branches at the same time, recent versions of Git have introduced a new feature to support this—worktrees (otherwise known as working trees). Worktrees provide a way to have and use multiple working directories with different branches (at the same time) all tied back to the same local Git repository.
>
> We discuss worktrees in detail in Chapter 14.

The Challenges

Now, to balance out the picture, let's look at a few of the things about Git that can be challenging—especially for new users. I'll have more to say about this topic, including what to watch out for, and strategies for effectively dealing with these challenges, throughout the book.

Very Different Model from Some Traditional Systems

Going from a more traditional, centralized version control system to a distributed version control system such as Git requires a change in how you think about your source management workflow. Git implements a local environment with multiple levels in addition to a separate *remote* repository. As well, it operates with units that map more closely to directory tree structures than just individual files. This leads to considerations when creating and working in Git repositories, in terms of size and scope, that you don't usually worry about with centralized systems.

Different Commands for Moving Content

In most traditional source control systems, there are one or two commands for getting content out (checkout) and one or two for putting content in (check-in, commit), with options for modifying their behavior to work in different ways if needed.

With Git, there are different commands for moving content between the different layers, and these commands must be used in a particular sequence. This isn't really an issue after you've been working with Git for a while, and actually is clearer when talking about the workflow. However, it can be a little confusing to new users.

Staging Area

As previously mentioned, Git includes a staging level. This is an intermediate area that new code has to travel through on its way to the local repository. This will seem cumbersome at first, because content must flow through it, even in some situations where it doesn't appear to add value.

However, once you are comfortable with it, it will allow you to work with a power and flexibility that you haven't experienced previously.

Mind Shift and Learning Curve

All of the things I'm talking about as advantages and challenges contribute to the power of Git—as well as the learning curve. As I alluded to previously, one of the fundamental mistakes that new Git users make is trying to map too many concepts and workflows that they've used in the past with other systems, too closely to Git concepts and workflows. They often expect a one-to-one fit, just with different names. The basic principles of source management still apply—tracking changes, putting code in, getting code out, and so on. However, Git adds layers of flexibility and power on top of those principles, at the cost of requiring you to think differently about the units and stages of source control.

This requires a learning curve and a willingness to accept some features and requirements as useful, even if they don't immediately appear so. It's one of those situations where a feature won't seem beneficial until it is. As you continue to use the tool, it's a pleasant experience when you encounter those situations where you need to do X, you wonder if Git can do X, and you discover (in most cases) it can. Of course, there's also a learning curve with figuring out the exact invocation, and implications, of doing X.

Part of the mind shift comes early on in thinking about what should be in your Git repositories and branches. Just converting existing repositories one-to-one from another source management system is seldom the best approach. This is due to the way that Git manages scope in terms of changes and repositories. I'll discuss more about this as you learn more about Git.

Finally, it's worth pointing out that Git offers a built-in way to learn and explore the tool and workflow as you're going through this mind shift and learning curve—the local environment. I'll talk more about this in the next couple of chapters, but for now, know that you have the ability to make any source management changes (and mistakes) you need to in your local environment before you ever push them over to the remote environment, where others can see or access them.

Limited Support for Binary Files

Most source management systems do not have strong support for binary files, and Git is no exception. There are two aspects of dealing with binary files that are challenging here: internal format and 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. Very 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.

Of course, larger text files can also pose size challenges, but with text files, the ability to compute differences between versions and more compressibility can work better with Git's internal strategies for efficiently storing and serving these files.

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. Git Attribute files are covered in detail in Chapter 10.

The challenges with large binary files for source management in general have led to the development of several separate applications to help. Artifact repositories, such as Artifactory and Nexxus, are targeted specifically at storing and managing revisions of binary files. And 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.

No Version Numbers

As referenced in the previous section on SHA1s, Git creates checksums (SHA1s) for everything that it stores. From one perspective, the overall SHA1 value for a change can function like the version number in most other source control systems. However, unlike traditional version or revision numbers, these are not short, easily remembered identifiers. SHA1s are actually 40-character hexadecimal strings. So, from a user perspective, SHA1s are not as convenient to remember, find, or communicate about. Typing one also requires some care.

