

sitepoint

JUMP START GIT

SECOND EDITION
BY BRUCE A. GRIFFITHS



TAKE CONTROL OF YOUR CODE AND ASSETS

Jump Start Git, Second Edition

Copyright © 2020 SitePoint Pty. Ltd.

Ebook ISBN: 978-1-925836-35-6

- **Product Manager:** Simon Mackie
- **Technical Editor:** Craig Buckler
- **English Editor:** Ralph Mason
- **Cover Designer:** Alex Walker

Notice of Rights

All rights reserved. No part of this book may be reproduced, stored in a retrieval system or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embodied in critical articles or reviews.

Notice of Liability

The author and publisher have made every effort to ensure the accuracy of the information herein. However, the information contained in this book is sold without warranty, either express or implied. Neither the authors and SitePoint Pty. Ltd., nor its dealers or distributors will be held liable for any damages to be caused either directly or indirectly by the instructions contained in this book, or by the software or hardware products described herein.

Trademark Notice

Rather than indicating every occurrence of a trademarked name as such, this book uses the names only in an editorial fashion and to the benefit of the

trademark owner with no intention of infringement of the trademark.



Published by SitePoint Pty. Ltd.

Level 1, 110 Johnston St
Fitzroy VIC Australia 3065
Web: www.sitepoint.com
Email: books@sitepoint.com

About SitePoint

SitePoint specializes in publishing fun, practical, and easy-to-understand content for web professionals. Visit <http://www.sitepoint.com/> to access our blogs, books, newsletters, articles, and community forums. You'll find a stack of information on JavaScript, PHP, design, and more.

About Shaumik Daityari

Shaumik is an optimist, but one who carries an umbrella. He is currently working at American Express as a business analyst. Co-founder of The Blog Bowl, he loves writing, when he's not busy keeping the blue flag flying high.

Preface

Most organizations involved with software development make use of version control. However, despite it being so useful, developers often think of version control as a separate skill, and only learn the bare minimum to get by, or put off learning version control until absolutely necessary. This is to miss out on some of the powerful utilities that version control provides.

This book is about Git—a free, open-source version control system. The aim of this book is to help beginners get up and running with version control quickly, and then to take a deeper dive into its mechanics if they so desire.

Who Should Read This Book?

This book is suitable for anyone interested in managing multiple revisions of code, data and documents. It's ideal for beginners who plan to start working with Git, but it's also useful for seasoned developers who are looking to consolidate their understanding of Git.

Conventions Used

CODE SAMPLES

Code in this book is displayed using a fixed-width font, like so:

```
<h1>A Perfect Summer's Day</h1>
<p>It was a lovely day for a walk in the park.
The birds were singing and the kids were all back
at school.</p>
```

Where existing code is required for context, rather than repeat all of it, `:` will be displayed:

```
function animate() {  
    :  
    new_variable = "Hello";  
}
```

Some lines of code should be entered on one line, but we've had to wrap them because of page constraints. An `↪` indicates a line break that exists for formatting purposes only, and should be ignored:

```
URL.open("http://www.sitepoint.com/responsive-web-  
↪design-real-user-testing/?responsive1");
```

You'll notice that we've used certain layout styles throughout this book to signify different types of information. Look out for the following items.

TIPS, NOTES, AND WARNINGS

Hey, You!

Tips provide helpful little pointers.

Ahem, Excuse Me ...

Notes are useful asides that are related—but not critical—to the topic at hand. Think of them as extra tidbits of information.

Make Sure You Always ...

... pay attention to these important points.

Watch Out!

Warnings highlight any gotchas that are likely to trip you up along the way.

Supplementary Materials

- <https://github.com/spbooks/jsvuejs1> is the book's code archive, which contains code examples found in the book.
- <https://www.sitepoint.com/community/> are SitePoint's forums, for help on any tricky problems.
- **books@sitepoint.com** is our email address, should you need to contact us to report a problem, or for any other reason.

Chapter 1: Introduction

Introduction

In my freshman year in college, I started work on my first intranet application. The files in the main directory of the partially functioning application looked something like Figure 1-1.

```
ajaxify1.js      db_struct      examfile.php   gre_day.php
ajaxify2.js      display.php    examfile1.php  header.php
auth.php         donny@192.168.121.160  faqs.php       images
check.php        dump.sql       fav1.php        index.php
ckeditor         dump.sql.1     favfile.php     instructions.php
confirmation.js  dump.sql.2     favourites.php  left_col.php
connection.php   exam.php       footer.php      login.php
database.php     exam1.php      get_name.php    logout.php
```

Looking at the file names in this directory, you can see that I used some very similar names, such as `exam.php`, `exam1.php` and `examfile.php`. The purpose of that naming convention was to create new versions of my application without losing the old, working logic—in case the new ideas failed! I assumed that, because I understood what each of those files did, it should be fine to have a bunch of similarly named files.

However, there were two flaws in that thinking. Firstly, anyone else examining this code wouldn't be able to make sense of this mess. Secondly, after a few months, *even I* was struggling to recall what each version of these files was for. Clearly, I needed a better system for managing the various versions of my files.

If I had this much trouble working on a small, personal project, imagine how difficult it must have been for larger software projects, with thousands of files and contributors distributed all over the world! Developers once used emails to coordinate changes among team members. When they made changes to a project, they

would each create a “diff” file with all their changes and email it to the lead developer, who would incorporate them into the project if everything worked properly.

When you’re working on the same files as other developers, keeping track of what you’ve changed and trying to merge it with work done by your peers becomes very difficult. It can result in a lot of confusion and time wasting.

Imagine another situation, where you’re working on an idea and your boss wants to see what you’ve already completed. Ideally, you’d want to be able to do the following:

- stash away the changes and revert to the last stable state
- show your boss the latest completed work
- resume your work with the current state once that’s done

All of the situations I’ve described above give rise to the need for what’s known as “version control”. So let’s find out what that is.

Version Control

Version control (or **revision control**) is a system that records changes to a file or a group of files and directories over time, so that you can review or go back to specific versions later. Over the course of this book, I’ll demonstrate how this works. But first, let’s examine in more detail what version control is.

Quite literally, version control means maintaining versions of your work—perhaps most commonly in the form of source code, though it can be used for other kinds of work too. You may like to think of version control as a tool that takes snapshots of your work across time, creating checkpoints. You can return to those checkpoints any time you want. Not only are the changes

recorded in these checkpoints, but also information about who made the changes, when they made them, and the reasons behind the changes.

I've already mentioned the first objective of version control—to back up and restore. Version control eliminates the need to create backup files like I was doing in my college days (that is, endless duplicates with different names). Version control also gives you the ability to return to previous states of your work without losing the current state.

Version Control Doesn't Replace the Need for a Regular Backup Solution

The word "backup" above, as noted, refers to the process of creating multiple copies of the same file. Git removes the need for that. However, this is different from regularly backing up your files to an external source—such as a portable drive or cloud storage—to ensure you don't lose anything following a disk failure.

Next, version control lets you synchronize your work with peers who are working on the same projects. In other words, it enables you to collaborate with others without the possibility of someone's changes overwriting someone else's work accidentally.

Version control also tracks changes to a project and other data associated with the changes. It makes the process of debugging your code easy too, which we'll explore in some detail.

Conflicts in files can also be resolved through version control—such as when multiple people have made changes to a file that clash. A version control system highlights the conflicts and provides an opportunity to fix them.

Yet another feature of version control is that it enables work on multiple features of a project at the same time. This gives great scope for experimentation, trial and

error. Each feature can be developed independently of the others, and can easily be removed if it doesn't work out.

Now that you've been introduced to the concept of version control, let's look at how we may already be using version control in our daily lives.

Examples of Version Control in Daily Life

You've probably visited the [Wikipedia](#) site at some point. You may even have taken the opportunity to update its content, too—as we're all invited to do. When editing a page, you may also have checked its history. That's where things get really interesting.

Article [Talk](#)

[Read](#) | [View source](#) | [View history](#)

B. R. Ambedkar: Revision history



[View logs for this page](#) ([view filter log](#))

▼ **Filter revisions**

External tools: [Find addition/removal](#) ^(Alternate) · [Find edits by user](#) · [Page statistics](#) · [Pageviews](#) · [Fix dead links](#)

For any version listed below, click on its date to view it. For more help, see [Help:Page history](#) and [Help:Edit summary](#). (cur) = difference from current version,

(prev) = difference from preceding version, m = minor edit, → = section edit, ← = automatic edit summary

(newest | oldest) [View](#) (newer 50 | older 50) (20 | 50 | 100 | 250 | 500)

Compare selected revisions

- [\(cur | prev\)](#) 10:53, 4 March 2020 [InternetArchiveBot](#) (talk | contribs) .. (94,962 bytes) (+89) .. *(Blueinking 1 books for verifiability)* #iABot (v2.1alpha3)
- [\(cur | prev\)](#) 19:27, 27 February 2020 [Aman.kumar.gpai](#) (talk | contribs) .. (94,873 bytes) (-8,206) .. *(RV POV pushing with shoddy sources and original research)* (Tags: Mobile edit, Mobile web edit)
- [\(cur | prev\)](#) 09:28, 27 February 2020 [MaterialsScientist](#) (talk | contribs) .. (103,078 bytes) (+385) .. *(→In popular culture: restore)*
- [\(cur | prev\)](#) 05:12, 26 February 2020 [ML MEGHWANSHI](#) (talk | contribs) .. (102,693 bytes) (0) .. *(Updated)* (Tags: Mobile edit, Mobile web edit)
- [\(cur | prev\)](#) 05:00, 26 February 2020 [ML MEGHWANSHI](#) (talk | contribs) .. (102,693 bytes) (+497) .. *(Updated)* (Tags: Mobile edit, Mobile web edit)
- [\(cur | prev\)](#) 04:15, 26 February 2020 [ML MEGHWANSHI](#) (talk | contribs) .. (102,196 bytes) (+497) .. *(Updated)* (Tags: Mobile edit, Mobile web edit)
- [\(cur | prev\)](#) 04:02, 26 February 2020 [ML MEGHWANSHI](#) (talk | contribs) .. (101,699 bytes) (+4,396) .. *(Updated)* (Tags: Mobile edit, Mobile web edit)
- [\(cur | prev\)](#) 05:28, 25 February 2020 [Siddsg](#) (talk | contribs) .. (97,303 bytes) (-6) .. *(→Political career)*
- [\(cur | prev\)](#) 18:55, 24 February 2020 [Siddsg](#) (talk | contribs) .. (97,309 bytes) (+2,427) .. *(→Political career)*
- [\(cur | prev\)](#) 11:10, 19 February 2020 [Aman.kumar.gpai](#) (talk | contribs) .. (94,882 bytes) (-238) .. *(Revert POV pushing)* (Tags: Mobile edit, Mobile web edit)
- [\(cur | prev\)](#) 08:04, 16 February 2020 [Suryakant Sadu](#) (talk | contribs) .. (95,120 bytes) (+74) .. (Tags: Mobile edit, Mobile web edit)
- [\(cur | prev\)](#) 07:45, 16 February 2020 [Suryakant Sadu](#) (talk | contribs) .. (95,046 bytes) (+88) .. (Tags: Mobile edit, Mobile web edit)
- [\(cur | prev\)](#) 07:09, 16 February 2020 [Suryakant Sadu](#) (talk | contribs) .. (94,958 bytes) (+4) .. (Tags: Mobile edit, Mobile web edit)
- [\(cur | prev\)](#) 06:47, 16 February 2020 [Suryakant Sadu](#) (talk | contribs) .. (94,954 bytes) (+16) .. (Tags: Mobile edit, Mobile web edit)
- [\(cur | prev\)](#) 06:42, 16 February 2020 [Suryakant Sadu](#) (talk | contribs) .. (94,938 bytes) (0) .. (Tags: Mobile edit, Mobile web edit)
- [\(cur | prev\)](#) 06:30, 16 February 2020 [Suryakant Sadu](#) (talk | contribs) .. (94,938 bytes) (+2) .. *(→Conversion to Buddhism)* (Tags: Mobile edit, Mobile web edit)
- [\(cur | prev\)](#) 06:29, 16 February 2020 [Suryakant Sadu](#) (talk | contribs) .. (94,936 bytes) (-4) .. *(→Conversion to Buddhism)* (Tags: Mobile edit, Mobile web edit)

The history page shown in Figure 1-2 lists changes to that page. It also records the time of the change, the user who made it, and a message associated with the change. You can examine the complete details of each edit, and even revert back to an older version of the page. This is a good example of a simple form of version control.

The screenshot shows a Google Docs document titled "API Calls" with a revision history sidebar open on the right. The document content includes two GET API endpoint examples with their descriptions. The revision history sidebar lists several changes, with the most recent one highlighted in red.

Document Content:

```
GET /api/students/{student_id}/courses?title={title}&release_date={release_date}&created_date={created_date}&category_id={category_id}&primary_language={primary_language}
Get all courses to which a student is enrolled to. Available to admin, or student himself. Can be used by student to manage his courses.
```



```
GET /api/instructors/{instructor_id}/courses
GET /api/instructors/{instructor_id}/courses/{course_id}
GET
/api/instructors/{instructor_id}/courses?title={title}&release_date={release_date}&created_date={created_date}&category_id={category_id}&primary_language={primary_language}
Get all courses which instructor is teaching. Available to instructor (for himself) and admins. Instructor can manage his courses.
```

Revision History:

Time	User
21 June, 23:08	Shaunik Dabiyari
21 June, 00:08	Shaunik Dabiyari
5 June, 21:12	Shaunik Dabiyari
5 June, 19:44	Shaunik Dabiyari
3 June, 00:15	Alexey Novak
2 June, 23:25	Shaunik Dabiyari
29 May, 18:43	Shaunik Dabiyari
29 May, 18:03	Shaunik Dabiyari
29 May, 00:05	Shaunik Dabiyari
28 May, 01:33	Shaunik Dabiyari
24 May, 19:53	Shaunik Dabiyari

Google Docs provides another example of version control that you might experience in daily life. If you check the revision history of a file in Google Docs, shown the figure

above, you'll notice that Google saves the state of your file after every few changes. You can preview the status of the document in any of those previous states—and choose to revert back to it, if needed.

Version Control Systems: the Options

There are two types of version control systems (VCS), known as “centralized” and “distributed”.

Centralized systems have a copy of the project hosted on a centralized server, to which everyone connects to in order to make changes. Here, the “first come, first served” principle is adopted: if you're the first to submit a change to a file, your code will be accepted.

In a **distributed** system, every developer has a copy of the entire project. Developers can make changes to their copy of the project without connecting to any centralized server, and without affecting the copies of other developers. Later, the changes can be synchronized between the various copies.

In the earliest version control systems, files were tracked only locally, and only one person could work on a file at a time. Examples of these include Source Code Control System (SCCS) and Revision Control System (RCS), which were common in the 1970s and 1980s.

The next step forward was the introduction of client-server version control systems, which enabled multiple authors to work on the same file (although some still worked on the first come, first served basis). Examples of such systems include Concurrent Versions System (CVS) and Subversion, which are still in use today.

Since around 2005, distributed systems have gained widespread acceptance, with the emergence of systems

such as Git, Mercurial and Bazaar.

VCS Is Not CVS

Don't confuse the abbreviations VCS (version control system) and CVS (concurrent versions system). CVS is just one of the many kinds of VCS.

Back in my freshman year, version control systems were available. However, in the example of my small project, I didn't use one, simply because I was a beginner and didn't know they existed. Many people first get introduced to version control systems when they start working with a team. Nowadays, most people get the first taste of version control when dealing with open-source projects.

Enter Git

This book is about Git, a distributed version control system. Git tracks your project history, enabling you to access any version of it back in time. It also allows multiple people to work on the same project, helping avoid confusion when more than one person tries to edit the same file.

Git was created by Linus Torvalds (who is also known for the Linux kernel), and Junio Hamano is its primary developer. Git, as described on the Git website, is a source code management (SCM) solution, but essentially it's just a type of version control system.

The primary objective behind Git was to implement and design a version control system that was distributed, reliable and fast. While working on Linux, Torvalds needed a version control system to manage the Linux codebase. BitKeeper was a distributed system at that time, but Torvalds believed that, although BitKeeper was a good option, being a commercial product made it

unsuitable for the development of an open-source project like Linux.

Torvalds had three criteria for a version control system: it had to be distributed, efficient and safe from corruption. There was no open-source, distributed version control system in the mid 2000s that could satisfy all these conditions. Hence, Git was developed out of necessity.

Git's Philosophy

Torvalds once explained in a [Google Tech Talk](#) his reasons for creating Git. He has very strong views on the subject of version control, and I suggest you go through the talk once to understand the philosophy of Git. In this talk, Torvalds explains that he came up with the name Git because he believes the silliest names are our best creations. However, I recommend that you only watch the talk after you're comfortable with the basic Git operations, as it's not a tutorial: it's aimed at users who have some knowledge of Git or other version control systems.

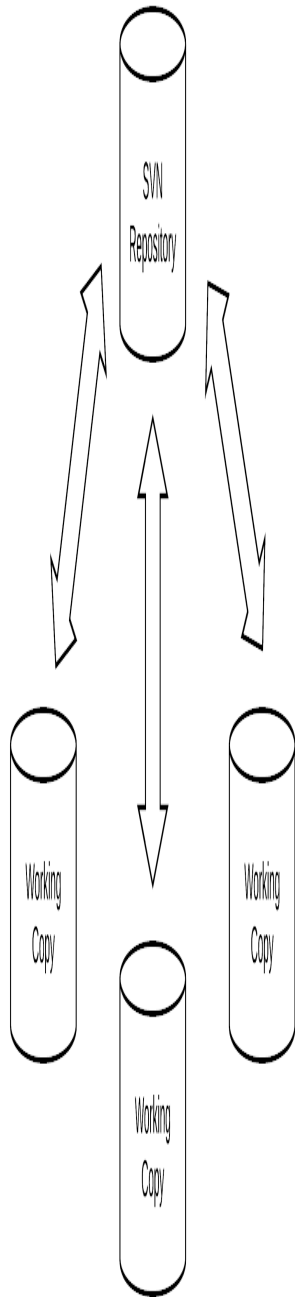
ADVANTAGES OF DISTRIBUTED VERSION CONTROL SYSTEMS

Torvalds insisted on a distributed system because of the independence it affords to developers. With a distributed system, you can work on your copy of the code without having to worry about ongoing work on the same code by others. What makes it even better is that any distributed copy of the project can contain all the history of the project. A distributed system also lets you work offline, meaning you can make changes without having access to the server that stores the central repository.

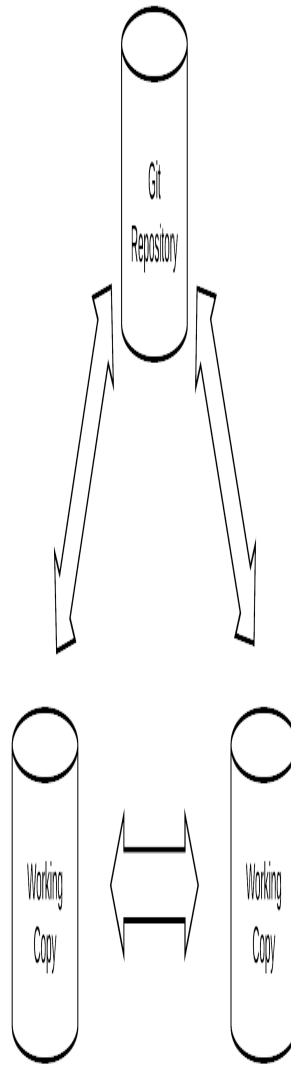
Another advantage of distributed systems is that you can sync your repositories among yourselves, bypassing the central location. Let's say the access to the main server goes down and you have to collaborate with a colleague. You can share changes with your colleague and continue to work on the project together, and then later push all your changes to the location everyone has access to.

In a centralized system, anyone who makes a change needs to be given access to the central location. In contrast, in a distributed system, new developers can make changes to their own repositories without being granted write access, while more experienced contributors can be given write access and the ability to review other contributions before merging them into the repository. Managing access is easier in distributed systems.

Central Repository to
Client Collaboration



Collaboration between
any two repositories



Git and GitHub

Since its creation, Git has become immensely popular—not only due to its own merits and the fact that Torvalds created it, but also because of the popular code sharing site [GitHub](#).

People often confuse Git and GitHub, but they are quite different things. GitHub provides services that are *related to* Git. It's a website that helps you manage Git-controlled projects.

GitHub allows users to put their Git repositories on the cloud, and to perform Git-based operations through a web interface. It also provides desktop and mobile apps that offer the same services. GitHub was launched a few years after Git, and remains very popular among enthusiasts of open source.

There are many other websites like GitHub, such as [Bitbucket](#) and [GitLab](#). GitHub and Bitbucket are cloud-based solutions, but GitLab allows you to set up this functionality on your own servers. Other, similar services have come and gone, but these options have remained popular over the last few years. We'll explore these code sharing websites in a later chapter, and discuss how you can make use of them.

Conclusion

WHAT HAVE YOU LEARNED?

- What is version control?
- How do we unknowingly use version control in our lives?
- What are the types of VCS?
- What is Git? What are its capabilities?

WHAT'S NEXT?

Now that we have a basic concept of what a version control system does, let's get our feet wet with Git. In the next chapter, we'll look at how to install Git and use it in a project.

Chapter 2: Getting Started with Git

The first step is to install Git. Git's official website provides detailed instructions on [installing Git on your local machine](#), depending on your operating system.

The easiest way to install Git is through a package manager based on your operating system. Package managers usually have older but more reliable versions of Git.

- If you're using Linux, you can install Git through the terminal using a package manager. For the popular Linux distro Ubuntu, Git can be installed using `apt-get`:

```
apt-get install git
```

- In macOS, if you have [Homebrew](#), you can install Git using the command line through the following command:

```
brew install git
```

- If you're on Windows, [the official build of Git](#) can be downloaded from the Git website.

GUI Tools

For Windows and macOS, you can also install Git as a part of a GUI tool such as [GitHub for Desktop](#) and [Sourcetree](#). We'll cover GUI tools in detail in [Chapter 9](#). However, for most parts of this book, we'll stick to the command-line interface to really understand how Git works.

If you're using an operating system other than these three, like [Minix](#) or [HelenOS](#), or if you want to get the latest development version of Git for testing and development, you can install Git from its source. Grab a tarball of the desired version of Git from [GitHub](#), untar it and check the README file for instructions on how to install Git. However, I wouldn't recommend following

this process unless you know what you're doing, as it can lead to errors, and development versions may be unstable.

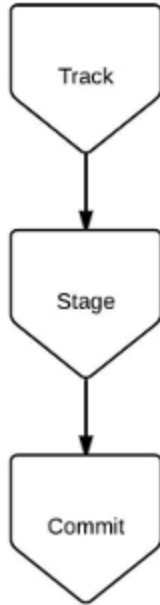
The Git Workflow

Git doesn't track all of the files stored on your computer. You need to instruct Git to track certain files and directories. This process is called **initialization**. The parent directory containing your project—all the files and directories to be tracked by Git—is called a **repository**. This repository might contain many files and directories, or even just a single file.

There are three basic operations performed by Git on your project (shown in Figure 2-1 below): track, stage, and commit.

- **Track.** Once you've initialized your repository, you'll need to add files to your project. Any files you add are initially untracked by Git. You need to specify that you want Git to **track** them. Git monitors tracked files for changes and ignores untracked files.
- **Stage.** After making the required changes to your files, you need to **stage** them. Staging is a way of tagging certain (or all) changes that you want to keep a record of.
- **Commit.** The next step is to create a **commit**. A commit is like a photograph that records the current state of your code. You can go back to a certain commit at a later time, view the status of the repository with respect to that commit, and check the changes that were made in the commit. The commit records the changes in a repository since the last commit. You can revert back to any commit at any point of time. Each commit contains a **commit hash** that uniquely identifies the commit, the author details, a commit message, and the list of changes in that commit.

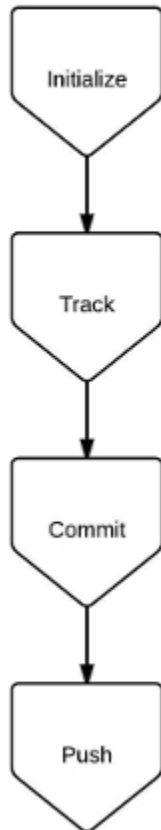
Commit Process



Once you've committed your files, you may wish to **push** them to a remote location. A push refers to the process of sending the changes you've made in your local repository to a remote location. A remote location is a copy of your repository stored on a remote server. (We'll set up a remote repository later in this chapter.)

Essentially, the flow chart in Figure 2-2 below illustrates the steps we'll follow in this chapter.

Git Workflow



Baby Steps with Git: First Commands

SET CONFIGURATION SETTINGS

Before we proceed with using Git in a project, let's define a few global settings:

```
git config --global user.name "Shaumik"  
git config --global user.email  
"sdaityari@gmail.com"  
git config --global color.ui "auto"
```

The commands are fairly self-explanatory. We set the default name and email to be associated with our

commits. We also set the `color.ui` to "auto" to enable Git to color-code the output of Git commands on the terminal. The `--global` setting allows these settings to be applied to any other repository you work on locally.

If you don't set the values for name and email, they're left empty. When you make a commit, it takes different values depending on the OS or the GUI tool you use. When you make a commit without setting these parameters, Git will automatically set them based on the username and hostname. For instance, the name is set to the name of the user that's logged in to the computer in macOS, whereas in Linux, the name is set to be the username of the active user account. In both cases, the email is set as `username@hostname`.

If you want to check all the configuration settings for your repository, you can run the following command:

```
git config --list
```

Also, if you want to edit any of your configuration settings, you can do so by editing the `~.gitconfig` file in Linux and macOS, where `~` refers to your home directory. In Windows, it's located in your home directory: `C:\Users\<<username>\.gitconfig`.

CREATE A GIT PROJECT

Let's first create a directory where we'll store the files for our project:

```
mkdir my_git_project  
cd my_git_project
```

The first command creates a new directory, and the second changes the active directory to the newly created one. These two commands work on all operating systems (Windows, macOS, and Linux).

So, `my_git_project` is the parent directory that will contain all the files for this project. From now on, we'll refer to it as our project's repository.

Now that we're in the repository, we need to initiate Git for that directory using the following command:

```
git init
```

Issuing Git Commands

Just like `git init`, all Git commands start with the keyword `git`, followed by the command.

Git Autocomplete

When working in the terminal, developers often use the `Tab` key for autocomplete. However, this doesn't work on Git commands by default. You can install an autocomplete script for Git using the following commands. Note that this only works on Linux and macOS.

- Download the autocomplete script and place it in your home directory:

```
curl
https://raw.githubusercontent.com/git/git/master/
contrib/completion/git-completion.bash -o
~/.git-completion.bash
```

- Add the following lines to the file `~/.bash_profile`:

```
if [ -f ~/.git-completion.bash ]; then
. ~/.git-completion.bash
fi
```

If you're using Git Bash on Windows, autocomplete is preconfigured. If you're using Windows command prompt (`cmd.exe`), you'll need to install [Clink](#).

CREATE OUR FIRST COMMIT

Let's look at the repository again. Notice the newly created `.git` directory, the output of which is shown below (line 2). All information related to Git is stored in this repository. The `.git` directory, and its contents, are normally hidden from view:

```
$ my_git_project shaumik$ git init
Initialized empty Git repository in
/Users/shaumik/test/my_git_project/.git/
$ my_git_project shaumik$ ls -al
total 0
drwxr-xr-x  3 shaumik  staff   96 Mar 21 23:05 .
drwxr-xr-x  3 shaumik  staff   96 Mar 21 23:05 ..
drwxr-xr-x  9 shaumik  staff  288 Mar 21 23:05
.git
```

Don't Edit .git

Never edit any files in the `.git` directory. It can corrupt the whole repository. This book doesn't discuss the internals of Git, and thus doesn't include working on this hidden `.git` directory.

Now that we've initialized Git, let's add a few files to our repository. On your computer, navigate to the `my_git_project` directory and add three text files with the following names: `my_file`, `myfile2` and `myfile3`. Place some content in each one, such as a simple sentence.

Demonstration Only

The file names `my_file`, `myfile2` and `myfile3` are used for demonstration purposes. They signify three different files and not different versions of the same file.

After adding the files, let's return to the terminal and run the following command to see how Git reacts:

```
git status
```

You can see the output below:

```
$ git status
On branch master

No commits yet

Untracked files:
  (use "git add <file>..." to include in what will
  be committed)
```

```
my_file
myfile2
myfile3
```

```
nothing added to commit but untracked files
present (use "git add" to track)
```

Checking the Status

`git status` is perhaps the most-used Git command—as you’ll see over the course of this book. In simple terms, this command shows the status of your repository. It provides a lot of information, such as which files are untracked, which are tracked and what their changes are, which is the current “branch”, and what the status of the current branch is with respect to a “remote” (we’ll discuss branches and remotes later). You should frequently check the status of your repository.

In a Git repository, any file that’s added is either “tracked” or “untracked”. A file is said to be **tracked** when Git monitors the changes being made to that file, whereas the changes to an **untracked** file are ignored by Git and don’t form a part of any commits.