Fortunately, in any Git instance, you only need to use enough of the characters from any SHA1 to uniquely identify that SHA1 from any other—usually the first seven characters. You can also use other references, such as tags or branch names, to indicate revisions where appropriate.

Merging Scope

While talking about the Git model, I mentioned that Git *thinks* in units that more closely map directory structures than individual files. This difference in granularity provides advantages in managing and manipulating changes in source control. However, it can also create disadvantages in merge situations where there are conflicts. Simply put, 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.

Ability to Rewrite History

Git's ability to rewrite history falls into both categories. On the challenging side of the scale is the potential impact that uncoordinated use can have on other users. Suppose that multiple users have obtained content from a remote (shared) Git repository. One user decides to perform an operation that changes the revision history. Changing the history results in new internal checksums (SHA1s) for changes in the repository, starting at whatever points the revisions were made. Once the updates

are put back on the remote side, any other users that need to merge in updates will have to deal not only with the newest content, but also with the changes to the revisions in the history made by the other user. At best, this can be surprising. At worst, it can be very time-consuming and resource-intensive, because it requires them to incorporate all of the changes.

As a highly recommended guideline, changes that alter history should only be made in a user's local environment *before* the affected revisions are pushed across to the remote side. If there is a critical need to change revisions in the history of a repository after it has been made available on the remote side, then there is a recommended approach: other users should be informed in advance, and given a chance to get their changes in before the changes to the history are made. After the changes are completed, they can get a fresh copy to work with locally. This will allow them to avoid potentially difficult merge situations.

Timestamps

When using most source control systems, timestamps that reflect when changes were made in the repositories are a useful and static property. Given any point in time, it is possible to pull the content from the repository as it was at that point and always get the same set of content on subsequent pulls. Not so with Git. Due to the way that remote repositories are synched 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 synched to the remote.

This means that it's possible to pull content from the remote side based on a particular timestamp and get a certain set of content, then later pull it again based on the same timestamp, and get a different set of content. This can happen if one or more changes were made in a user's local environment, prior to that timestamp, but weren't synched to the remote until between the two pulls.

In this case, a new change with an older timestamp would suddenly show up in the remote. For this reason, you can't rely on timestamps for some of the cases where they are traditionally employed with existing source control systems. I will discuss what the alternative is for Git when I talk more about the remote side in Chapter 12.

Access and Permissions

Out of the box, Git does not provide a layer to set up users or to grant and deny access. For the local environment, this doesn't matter because everything is, well, local. 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
- Limiting access via client-server protocols (SSH, HTTPS)
- Adding an external applications layer that implements a more fine-grained permissions model and interface

Note that these are not mutually exclusive. In a corporate environment that chooses to host its own shared, server-side repositories, for example, you would want to limit who could directly access the actual repositories on disk at the system level, have authentication for users who need to put content into them from their local environments, and potentially have a permissions layer that can be centrally managed or managed by a team within a selected scope.

SUMMARY

In this chapter, I introduced Git, discussed where it came from, and talked about some of the advantages and disadvantages that users should be aware of when working with this tool. Along the way, I also introduced a number of terms and concepts that are part of Git. In subsequent chapters, I will be expanding on and explaining what each of these terms and concepts means, along with teaching you how to use them.

If you're coming from an environment where you used a traditional centralized source control system, you'll find that Git is significantly different and has a learning curve. The workflow is different as well. Trying to map commands, structures, and workflows from your previous system is not an effective strategy. Rather, you should take the time to read through the following chapters and examine the concepts and examples. Equally important is that if you can work through the Connected Labs, they will go a long way toward helping you internalize the concepts, ensure a deeper understanding of the material, and help you be ready to apply Git to your job when you need it.

In Chapter 2, you'll look at some of the primary design concepts that Git uses internally and that are helpful for users to understand before going further with it.