Checking the status of our repository, we can see that three files are currently marked in red. They’re also grouped as untracked. Git doesn’t track all files in a repository. You can explicitly tell Git which files to track and which to ignore.

In order to track these files, we run the following command:

```
git add my_file myfile2 myfile3
```

As an alternative, you can simply run the following:

```
git add .
```

The `.` (period) is an alias for the current directory. Running `git add .` tells Git to track the current directory, as well as any files or subdirectories within the current directory.

Beware of Adding Unwanted Files

Don't make a habit of using `git add .`, as you may end up adding unnecessary files to the repository. You should add only those files that are a part of your package. Adding files like compiled files and configuration files just increases the size of your repository. Configuration files may also contain database passwords, which could lead to a security risk if committed to a repository that's open to the public.

Now that we've set our new files to be tracked by Git, let's check the status of the repository again:

```
$ git add my_file myfile2 myfile3
$ git status
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
   new file:   my_file
   new file:   myfile2
   new file:   myfile3
```

We're now ready to make a commit:

```
$ git commit -m "First Commit"
[master (root-commit) ed90340] First Commit
 3 files changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 my_file
 create mode 100644 myfile2
 create mode 100644 myfile3
```

The `-m` option specifies that you're going to add a message within the command. (The message is the text in quotes after `-m`: "First Commit".) Alternatively, you can just run `git commit`, and a text editor will open up and ask you to enter a commit message.

Make Your Commit Messages Meaningful!

A meaningful commit message is an essential part of your commit. You can give a meaningless commit message like "Commit X", but in the future, it might be difficult for someone else (or even you) to understand why you created that commit.

Notice the string `ed90340` shown in the code above (second line). It's the hash of the commit, or its identity. A **hash** is a unique, identifying signature for each commit, generated automatically by Git. If you're interested in how a commit hash is formed, you may want to check out [“The anatomy of a Git commit”](#). What's shown here is a short version of a considerably longer string, which we'll look at further below.

Further Commits with Git

The first commit in a Git repository is a little different from subsequent commits. In subsequent commits, Git is already tracking the files you're working on (unless you're adding new files). So we'll need another important command, `git diff`, which shows you the changes in the tracked files since the last commit.

Let's make some changes to the files and see how Git reacts. For demonstration purposes, I've added a line to `my_file`, and some extra words to an existing line in `myfile2`. Let's check the status of the repository by running the following command:

```
$ git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be
  committed)
  (use "git restore <file>..." to discard changes
  in working directory)
    modified:   my_file
    modified:   myfile2
```

As shown here, Git shows that certain changes have been made to two files. We can also see exactly what was changed in the files by running the following command:

```
$ git diff
diff --git a/my_file b/my_file
index e69de29..e32ce9e 100644
--- a/my_file
+++ b/my_file
```

```
@@ -0,0 +1 @@
+Sample line
diff --git a/myfile2 b/myfile2
index e69de29..d00491f 100644
--- a/myfile2
+++ b/myfile2
@@ -0,0 +1 @@
-Some more info
+Some more info! Changing this file too.
```

The `diff` command shows the changes that have been made to the tracked files in the repository since the last commit. In the output shown above, lines starting with a `+` sign (colored green) show what's been added, and the line starting with a `-` sign (colored red) shows what's been removed. (When you edit a line of code, the same thing happens: the old line is shown in red with a `-` sign, and the new version of the line is shown in green with a `+`.)

If you want to check the changes in a single file, add the file name after the `diff` command. For instance:

```
git diff my_file
```

Diff Only Shows Changes in Tracked Files

As mentioned earlier, Git tracks only the files that you ask it to. The `git diff` command shows the changes only in tracked files.

After you've reviewed the changes you made, you need to "stage" the changes to be committed:

```
git add my_file myfile2
```

Alternately, you can add all tracked files like so:

```
git add -u
```

You can go one step further and add only parts of the changes to a file to the commit. This process is a bit

complex, though, and we'll tackle it in [Chapter 6, "Correcting Errors While Working with Git"](#).

Now that you've staged the files, they're ready to be committed:

```
git commit -m "Made changes to two files"
```

Beware of Shortcuts

You can skip the adding (staging) of a modified file by postfixing `-a` to the `git commit`, which performs the add operation. However, you should avoid doing this, because it can lead to mistakes. Firstly, postfixing `-a` only adds tracked files—so you'd miss any untracked files that you may have wanted in the commit. Secondly, it may be that you've modified two files but want them to appear in separate commits. A `git commit -a` would add both files to the same commit.

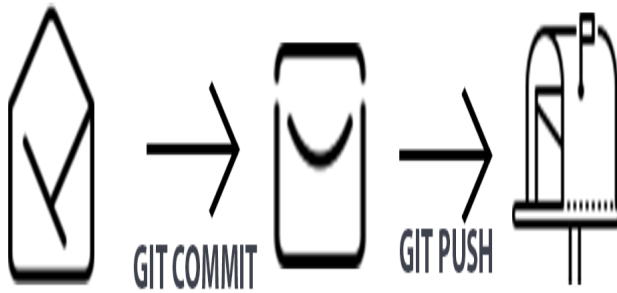
Always Review Your Changes

I mentioned earlier that `git status` is perhaps the most-used command. However, the most important command is probably `git diff`. Never stage files for commit before reviewing the changes you've made in them. Also, stage files for commit individually after carefully reviewing the changes that were made to them.

WHY GIT ADD AGAIN?

At this point, you may think "Why add tracked files again?" Well, before you commit, Git needs you to specify which files you want to commit. It may happen that you've made changes to two files but only want to commit one of those files.

The process is like sending a package. `git add` is adding an item to the package. `git commit` is sealing the package and writing a note on it. `git push` (which I'll explain shortly) is sending the package to the recipient.



Commit History

Now that we have more than one commit, let's explore a new area of Git—the **history** of the project. The simplest way of reviewing the history of a project is by running

`git log`, which shows the commits that we've made so far:

```
$ git log
commit 870e4d76e6dc6539315992f16a20f47a49e2ea79
(HEAD -> master)
Author: Shaumik Daityari <sdaityari@gmail.com>
Date:   Sat Mar 21 23:31:16 2020 +0530

    Made changes to two files

commit ed90340105b9511381d76706f8e5d4e7df3f6458
Author: Shaumik Daityari <sdaityari@gmail.com>
Date:   Sat Mar 21 23:16:28 2020 +0530

    First Commit
```

The history shows the list of commits, each with a unique hash, an author, a timestamp and a commit message.

Previously in this chapter, we encountered a commit hash that was truncated. Although the long, 40-character commit hash uniquely identifies each commit, usually five or six characters are enough to identify them in a repository:

```
git show ed90340
```

The `git show` command lists information about a commit. Let's see how short we can go until Git fails to identify the hash:

```
git show ed90340
git show ed9034
git show ed903
git show ed90
git show ed9
```

It's only once we're down to the first three characters that Git gives us a fatal error:

```
ambiguous argument 'ed9': unknown revision or path
not in the working tree.
```

Although it only failed at three characters in our repository with a very short history, it will probably need to be longer in repositories with a considerably longer history.

The `.gitignore` File

Although I've mentioned that Git only tracks files you explicitly ask it to, it could happen that you ask it to track some files by mistake. You need a way to hide certain files, directories, or file extensions from Git that you know you'll never want it to track. This is exactly what a `.gitignore` file does.

A `.gitignore` file is added to the root directory of the repository, and it lists files you don't want Git to track or display as part of `git status`. You can add items to the `.gitignore` file and commit them.

Unintentionally Tracking a File Listed in `.gitignore`

Although a file listed in `.gitignore` is not meant to be tracked, it's possible that you could accidentally tell Git to track a file that's listed in there. In earlier versions of Git (before Git 1.5.3.6), you won't get any error message. This is another reason you should avoid running `git add .`, as it may cause files to be tracked by Git unintentionally.

Examples of files that you might want to add to `.gitignore` include compiled files with extensions like `.exe` and `.pyc`, local configuration files, macOS `.DS_Store` files, `Thumbs.db` on Windows, directories of node modules in Node.js, and build folders of Grunt or gulp.js.

Let's have a look at what a `.gitignore` file looks like:

```
configuration/  
some_file.m  
*.exe
```

The three lines in this sample file are used to tell Git to ignore a whole repository and its contents (the configuration directory), a single file `some_file.m`, and all files with a `.exe` extension.

The code sample below shows the effect of a `.gitignore` file that tells Git to ignore `*.exe` files that have already been committed to the repository. I've created a new file called `somefile.exe` in our project directory, but Git is ignoring it. `git status` shows that there is nothing to commit:

```
$ echo "some line" > somefile.exe
$ git status
On branch master

nothing to commit, working tree clean
```

Hiding `.gitignore` from Git

Although it's advisable to add the `.gitignore` file to your repository, you can even hide the `.gitignore` file from Git. Just add a line `.gitignore` to the file and Git will ignore the `.gitignore` file. However, in such a situation, the file will only reside in the local copy of the repository.

Nowadays, many `.gitignore` templates are available online, depending on the framework you're working on, such as [Rails](#). You may want to browse through this [huge collection](#) of `.gitignore` files on GitHub. These `.gitignore` templates serve as handy starting points for new projects.

Set up Your `.gitignore` Early

Beginners often have a tendency to add a `.gitignore` file at the late stages of a project. However, if a file is already committed and you add it to the `.gitignore` file, it will continue to be committed in your repository and tracked by Git. The only way out in this case is to explicitly untrack the file in Git—after which Git will ignore the file. We'll discuss how to untrack a tracked file in Git in a later chapter.

Remote Repositories

As we've seen so far, you can use Git on your local machine to manage versions of your work. However, because Git is a distributed version control system, many copies of the same repository can exist. So rather than just keep your repository locally, it's common to store another copy in a centralized location on a centralized server (or in the cloud).

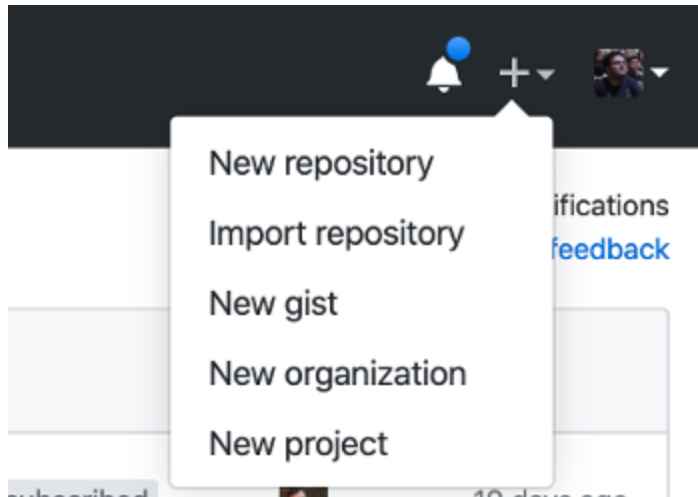
This also enables you to work in a team, as others can access the repository from the centralized copy. Any such copy of your repository can be linked to your repository to enable synchronization. Such an external copy is called a **remote**. A remote is simply a copy of your repository. It can be on a remote server, on a peer's system, or even on a different location within your local system. Interestingly, if you have access to your co-worker's repository (through SSH for instance), even that can be added as a remote.

For demonstration purposes, let's create such a copy on GitHub.

GitHub Isn't the Only Option

GitHub isn't the only option for setting up a remote. A remote may also be on your own server. However, using cloud services like GitHub offers benefits like eliminating the need to run a separate server. You could also create remotes on GitLab or Bitbucket.

To set up a remote repository on GitHub, you first need to create an account on GitHub, or log in to GitHub with your credentials if you already have an account. After login, click on the + arrow on the top right and select **New repository** to create a new repository in the cloud, shown in the figure below.



Choose a name for your repository. You can also choose whether to display your repository publicly or to keep it private.

Once the repository has been created, we have three options: create a new repository from the command line and push to GitHub; push the code from an existing repository from the command line; or import code from another GitHub repository. We'll take the second option here.

GitHub Offers Student Pricing

If you're a student, you can apply for the [GitHub Student Developer Pack](#) to get a free GitHub Pro account, in addition to a lot of other services—which lasts as long as you're a student.

Returning to your local repository, run the following command to synchronize it with the remote repository:

```
git remote add origin
https://github.com/sdaityari/my_git_project.git
git push -u origin master
```

We first add a remote named `origin` to our repository, which points to the GitHub location. Next, the push command sends the commits from your local repository to the cloud repository. The `-u` option links your

repository to the remote for future reference. When you add commits later, Git will show the status of your local copy in relation to this remote repository. `master` is the name of the branch that we want to synchronize with the `origin` remote. We'll discuss branches in detail in the next chapter.

Conclusion

WHAT HAVE YOU LEARNED?

In this chapter, we covered the basics of Git:

- the various ways to install Git on your system
- the three basic operations of track, stage, and commit
- the Git workflow of initialization, tracking, committing and pushing a repository
- starting a Git project from scratch
- the history of a repository
- the use of `.gitignore`
- setting up a remote on GitHub and pushing your code to the cloud

WHAT'S NEXT?

In the next chapter, we'll explore a few more Git commands, focusing on the use of branches in Git.

You've encountered quite a few new things in this chapter, especially if you're new to version control. I think you may want to call it a day. Get a coffee and enjoy a well-deserved break!

Chapter 3: Branching in Git

In Chapter 1, I talked about my one-time fear of trying out new things in a project. What if I tried something ambitious and it broke everything that was working earlier? This problem is solved by the use of branches in Git.

What Are Branches?

Creating a new **branch** in a project essentially means creating a new copy of that project. You can experiment with this copy without affecting the original. So if the experiment fails, you can just abandon it and return to the original—the `master` branch.

But if the experiment is successful, Git makes it easy to incorporate the experimental elements into the `master`. And if, at a later stage, you change your mind, you can easily revert back to the state of the project before this merge.

So a branch in Git is an independent path of development. You can create new commits in a branch while not affecting other branches. This ease of working with branches is one of the best features of Git. (Although other version control options like CVS had this branching option, the experience of merging branches on CVS was a very tedious one. If you've had experience with branches in other version control systems, be assured that working with branches in Git is quite different.)

In Git, you find yourself in the `master` branch by default. The name “`master`” doesn't imply that it's

superior in any way. It's just the convention to call it that.

Branch Conventions

Although you're free to use a different branch as your base branch in Git, people usually expect to find the latest, up-to-date code of a particular project in the master branch.

You might argue that, with the ability to go back to any commit, there's no need for branches. However, imagine a situation where you need to show your work to your superior, while also working on a new, cool feature that's not a part of your completed work. As branching is used to separate different ideas, it makes the code in your repository easy to understand. Further, branching enables you to keep only the important commits in the master branch or the main branch.

Yet another use of branches is that they give you the ability to work on multiple things at the same time, without them interfering with each other. Let's say you submit feature 1 for review, but your supervisor needs some time before reviewing it. Meanwhile, you need to work on feature 2. In this scenario, branches come into play. If you work on your new idea on a separate branch, you can always switch back to your earlier branch to return the repository to its previous state, which doesn't contain any code related to your idea.

Further, imagine your team suddenly discovers a bug in your project and you urgently need to fix it. A new branch would have to be created for this fix and merged with all existing branches once it's fixed. You'll learn more about branch naming conventions in [Chapter 5, "Git Workflows"](#).

Let's now start working with branches in Git. To see the list of branches and the current branch you're working

on, run the following command:

```
git branch
```

Cloning is the process of creating a local copy of a repository from a different source. If you've cloned your repository or set a remote, you can see the remote branches too. Just postfix `-a` to the command above:

```
$ git branch
* master
$ git branch -a
* master
remotes/origin/HEAD -> origin/master
remotes/origin/another_feature
remotes/origin/master
remotes/origin/new_feature
```

As shown above, the branches that start with “remotes” signify that they’re on a remote. In our case, we can see the various branches that are present in the `origin` remote.

Create a Branch

There are various ways of creating a branch in Git. To create a new branch and stay in your current branch, run the following:

```
git branch test_branch
```

Here, `test_branch` is the name of the created branch. However, on running `git branch`, it seems that the active branch is still the `master` branch. To change the active branch, we can run the `checkout` command:

```
$ git checkout test_branch
$ git branch
git branch
  master
* test_branch
```

What Does checkout Do?

checkout is used for multiple purposes in Git. You'll come across many such examples over the course of this book. In this example, checkout enables you to change the current branch of the repository, essentially "checking out" to a new branch.

You can also combine the two commands above and thereby create and checkout to a new branch in a single command by postfixing `-b` to the checkout command:

```
$ git checkout -b new_test_branch
Switched to a new branch 'new_test_branch'
$ git branch
  master
* new_test_branch
  test_branch
```

The branches we've just created are based on the latest commit of the current active branch—which in our case is `master`. If you notice an unwanted change or error in the latest commit and would like to explore an earlier version of the repository, you can create a branch from an older commit. To create a branch (say `old_commit_branch`) based on an older commit—such as `cafb55d`—you can run the following command:

```
$ git checkout -b old_commit_branch cafb55d
Switched to a new branch 'old_commit_branch'
$ git log --oneline
cafb55d (HEAD -> old_commit_branch) Merge commit
'5ef655a4caf8'
cc48fb3 Added lines 1 and 3 using add -p
5ef655a Fixed conflict from another_feature branch
96f7c5e Another change in the master branch
7534bc2 Some change in the master branch
49ed357 Added another feature
7e0eea2 (origin/new_feature) Removed line
f87d1a5 Dummy change
f934591 - Changed two files - This looks like a
cooler interface to write commit messages
8dd76fc My first commit
```

The `--oneline` option shows a compact form of the Git history, with one line for each commit.

To rename the current branch to `renamed_branch`, run the following command:

```
git branch -m renamed_branch
```

Delete a Branch

To delete a branch, run the following command:

```
git branch -D new_test_branch
```

Don't Delete Branches Unless You Have To

As there's not really any downside to keeping branches, as a precaution I'd suggest not deleting them unless the number of branches in the repository becomes too large to be manageable.

The `-D` option used above deletes a branch even if it hasn't been synchronized with a remote branch. This means that, if you have commits in your current branch that haven't been pushed yet, `-D` will still delete your branch without providing any warning. To ensure you don't lose data, you can postfix `-d` as an alternative to `-D`. `-d` only deletes a branch if it's been synchronized with a remote branch. Since our branches haven't been synced yet, this is what happens if we postfix `-d`:

```
$ git branch -d new_test_branch
The branch 'new_test_branch' is not fully merged.
If you are sure you want to delete it, run 'git
branch -D test_branch'.
```

As you can see, Git gives you a warning and aborts the operation, as the data hasn't been merged with a branch yet.

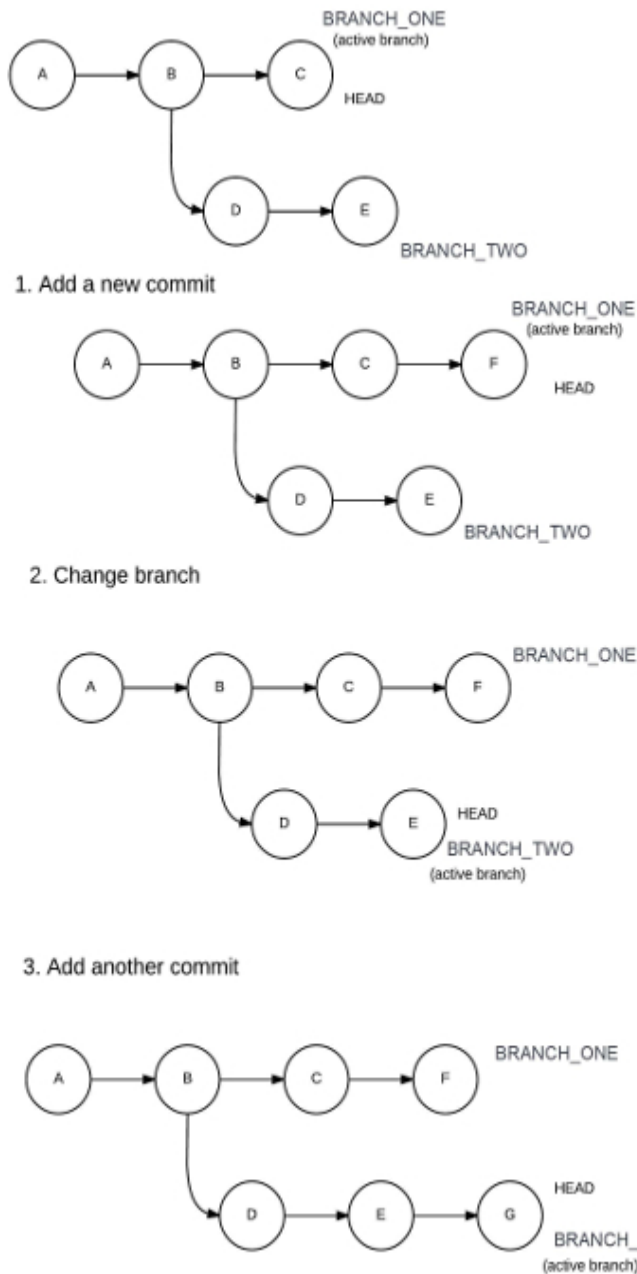
BRANCHES AND HEAD

Now that we've had a chance to experiment with the basics of branching, let's spend a little time discussing

how branches work in Git, and also introduce an important concept: HEAD.

As mentioned above, a branch is just a link between different commits, or a pathway through the commits. The **HEAD** of a branch points to the latest commit in the branch. In other words, it refers to the tip of a branch. We'll refer to HEAD a lot in upcoming chapters.

A branch is essentially a pointer to a commit, which has a parent commit, a grandparent commit, and so on. This chain of commits forms the pathway I mentioned above. How, then, do you link a branch and HEAD? Well, HEAD and the tip of the current branch point to the same commit. The following diagram illustrates this idea.



As shown in Figure 3-1, BRANCH_ONE initially is the active branch and HEAD points to commit C. Commit A is the base commit and doesn't have any parent commit, so the commits in BRANCH_ONE in reverse chronological order (which also forms the pathway I've talked about) are $C \rightarrow B \rightarrow A$. The commits in BRANCH_TWO are $E \rightarrow D \rightarrow B \rightarrow A$. The HEAD points to the latest commit of the

active `BRANCH_ONE`, which is commit C. When we add a commit, it's added to the active branch. After the commit, `BRANCH_ONE` points to F, and the branch follows $F \rightarrow C \rightarrow B \rightarrow A$, whereas `BRANCH_TWO` remains the same. `HEAD` now points to commit F. Similarly, the changes when we add yet another commit are demonstrated in the figure.

Advanced Branching: Merging Branches

As mentioned earlier, one of Git's biggest advantages is that, compared to Subversion, merging branches is especially easy. For instance, it's difficult to store linkages between branches and the master branch (called trunk) in Subversion. Working on a branch for a long time makes it really difficult to go back and merge with the trunk, as it requires the developer to figure out where to merge. All of these issues are fixed in Git.

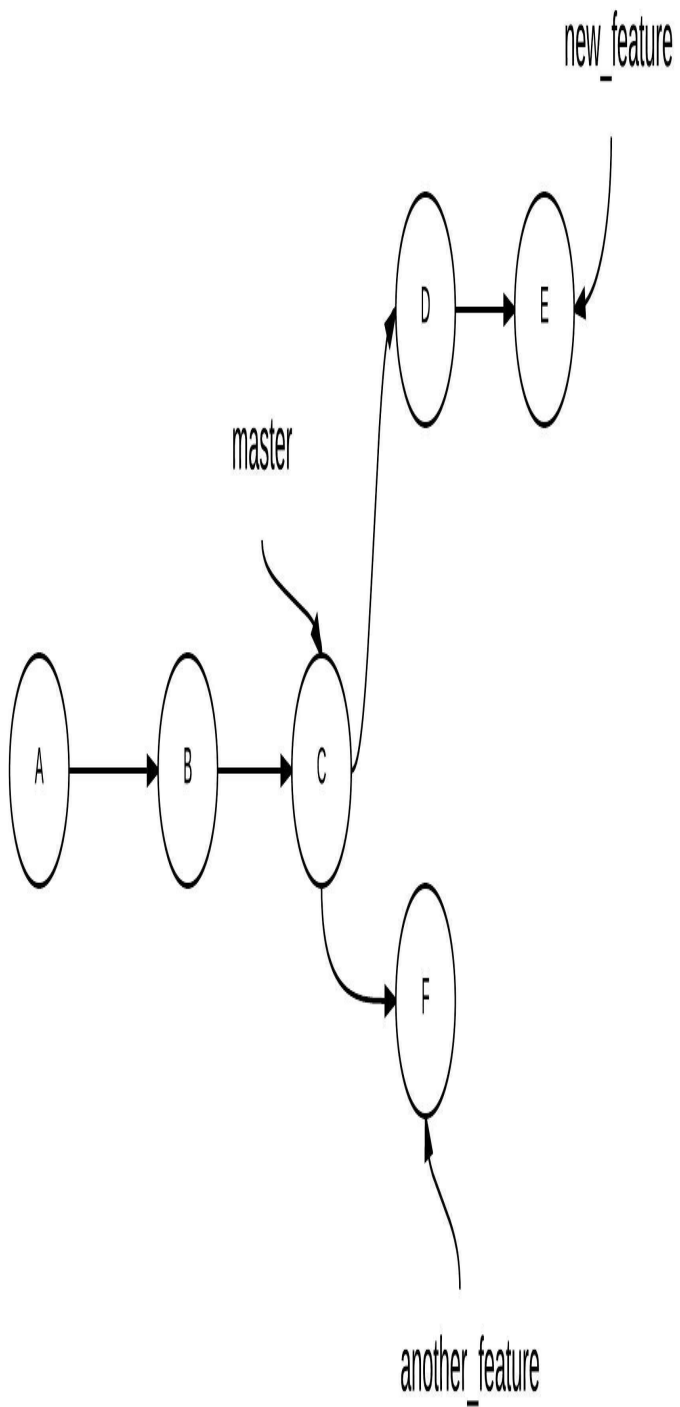
Let's now look at how branching works in Git.

We'll create two new branches—`new_feature` and `another_feature`—and add a few dummy commits. Checking the history in each branch shows us that the branch `another_feature` is ahead by one commit, as shown below:

```
$ git checkout another_feature
Switched to a new branch 'another_feature'
$ git log --oneline
49ed357 Added another feature
7e0eea2 Removed line
f87d1a5 Dummy change
f934591 - Changed two files - This looks like a
cooler interface to write commit messages
8dd76fc My first commit
$ git checkout new_feature
Switched to a new branch 'new_feature'
$ git log --oneline
7e0eea2 Removed line
f87d1a5 Dummy change
f934591 - Changed two files - This looks like a
```

```
cooler interface to write commit messages
8dd76fc My first commit
```

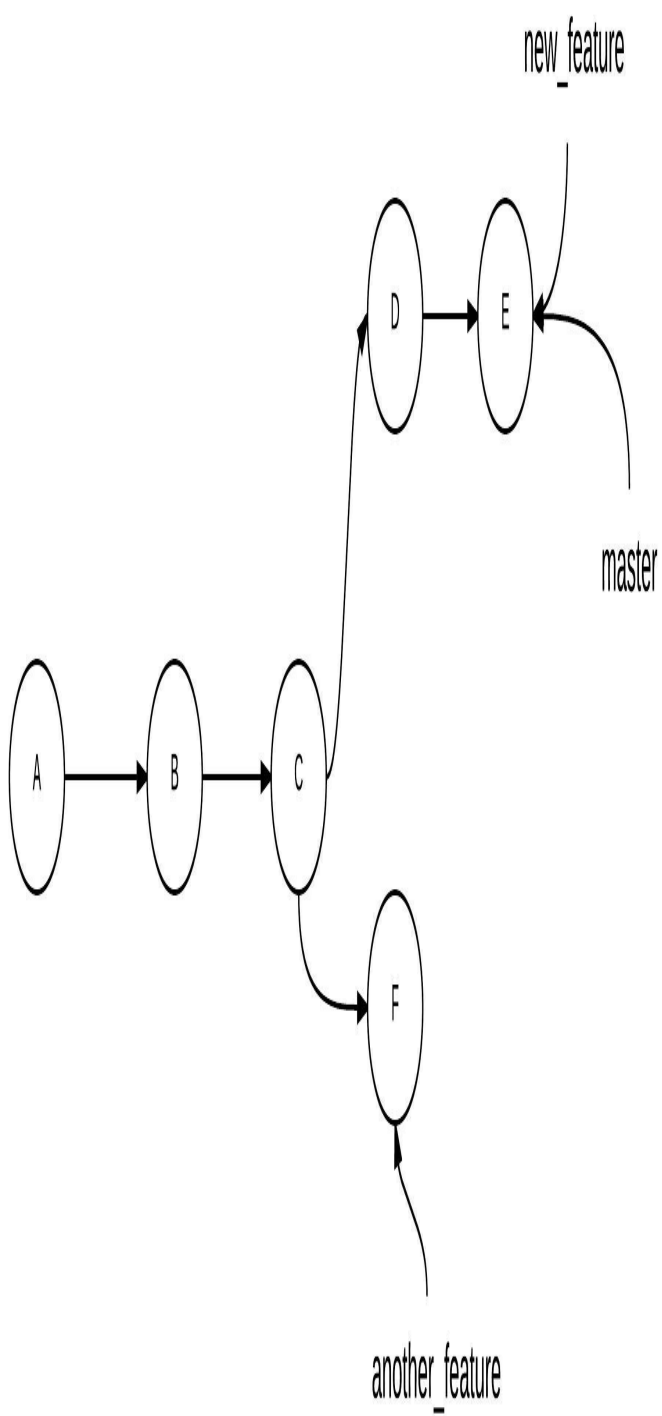
This situation is illustrated in Figure 3-2. Each circle represents a commit, and the branch name points to its HEAD (the tip of the branch).



To merge `new_feature` with `master`, run the following (after first making sure the `master` branch is active):

```
git checkout master  
git merge new_feature
```

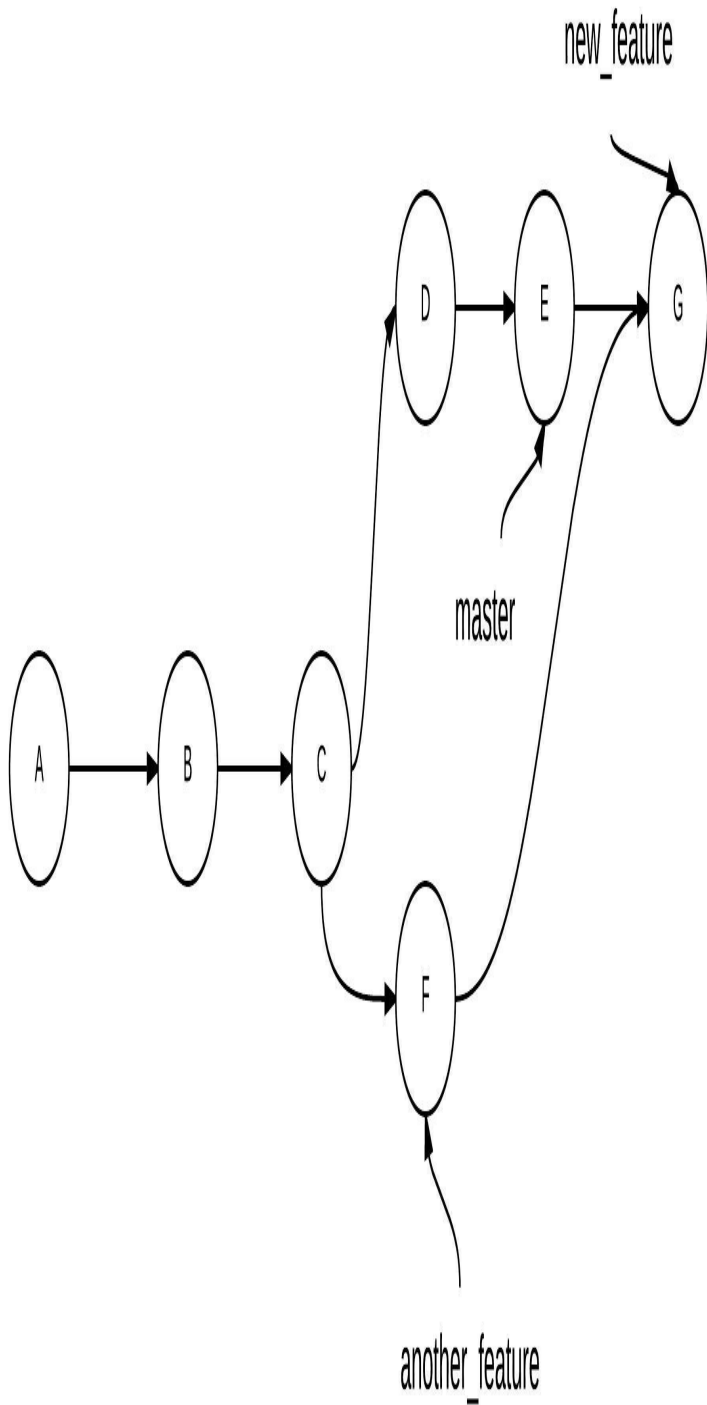
The result is illustrated in Figure 3-3.



To merge `another_feature` with `new_feature`, just run the following (making sure that the branch `new_feature` is active):

```
git checkout new_feature  
git merge another_feature
```

The result is illustrated in Figure 3-4.



Watch Out for Loops

The diagram above shows that this merge has created a loop in your project history across the two commits, where the workflows diverged and converged, respectively. While working individually or in small teams, such loops might not be an issue. However, in a larger team—where there might have been a lot of commits since the time you diverged from the main branch—such large loops make it difficult to navigate the history and understand the changes. We’ll explore a way of merging branches without creating loops using the rebase command in Chapter 7, “Unlocking Git’s Full Potential”.

This merge happened without any “conflicts”. The simple reason for that is that no new commits had been added to branch `new_feature` as compared to the branch `another_feature`. **Conflicts** in Git happen when the same file has been modified in non-common commits in both branches. Git raises a conflict to make sure you don’t lose any data.

We’ll discuss conflicts in detail in the next chapter. I mentioned earlier that branches can be visualized by just a simple pathway through commits. When we merge branches and there are no conflicts, such as above, only the branch pathway is changed and the HEAD of the branch is updated. This is called the **fast-forward** type of merge.

The alternate way of merging branches is the **no-fast-forward** merge, by postfixing `--no-ff` to the merge command. In this way, a new commit is created on the base branch with the changes from the other branch. You’re also asked to specify a commit message:

```
git checkout master
git merge --no-ff new_feature
```

In the example above, the former (merging `new_feature` with `master`) was a fast-forward merge, whereas the latter was a no fast-forward merge with a merge commit.

While the fast-forward style of merges is default, it's generally a good idea to go for the no-fast-forward method for merges into the master branch. In the long run, a new commit that identifies a new feature merge might be beneficial, as it logically separates the part of the code that's responsible for the new feature into a commit.

Conclusion

WHAT HAVE YOU LEARNED?

In this chapter, we discussed the following characteristics about branches in Git:

- what branches are in Git
- how to create new branches from existing branches
- the process of merging branches, and how Git's history is affected

WHAT'S NEXT?

I've already spoken about how Git is beneficial to developers working in teams. The next chapter will look at this in more detail, as well as specific Git actions and commands that are frequently used while working in a distributed team.

So far, we've looked at managing source code by starting a Git project, working with branches, and pushing code to a remote repository. In the following chapter, we'll focus on the features of Git that help you contribute in a team.

We've seen how useful Git's version control tools can be for a sole coder. Git's power is even more evident when it

comes to managing a project with many contributors. It enables members of a team to work independently on a project and stay in sync—even when they're located far apart from each other.

Chapter 4: Using Git in a Team

Earlier, we performed a push operation to GitHub, sending a copy of our local repository to the cloud. This is the process you follow when the repository has been created on your local system.

However, if you're working on a team, it's possible that some work has already been done on the repository when you join. In this scenario, you need to grab a copy of the code from a central repository and work on it. The process of grabbing this repository is called "cloning". **Cloning** is the process of creating a copy of a remote repository. The copy (or clone) that you create has its own project history, and any work done on it is independent of the development on the remote.

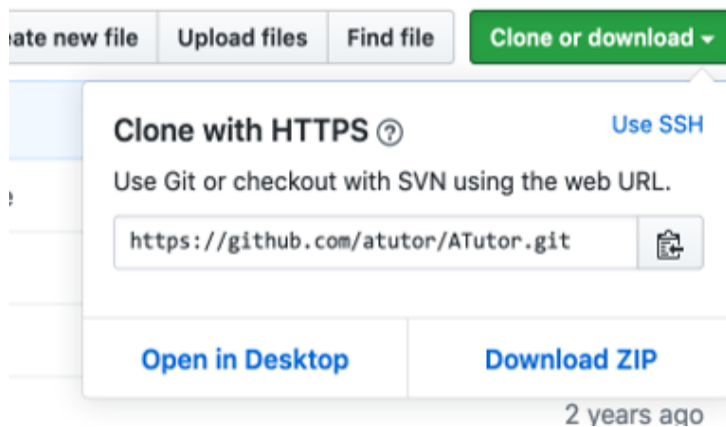
The Source Is the origin

If you clone a repository, the source you clone it from is designated as the `origin` remote by default.

Think of cloning as creating photocopies of a document. If you overwrite something in the photocopy, the original document remains untouched. Similarly, if you change the original document after making the photocopy, the photocopy retains the contents of the original document. Until you merge the clone with the original remote, they are separate entities.

To clone a remote repository, you need to know its location. This location usually takes the form of a URL. In GitHub, you can find the URL of a project on the bottom-right corner of the home page of that project.

Let's look at an example of a repository on my own GitHub account, as shown in Figure 4-1.



To clone this project, we need to run the following command:

```
git clone
https://github.com/sdaityari/my_git_project.git
```

When the repository is successfully cloned, a local directory is created with the same name as the project name (in our case, `my_git_project`), and all the files under the repository are present in that directory. It's not necessary to keep the directory name; you can change it any time. If you want to change the root directory name of the repository while cloning it—let's say to `my_project`—you'll need to provide the name to the clone command:

```
git clone
https://github.com/sdaityari/my_git_project.git
my_project
```

You may also rename the directory after you've cloned the repository.

If you wish, you can verify that the `origin` remote points to the URL you just cloned:

```
$ git remote -v
origin
https://github.com/sdaityari/my_git_project.git
(fetch)
origin
https://github.com/sdaityari/my_git_project.git
(push)
```

The `-v` option is short for `--verbose`, and tells Git to display the URLs of the remotes next to the names.

OPTIONAL: DIFFERENT PROTOCOLS WHILE CLONING

In the command we used to clone the repository, you may have noticed that the URL starts with `https`. However, a situation may arise where this protocol won't be useful. Perhaps your organization doesn't use cloud services to host its projects, or a certain protocol may be restricted by the firewall.

In these situations, it makes sense to know about Git's other protocols. The available options for any Git remote are as follows:

- local protocol
- Git protocol
- HTTP/HTTPS protocol
- SSH protocol

The local protocol involves cloning in the same local network. This protocol is helpful when all team members have access to a shared file system. For instance, if you're working on a sensitive project that should remain in the confines of a single server, you could have team members remotely log in to the common system and continue their development.

You can clone a repository like so:

```
git clone /Users/donny/my_git_project
```

The biggest disadvantage is the access this protocol provides, which is limited to the local computer.

If you clone over the Git protocol, your URL starts with `git` instead of `https`:
`git://github.com/sdaityari/my_git_project.git`. This doesn't provide any security. You only get read-only access over the `git` protocol, and therefore you can't push changes.

With the `https` protocol, your connection is encrypted. GitHub allows you to clone or pull code anonymously over `https` if the repository is public. However, for pushing any code, your username and password are verified first. GitHub recommends using `https` over `ssh`, because the `https` option always works, even if you're behind a firewall or a proxy.

If you're using the `https` protocol, you need to type in your credentials every time you push code. However, if you push your code frequently, you can make Git remember your credentials for a given amount of time after you successfully enter them once. This is done with the `credential.helper` setting. Run the following to enable credential storage:

```
git config --global credential.helper cache
```

By default, Git stores your credentials for 15 minutes. You may also set the timeout limit in seconds:

```
git config --global credential.helper "cache --  
timeout=3600"
```

This command makes Git store your credentials for an hour.

[Alternative Credential Storage](#)

An alternative but less secure way of saving the username and password indefinitely would be to store them within the remote path itself. In such a case, your remote would look like this:
`https://sdaityari:password@github.com/sdaityari/my_git_project.git.`

The `ssh` protocol, on the other hand, authenticates your requests using public key authentication. You establish a connection with the remote server over `ssh` first, and then you request the resource. To set up authentication using `ssh`, you need to generate your public/private key pair.

In Linux or macOS, the following command generates a key pair:

```
ssh-keygen -t rsa -C "sdaityari@gmail.com"
```

In Windows, you need either PuTTY or Git Bash to generate the key. GitHub provides [detailed instructions on the process of generating the key pair on Windows](#).

Git GUI Tools Can Generate Keys for You

If you use a Git GUI desktop client, the process of generating a key pair and linking it with your GitHub account is done automatically by the client. We'll review clients in [Chapter 9](#).

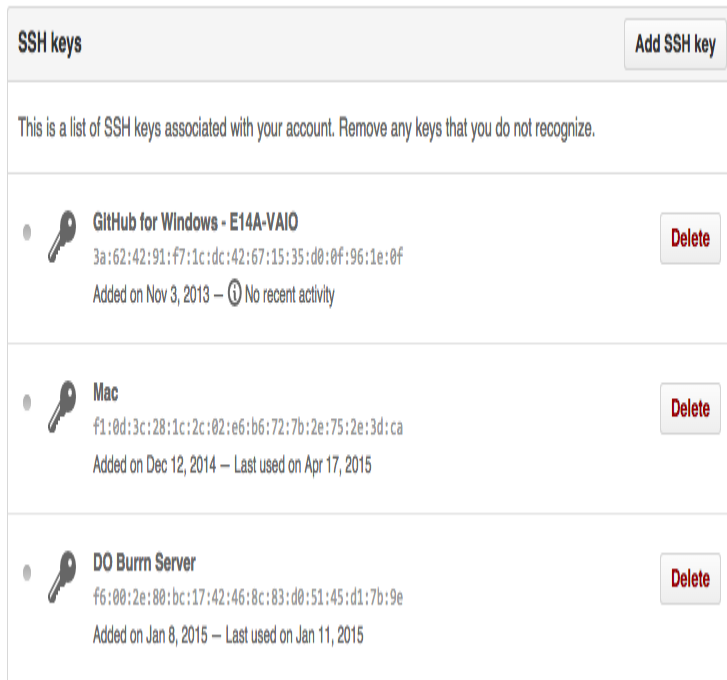
Your public key is stored in the file `~/.ssh/id_rsa.pub`. You can view it using the `cat` command:

```
cat ~/.ssh/id_rsa.pub
```

The `cat` command prints the contents of a file in the terminal. `~` stands for the home directory of the current active user. For instance, if your username is `donny`, `~` points to `/Users/donny/` on macOS and `/home/donny` on Linux.

You need to add the contents of the public key to your [GitHub SSH settings](#) in order to establish ssh connections to GitHub, as shown in Figure 4-2.

Need help? Check out our guide to [generating SSH keys](#) or troubleshoot [common SSH Problems](#)



The screenshot shows the 'SSH keys' management page in GitHub. At the top right is an 'Add SSH key' button. Below the header, a message states: 'This is a list of SSH keys associated with your account. Remove any keys that you do not recognize.' The list contains three entries, each with a key icon, a title, a hexadecimal public key, and a 'Delete' button. The first entry is 'GitHub for Windows - E14A-VAIO' with key '3a:62:42:91:f7:1c:dc:42:67:15:35:d0:0f:96:1e:0f' and notes 'Added on Nov 3, 2013' and 'No recent activity'. The second is 'Mac' with key 'f1:0d:3c:28:1c:2c:02:e6:b6:72:7b:2e:75:2e:3d:ca' and notes 'Added on Dec 12, 2014' and 'Last used on Apr 17, 2015'. The third is 'DO Burrn Server' with key 'f6:00:2e:80:bc:17:42:46:8c:83:d0:51:45:d1:7b:9e' and notes 'Added on Jan 8, 2015' and 'Last used on Jan 11, 2015'.

Contributing to the Remote: Git Push Revisited

Earlier in this book, we created a repository in the cloud and pushed our local code to it. Once you've made changes to a repository, they need to be pushed to the remote if the central repository is to reflect them. `git push` is a simple command that does the trick:

```
git push
```

We'll now explore push a little further. There are various ways to push code to a remote.

A `git push` simply pushes the code in the current branch to the `origin` remote branch of the same name.

A branch is created if the branch with the same name as the current local branch doesn't exist on the `origin`.

```
git push remote_name
```

This command pushes the code in the current branch to the `remote_name` remote branch. A branch is created on the remote if the branch with the same name as the current local branch doesn't exist on the `remote_name` remote.

```
git push remote_name branch_name
```

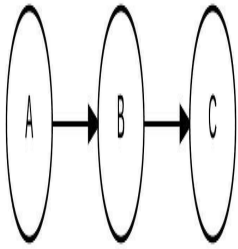
This command pushes the code on the `branch_name` branch (irrespective of your current branch) to the remote branch of the same name. If `branch_name` doesn't exist on the remote, it's created. If `branch_name` doesn't exist on the local repository, an error is shown.

```
git push remote_name local_branch:remote_branch
```

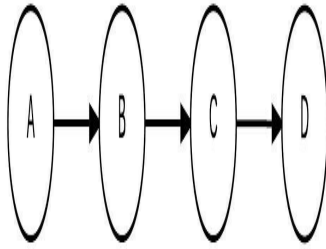
This command pushes the `local_branch` from the local repository to the `remote_branch` of the remote repository. Although it involves typing a longer command, I would always advise that you use this syntax for pushing your code, as it avoids mistakes.

Figure 4-3 gives a rough idea of how the states of the `master` and `origin/master` look before and after a push operation.

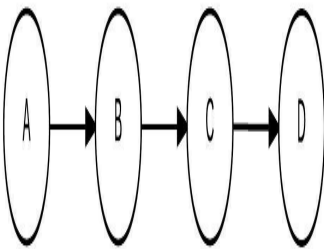
origin/master



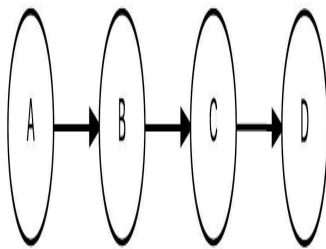
origin/master



master



master



Before Push

After Push to origin/master

You Can Delete Branches Using `git push`

You can modify the syntax listed above to delete a branch on the remote:

```
git push remote_name :remote_branch
```

In this command, you're essentially sending an empty branch to the `remote_branch` branch of `remote_name`, which empties the `remote_branch`, or in other words deletes it on the remote. You should therefore be careful while attempting this operation.

Keeping Yourself Updated with the Remote: Git Pull

Now that we've looked at how to push the changes to the remote, let's explore the situation where others are working on the same project and you need to update your local repository with the changes other contributors have made.

The ideal way to update your local repository with the commits others have made to the remote is, firstly, by downloading the new data, and then by merging it with the appropriate branches.

To download the changes that have appeared in the remote, we run the following command:

```
git fetch remote_name
```

This updates our local branches from the remote `remote_name`. (We can skip the name of the remote by running just `git fetch`, and the command will update the branches of the local repository from the remote `origin`.)

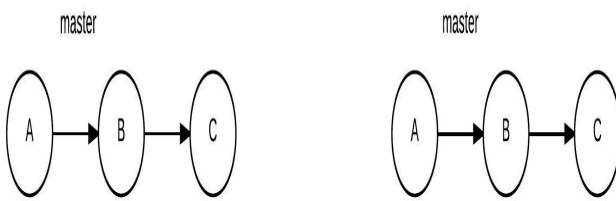
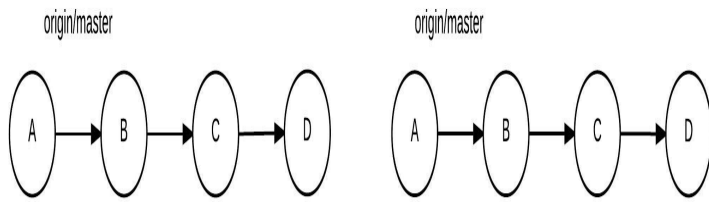
When you clone a repository, local versions of its branches are also maintained. The `fetch` command updates these local versions with the latest commits from the remote.

Following a `fetch`, to update your local branch you need to merge it with the appropriate branch from the remote. For instance, if you're planning to update the local master branch with the remote's master branch, run the following command:

```
git merge origin/master
```

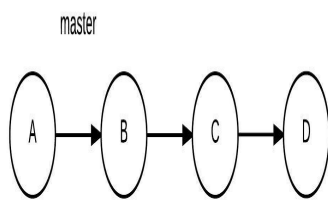
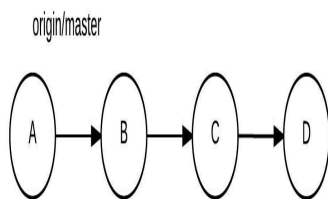
This is basically merging the branch `origin/master` with your current active branch. Following the `fetch`, your `origin/master` is updated with the latest commits of the branch on the remote. You've therefore succeeded in updating a local branch with the latest commits from a remote branch.

To understand what's going on, let's explore further with the help of a diagram.



Before fetch

After fetch



After merge with origin/master

Alternatively, a shorter way of updating the local branch by downloading and merging a remote branch is by using `pull`. The `git pull` command is essentially a `git fetch` followed by a `git merge`. To update the current active branch through `pull`, run the following:

```
git pull origin master
```

Pulls Are Fast-forward by Default

Just as with merging, you can specify whether or not a pull should be a fast-forward. It is by default, but this can be overridden with the `--no-ff` postfix.

As with `git push`, it's possible to specify different local and remote branches for `git pull` too:

```
git pull
```

A `git pull` simply downloads the code from the `master` branch of the `origin` remote branch. It then merges the code with the current active branch.

```
git pull remote_name
```

The command above first downloads the code from the `master` branch of the `remote_name` remote branch. It then merges the code with the current active branch.

```
git pull remote_name branch_name
```

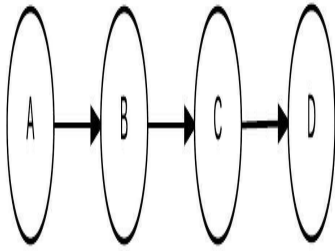
The command above first downloads the code from the `branch_name` branch of the `remote_name` remote branch. It then merges the code with the current active branch.

```
git pull remote_name local_branch:remote_branch
```

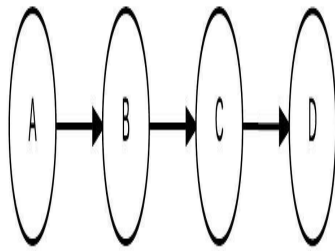
This command first downloads the code from the `remote_branch` branch of the `remote_name` remote branch. It then merges the code with the `local_branch` in the local repository.

To help visualize the process of a `git pull`, the following diagram shows the status of the local repository before and after a pull.

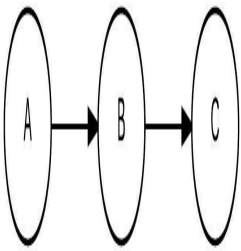
origin/master



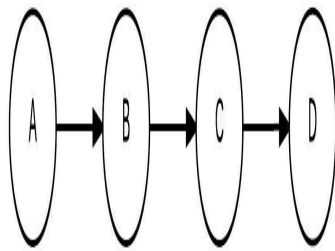
origin/master



master



master



Before pull

After pull

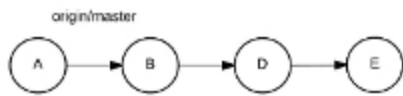
Here Be Conflicts!

A `fetch-merge` or `pull` may result in a conflict. Git raises a conflict when a change is made to similar lines in the same file in both branches that you're trying to merge. In such a case, you'll need to resolve the conflicts before completing the `merge` or `pull`. We'll discuss conflicts later in this chapter.

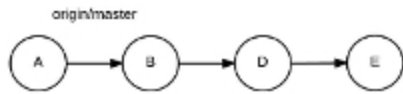
Dealing with a Rejected Git Push

Now that you have the knowledge of both sending and receiving updates in your local repository, let's look at a special situation. It involves pushing new code to a remote branch that's been updated since your last synchronization. In this case, your push would be rejected—with the message that it's “non-fast-forward”. This simply means that, since changes were made to both the remote and your local copy, Git is not able to determine how to merge them.

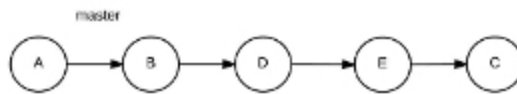
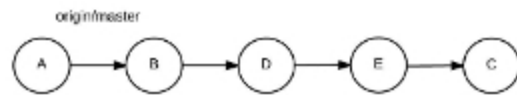
In such a situation, you last synced the `master` branch from `origin` (hence referred to as `origin/master`) when it was at commit B (as named in the diagram below). You've proceeded with two commits, D and E. Since your last sync, a new commit C has been added to `origin/master`. Git doesn't merge both these workflows, as they've taken different pathways. Therefore, you should first pull from `origin/master` and merge it with `master`, resolving any conflicts that appear. This would make commit C appear in your `master` branch. Git will then be able to accept the push.



Rejected Push Situation



Step 1: After Pull



Step 2: After Push

Rebase?

In this example, we demonstrate a `pull --rebase` in Figure 4-6 rather than just a `pull`. For now, just ignore this, as I'll explain `rebase` in [Chapter 7](#), "Unlocking Git's Full Potential".

Conflicts

Let's now address conflicts—the topic perhaps most dreaded by people working with Git.

Conflicts can occur when you're trying to merge two branches or to perform a pull. However, as a pull operation essentially involves merging, we'll address conflicts only during a merge. If you encounter a conflict during a pull, the process of resolving it remains the same.

A **conflict** arises when your current branch and the branch to be merged have diverged, and there are commits in your current branch that aren't present in the other branch, and vice versa. Git isn't able to determine which changes to keep, so it raises a conflict to ask the user to review the changes. The last common commit between the two branches—which is also the point where they diverged—is called the **base commit**.

When Git merges the two branches, it looks at the changes in each branch since the base commit. When there are unambiguous differences—like changes to different files, and sometimes different parts of the same file—the changes are applied. However, if there are changes to the same parts of the same file, and Git can't determine which changes to keep, it raises a conflict.

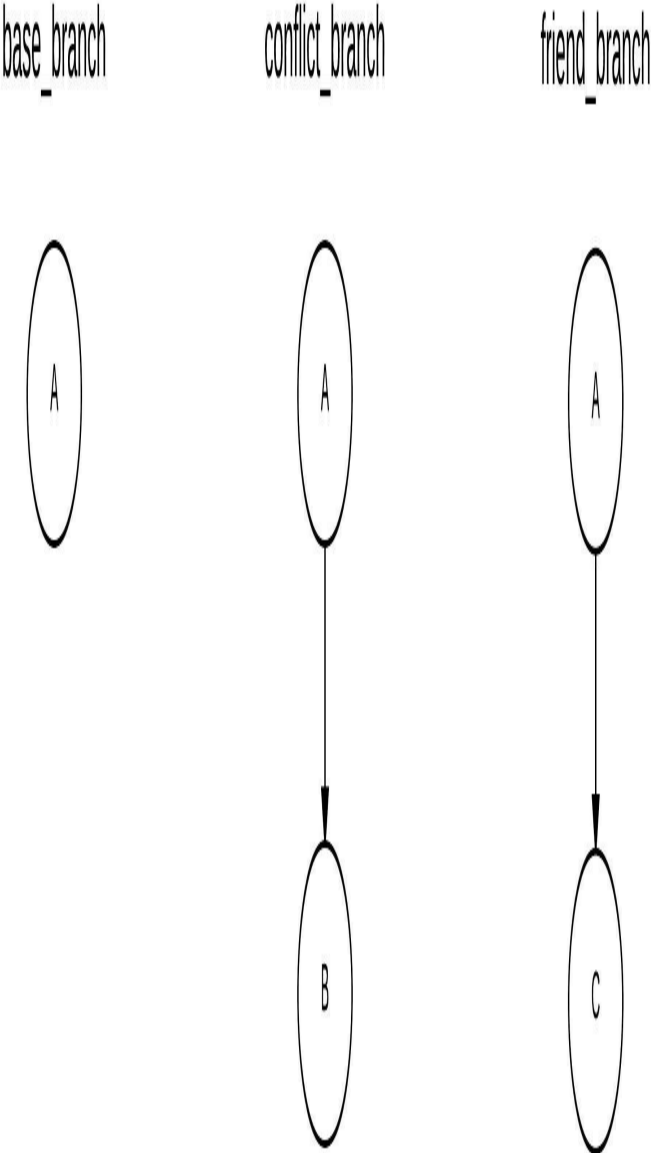
To understand conflicts properly, let's try to create an example conflict ourselves. We'll create a reference branch named `base_branch`. Let's also create a sample program in Python—`sample.py`—the contents of which are shown below:

```
CONSTANT = 5

def add_constant(number):
    return CONSTANT + number
```

It's a simple program that adds a constant to a provided number. Now imagine a scenario where you make a branch, `conflict_branch`, where you change the value of `CONSTANT` to 7. And suppose a friend has worked on the same line numbers of the same file on the branch

friend_branch, and changed the CONSTANT to 9. We can visualize this with Figure 4-7.



Now, let's see what happens when we try to merge the `friend_branch` with our `conflict_branch`:

```
$ git merge friend_branch
Auto-merging sample.py
CONFLICT (content): Merge conflict in sample.py
Automatic merge failed; fix conflicts and then
commit the result.
```

Git shows a message that the automatic merge failed, and that there are conflicts in `sample.py` that need to be resolved.

That doesn't sound so great! Let's do a `git status` to see what's wrong:

```
$ git status
On branch conflict_branch

You have unmerged paths.
  (fix conflicts and run "git commit")
  (use "git merge --abort" to abort the merge)

Unmerged paths:
  (use "git add <file>..." to mark resolution)
    both modified:   sample.py

no changes added to commit (use "git add" and/or
"git commit -a")
```

Git shows that both files have been modified, and that we need to make a commit after fixing the conflicts.

Naturally, this isn't a fast-forward commit, as Git has failed to automatically resolve the merge. A new commit will be created once you fix the conflicts and commit your changes.

Note that a conflict arises only when Git is unable to determine which lines to keep. To make sure no data is lost, you're asked which lines should be kept.

Look at the contents of the file now. Since you initiated the merge, Git has modified the file to show you the changes in the two versions of the same file:

```
<<<<<< HEAD
CONSTANT = 7
=====
CONSTANT = 9
>>>>>> friend_branch

def add_constant(number):
    return CONSTANT + number
```

The lines between <<<<<< HEAD and ===== contain your version of the part of the file, whereas the lines between ===== and >>>>>> friend_branch contain the part of the file that's present in the friend_branch. You should review these lines and decide which lines to keep. You may need to take up the issue with your team before you decide which version to keep. In our case, let's keep the change we made in the branch friend_branch. In this case, the conflict is solved by keeping one set of changes. However, in a real-life situation, you may need to combine the two sets of changes as well.

Here are the contents of the edited file before we commit changes:

```
CONSTANT = 9

def add_constant(number):
    return CONSTANT + number
```

Multiple Conflicts

In our simple example, there was just one conflict in a single file. If there are conflicts in multiple files, they'll appear when you run `git status`. You need to edit them individually to check which version to keep. If there are multiple conflicts in the same file, you should search for the word HEAD or <<<<< (multiple "less than" signs together are rarely used in your source code) to find out the instances within a file where conflicts have arisen, and then work on them individually.

After you've resolved the conflicts, you should stage the changed files for commit. In our case, there's only a single file:

```
git add sample.py
```

You should then proceed to making a commit, as shown in the line of code below:

```
git commit -m "Concluded merge with friend_branch"
```

Aborting a Merge with Conflicts

After initiating a merge that's resulted in conflicts, if you're overwhelmed and want to go back to the pre-merge state, you can do so by aborting the merge:

```
$ git merge --abort
$ git status
On branch master

nothing to commit, working tree clean
```

Conclusion

With this, we come to the end of another fairly lengthy chapter. Let's briefly review the things we've covered.

WHAT HAVE YOU LEARNED?

In this chapter, we covered how to:

- clone from a remote repository
- create, update, merge and delete branches
- keep a local repository updated
- send the changes from a local repository to a remote
- manage conflicts during merges

We also looked at general workflows while working with organizations.

WHAT'S NEXT?

In the next chapter, we'll discuss Git workflows—a set of guidelines to follow when using Git in an organization.

Chapter 5: Git Workflows

So far, we've covered the basics of Git and how to use them as part of a team. But teams differ in the way they utilize Git in their projects. A Git **workflow** is a set of guidelines that a team should follow to manage a project.

In this chapter, we'll explore the most commonly used Git workflows. A workflow generally provides guidelines on the following items:

- the architecture of the project
- how contributions are made to the project
- how the work of others is merged into the project

Git's flexibility allows you to set up diverse guidelines for your project. This can potentially lead to a large number of workflows. How do you ensure that team members follow these guidelines? It may be a good idea to follow a specific, well-defined workflow.

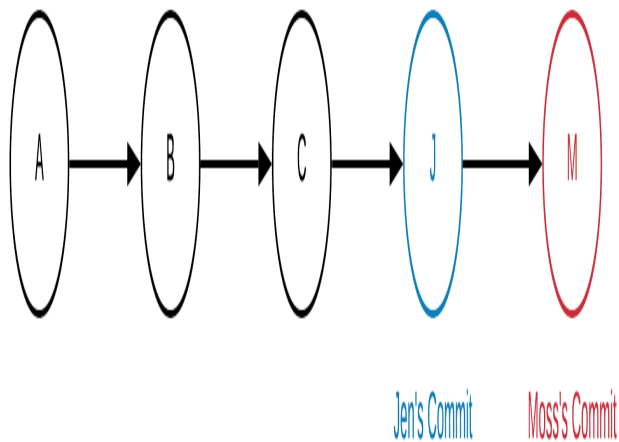
As workflows are described in this chapter, remember that they represent broad guidelines for using Git in your project, and that developers often make minor changes to these guidelines for their convenience. When you're assessing a workflow, make sure you ask the following questions:

- How difficult is it for a new team member to get started?
- How much effort goes into reverting the status of the repository after an unwanted change?
- Does the workflow scale up well to your team size growth projections?

As each workflow is described, the discussion will be structured around these items.

The Centralized Workflow

Centralized Workflow



FEATURES

While Git is a distributed version control system, it's still possible to implement a centralized workflow, inspired by centralized version control systems like Subversion. A **centralized** workflow is the simplest of Git workflows, in which just a single branch (typically the `master` branch) is used for all operations. It's called a "centralized" workflow because a single copy of the repository is treated as the main copy, into which every developer syncs their changes. The centralized workflow

is also called the “trunk” workflow, as subversion’s master branch is called trunk. For this workflow to work seamlessly, you need to give every team member access to your master branch. The central repository can be on a local server at a location every developer can access, or it can be hosted on a central platform like GitHub or Bitbucket.

NEW TEAM MEMBER ORIENTATION

If you’re a new team member, you start by cloning the central repository. All you need to do is make changes to your master branch and push it to the central repository. If someone has updated the master since you last updated your local branch, you’re prompted to merge the changes first and then push them.

PROS AND CONS

The biggest advantage of this workflow is its simplicity. The centralized workflow doesn’t use the branching feature of Git. Beginners often find the branching feature of Git to be the most difficult to understand, so the simplicity of this workflow works best for beginners. Developers who are familiar with Subversion and new to Git also find this workflow very intuitive. It’s also ideal for smaller teams that require minimal code review before a merge. If you’re managing a personal project, the centralized workflow is an intuitive choice as well.

On the other hand, the centralized workflow gets tedious with an increase in team size. Managing changes in code is a challenge if you have multiple people working on the same branch at the same time.

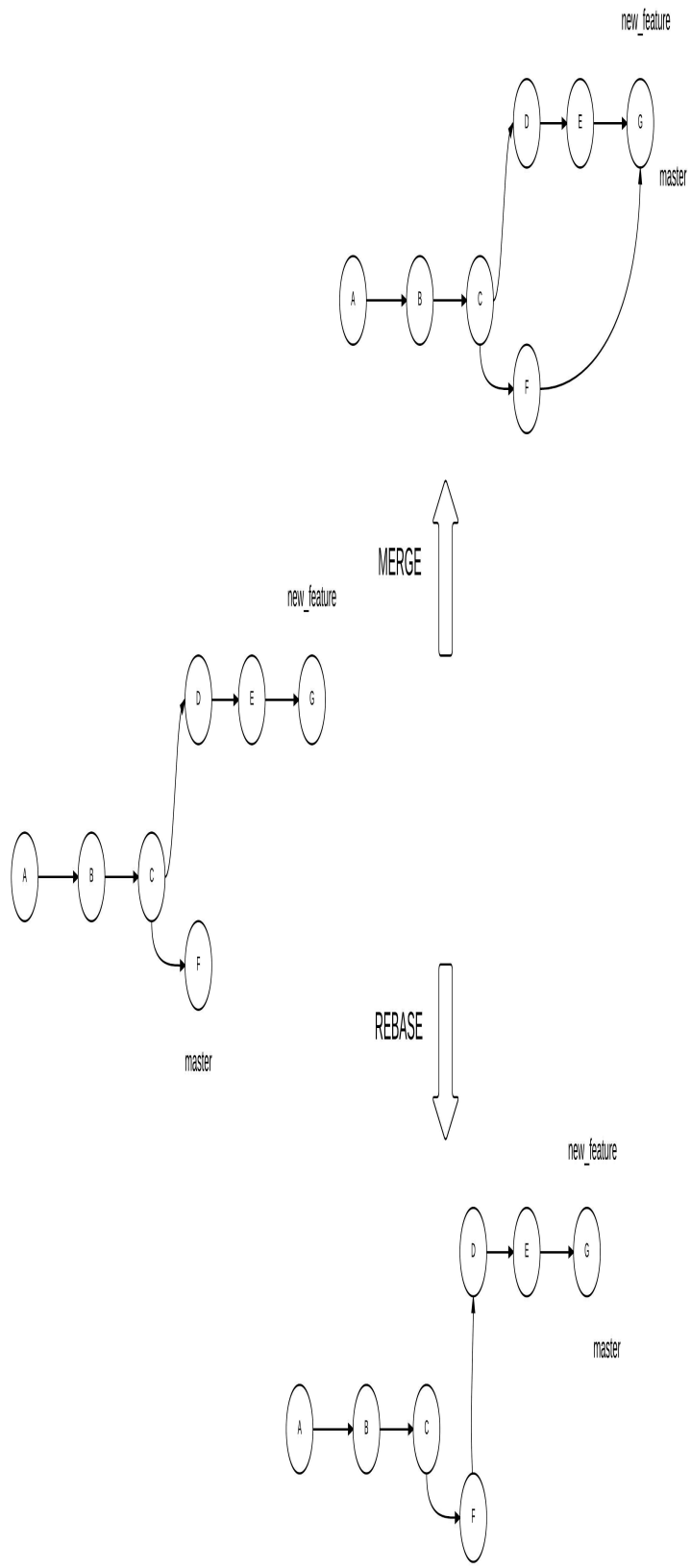
Finally, giving all team members access to the master branch may not be a good idea in a large team. A single error, if introduced in the codebase, can corrupt the whole repository. No one can really make changes to the

codebase until it's fixed. Therefore, a more robust workflow is needed as your team grows in size.

WHO SHOULD USE THE CENTRALIZED WORKFLOW

The simplicity of the centralized workflow makes it perfect for two types of users. If you're new to Git and your team is exploring the use of version control in your projects, you should start with the centralized workflow. Secondly, if you use Git to manage a personal project, the centralized workflow is ideal. For instance, if you're a student who manages academic assignments, or an author managing your texts, the centralized workflow fits into your needs perfectly.

The Feature-branch Workflow



FEATURES

Because the centralized workflow doesn't utilize the branching features of Git, the next logical step up from that is to introduce branches for specific changes in your codebase. This results in what's known as the "feature-branch" workflow. The **feature-branch workflow** follows the concept of feature development in separate branches, without affecting the `master` branch. You can either `merge` or `rebase` the feature branch into the `master`, as shown above (we'll cover `rebase` in detail later in the section "Rebase" in [Chapter 7](#)).

NEW TEAM MEMBER ORIENTATION

You must maintain a repository at a central location in the feature-branch workflow, with read access to the `master` branch to all developers. A new developer must first clone the `master` branch and create a new local branch for every feature they start. While the definition of a feature differs from project to project, it's a good idea to logically separate each "feature" before starting development on it. When a feature is ready, a developer should request the core developers to pull changes from this feature branch to the `master` of the core. This initiates the code review, which concludes with the merge of the feature into the `master`.

The separation of feature development from the codebase allows for a detailed code review process before merging into the main codebase. This allows the core developers to comment on proposed changes in a review and to request further action before merging them into the main codebase. Code reviews are interactive and easy if you're using a cloud-based solution to host your central repository, allowing feature-rich discussions before merging the code into the main repository.

PROS AND CONS

Interestingly, a consequence of this workflow is that a developer must never directly commit to the local master branch. When their feature is accepted into the core repository, they should pull changes from the core master to the local master to keep it up to date.

The departure from the centralized workflow makes it easy for developers to work on multiple features, while keeping the core codebase operational. At the same time, any critical bug fixes are directly committed to the master branch, and pulled into the feature branches being worked on.

As the core developers have a final say before a merge into the main codebase, you can selectively give write access to your master branch, thus making the process secure. An open-source project must follow at least a feature-branch workflow to ensure code reviews happen before code merges from contributors. Finally, this workflow gives you more control over your code, and therefore solves the issues of scaling up.

WHO SHOULD USE THE FEATURE-BRANCH WORKFLOW

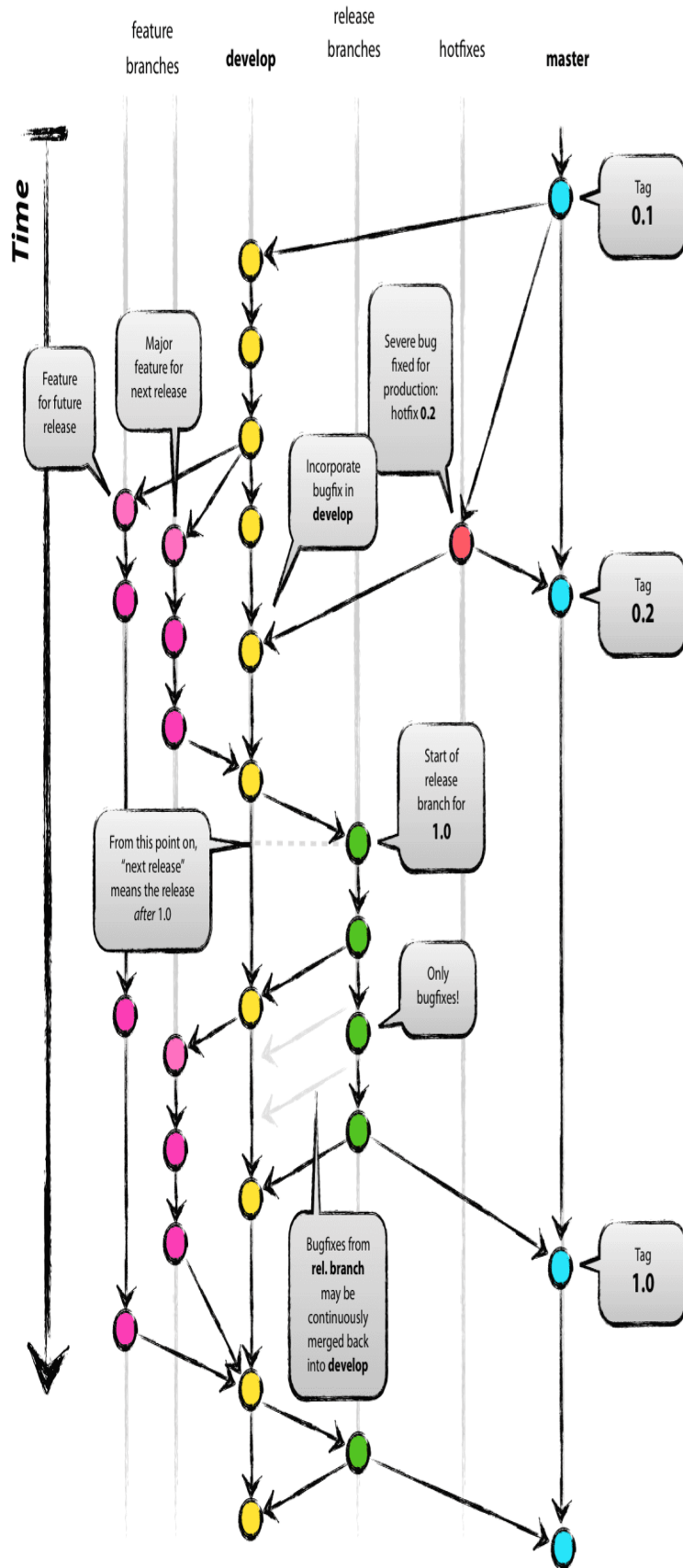
The feature-branch workflow provides a key benefit over centralized workflows: the ability to logically manage multiple changes and multiple contributors. If you initially followed the centralized workflow because your team was small, you may need to switch to the feature-branch workflow as your team grows. The target of every team project following the centralized workflow should be to eventually migrate to the feature-branch workflow.

Gitflow Workflow

FEATURES

While the feature-branch workflow works well for any project, adding specific roles for different types of

branches can further tighten up your development cycle. The Gitflow workflow was initially tested and popularized by [Vincent Driessen at nvie.com](http://nvie.com). The core Git concepts involved in the Gitflow workflow remain the same as the feature-branch workflow.



As mentioned, the Gitflow workflow works as an extension of the feature-branch workflow. A primary issue with the feature-branch workflow is the loose definition of the term “feature”: it could potentially be interpreted differently depending on the developer, culture, or project. Further, there was no well-defined provision for other tasks such as regular maintenance and bug fixes. The Gitflow workflow essentially solves this by defining many types of branches and their functions.

There are two core branches in the Gitflow workflow—the `master` and `develop` branches. The `develop` branch serves as the latest development version of the software, while the `master` branch contains only the last stable release.

If you want to work on a new feature, you create a `feature` branch from the `develop` branch. Once you’ve finished working on your feature, you request a merge to the `develop` branch. As a developer, you’re essentially never directly involved with the `master` branch, treating the `develop` branch as a pseudo master.

In addition to a feature branch, the Gitflow workflow also defines a `release` branch. All the planned changes, based on your roadmap for what should feature in the next release cycle, go into the `release` branch. A `release` branch is created from the `develop` branch. When all the changes for a release are done, it’s merged with the `master` with a relevant tag attached to it. Only the core developers get to work on the `release` branch by choosing which merges should go into it.

Finally, the next type of branch in the Gitflow workflow is the `hotfix` branch. Any critical bug that’s identified needs to be fixed immediately, so a `hotfix` branch is created from the `master` branch to solve the bug. Once

it's solved, the hotfix branch is merged with the master and develop branches to ensure the changes are reflected in both of these.

NEW TEAM MEMBER ORIENTATION

Because of the concepts involved, the Gitflow workflow is certainly more complex than the feature-branch workflow. Even so, getting started is arguably not very difficult. A developer working on just a single feature only needs to be concerned with the develop branch and a corresponding feature branch. Once the developer's role in the project grows, they may be introduced to new tasks and given further responsibilities.

PROS AND CONS

The Gitflow workflow shares the same advantages as the feature-branch workflow, with the added clarity of handling various scenarios in the software development cycle. Even if you're encouraged to adopt this workflow from the start, you may consider moving to it from the feature-branch workflow once your project has matured a bit.

WHO SHOULD USE THE GITFLOW WORKFLOW

The Gitflow workflow allows a team to manage a number of scenarios effectively. As the Gitflow workflow is an extension of the feature-branch workflow, the transition is easy to handle. Imagine that you have a project with a good mix of new and experienced developers. The end product also has a significant number of users, who may be using multiple versions of it. Such a project demands the use of the Gitflow workflow to effectively handle any situation that may come up. Popular open-source projects often use the Gitflow workflow.

Forking Workflow

FEATURES

The **forking** workflow is an implementation of the feature-branch workflow—in the cloud. It introduces an extra layer between the central repository of the organization and the local repository of a developer—known as a “fork”.

A **fork** is a developer’s personal copy of the central repository within the cloud. When you use a cloud-based Git solution, a developer clones their own fork from the cloud. Any changes they make on the local repository are pushed to this fork. To merge the code into the main repository, the developer creates a **pull request** from the fork to the main repository. This pull request initiates a code review, which the administrators of the repository assess before merging into the main repository.



ISSUE 5256: delete file storage comment #65

stahyan wants to merge 0 commits into autoraster from stahyan:rester



amk reviewed on 29 Apr 2013

View changes

```

mods_standard/file_storage/delete_comment.php Outdated
...
@@ -54,9 +54,10 @@
54 54 require(AT_INCLUDE_PATH.'header.inc.php');
55 55
56 56 $hidden_vars = array('id' => $id, 'ot' => $owner_type, 'oid' => $owner_
57 - $msg->addConfirm('DELETE', $hidden_vars);
57 + $sql = "SELECT comment FROM ".TABLE_PREF.A."files_comments WHERE commen

```



amk on 29 Apr 2013 Contributor



...

Please utilize queryDB for any MySQL queries. You can find documentation right here: <https://demo.audiospaces.com/documentation/developer/guidelines.html#querydb>

Read it carefully since this function incorporates sanitization of the passed arguments.



stahyan on 29 Apr 2013 Author



...

Hi,

Thanks for the tip.

I also display the query through AT_print to sanitize the results for display, as Harris pointed out. Also, other queries in the page were done manually, hence I did the same.

I will make the required changes, and get back to you soon.



stahyan on 29 Apr 2013 Author



...

Done.

Thanks again :)



Reply...

NEW TEAM MEMBER ORIENTATION

A new member first creates a fork of the main repository on the cloud, and then clones this fork to a local machine. Changes are made to a new feature branch and pushed to the developer's fork on the cloud. Next, the developer creates a pull request from a feature branch of the fork to a corresponding branch in the main repository. This initiates a review and conversation with the core developers to get the changes merged into the codebase.

The `origin` remote of the local repository typically points to the fork, and the `upstream` remote points to the central repository.

PROS AND CONS

Code management and review through pull requests is much easier when it's done on the cloud with the help of the web GUI of cloud hosts. Multiple developers can also get involved in the review process through a pull request.

At the same time, it may be overwhelming for a new member to work around Git's features and related cloud concepts towards the beginning of their project tenure.

WHO SHOULD USE THE FORKING WORKFLOW

Anyone using the cloud for their Git repositories should implement the forking workflow! This workflow serves as an additional layer to any other workflow. Thus, it allows you to incorporate features of other workflows without any issues.

If your project is open source, you shouldn't have any issue with hosting the code on the cloud. However, if you don't want the code of your project to be publicly available, you can use private repositories on the cloud.

Further, if your code is highly sensitive and you can't afford to have it on a public cloud, you can try the enterprise solutions of [GitHub](#) or [Bitbucket](#), which allow you to host your code on your own servers.

Conclusion

Now that we've discussed various workflows, you may wonder which is right for you. As mentioned earlier, a workflow is supposed to serve as a set of guidelines that you follow when managing your code with Git.

Therefore, you don't really need to be tied to a specific workflow to use Git. For instance, you may follow the Gitflow workflow but never use a release branch.

WHAT HAVE YOU LEARNED?

In this chapter, we covered:

- what workflows are
- the centralized workflow
- the feature-branch workflow
- the Gitflow workflow
- the forking workflow

WHAT'S NEXT?

In the next chapter, we'll explore common mistakes in Git. First, we'll focus on amending errors while working with Git. Then we'll move on to debugging in Git with two useful commands—`blame` and `bisect`.

Chapter 6: Correcting Errors While Working with Git

In the last few chapters, we've built a good foundation in Git basics. We've gone through the basic Git commands, followed by some more advanced processes that help you contribute to an organization. Up to this point, we haven't discussed how to fix mistakes you might make while working with Git.

Alexander Pope once said “To err is human”—and it's only human to commit mistakes during the Git workflow. Git makes it possible to correct mistakes at each stage of a project—which is yet another reason why it's so popular with developers.

In this chapter, we'll first look at how you can correct your own mistakes. Then we'll look at how to weed out bugs introduced at various points into a repository either by you or by others.

Amending Errors in the Git Workflow

With Git, it's fairly easy to undo changes you've made. In this section, we'll look at three examples: undoing a stage operation; undoing a commit, by reverting back to an older commit; and undoing a push, by rewriting the history of a remote repository.

UNDO GIT ADD

The `git add` command either tells Git to track an untracked file, or to stage the changes in a tracked file for a commit.

If you've just asked Git to track a new file that you've created but not yet committed—let's call it `mistake_file`—you can undo the operation by running the following command:

```
git rm --cached mistake_file
```

Here, `rm` stands for remove (just like the regular terminal command `rm`). When we postfix `--cached`, we ask Git to untrack the file, but let it remain in the file system.

Why Can't I Just Delete the File?

If we simply delete the file, Git will show that a tracked file has been deleted—a change that needs to be staged and committed to appear in the history.

You can check the status of the repository to confirm that the file is untracked again:

```
$ echo "something" >mistake_file
$ git add mistake_file
$ git status
On branch master

Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
    new file:   mistake_file

$ git rm --cached mistake_file
rm 'mistake_file'
$ git status
On branch master

Untracked files:
  (use "git add <file>..." to include in what will be committed)
    mistake_file

nothing added to commit but untracked files present (use "git add" to track)
```

The command `git rm --cached` can also be used to remove a file from the repository. Once a file has been removed, you need to commit the changes for them to take effect.

Forced Removal

If you run just `git rm` without the `--cached` option, it will lead to an error. The other option that can be postfixed with `git rm` is `-f` for forced removal. The `-f` option untracks the file and then removes it from your local system altogether. Therefore, you should be careful when you're removing tracked files if you use this option. All the same, there is way to backtrack from `rm -f` too. Even if you commit after using `rm -f` on a file, you can still get the file back by reverting to an old commit. We'll discuss the process of reset and reverting to an old commit shortly.

Let's say you make changes to a tracked file (`myfile2`), and then run `git add` to stage it for commit. Then you realize you made a mistake before committing it. You can run the following command to unstage the changes:

```
$ git status
On branch master

Changes to be committed:
  (use "git restore --staged <file>..." to
  unstage)
    modified:   myfile2

$ git reset HEAD myfile2
Unstaged changes after reset:
M   myfile2
$ git status
On branch master

Changes not staged for commit:
  (use "git add <file>..." to update what will be
  committed)
  (use "git restore <file>..." to discard changes
  in working directory)
    modified:   myfile2

no changes added to commit (use "git add" and/or
"git commit -a")
```

This command resets a file to the state where the HEAD, or the last commit, points to. This is the same as “unstaging” the changes in a file.

Once you've unstaged the changes in a file, you can undo the changes you made in the file as well, reverting it back to the state during the last commit. This is where the following command comes in:

```
$ git checkout myfile2
Updated 1 path from the index
$ git status
On branch master
   (use "git push" to publish your local commits)
nothing to commit, working tree clean
```

We've seen the `checkout` command used previously during the process of branching. It's also used to restore any unstaged changes in a file, as seen above.

What Does `checkout` Really Do?

Basically, `checkout` updates the file(s) in the current status of the repository to an earlier version.

When we were changing branches, `checkout` changed the status of files to a different branch. In this case, `checkout` restores the file to its version at the time of the last commit in the branch.

UNDO GIT COMMIT

If you've already committed your changes and then realize your mistake, there's a way to undo that too. Let's make an unnecessary commit and try to revert back to the original. Run the following command to see Git do some magic:

```
$ git reset --soft HEAD~1
$ git status
On branch master
Your branch is up to date with 'origin/master'.

Changes to be committed:
  (use "git restore --staged <file>..." to
  unstage)
   modified:   tests.py
```

The `--soft` option undoes a commit, but it lets the changes you made in that commit remain staged for you to review. `HEAD~1` means that you want to go back one commit from where your current `HEAD` points (which is the last commit).

What's with HEAD~1?

We encountered HEAD earlier, and we know that it points to the last commit in the current branch. I've added ~ to HEAD in the example above. This refers to the parent of the last commit in the current branch. You can also use ^. Using either ~ or ^ refers to the parent of the last commit in the current branch, while -- and ^^ both refer to the grandparent of the last commit in the current branch. You can also add numbers to move back a specific number of commits in the hierarchy. However, adding numbers after either ~ or ^ can mean different things:

- ~2 goes up two levels in the hierarchy of commits, via the first parent if a commit has more than one parent.
- ^2 refers to the second parent, where a commit has more than one parent (which could be the result of a merge).

You can also combine these postfixes. For instance, HEAD~3^2 refers to the second parent of the great-grandparent commit, which you reached through the first parent and grandparent.

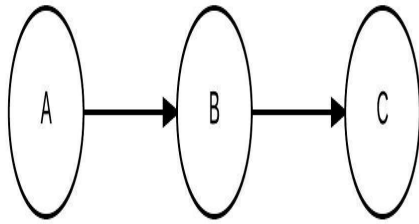
The second option here is postfixing the --hard option to permanently undo commits. It's generally advised that you avoid using the --hard option—unless you're absolutely sure you want to do away with the commits.

A third option of reset is --mixed, which is also the default option. In this option, the commit is reverted, and the changes are unstaged.

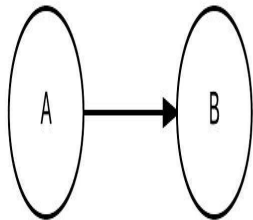
The process of committing involves three steps: making changes in a file, staging it for a commit, and performing a commit operation. The --soft option takes us back to just before the commit, when the changes are staged. The --mixed option takes us back to just before the staging of the files, where the files have just been changed. The --hard option takes us to a state even before you changed the files.

There's yet another Git command that could help you in case you've committed changes by mistake. This is the revert command. The reset command changes the history of the project, but revert undoes the changes made by the faulty commit by creating a new commit that reverses the changes. Figure 5-1 shows the difference between revert and reset.

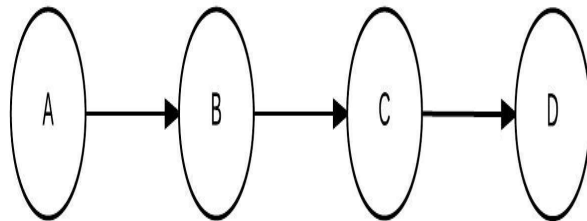
Before Reset/Revert



After Reset



After Revert



The following code shows how to go back one commit using `revert`. You can also modify the commit message for the commit that reverses the changes of the unwanted commits:

```
$ git revert HEAD~1
[master 623a519] Revert "Update data.csv"
 1 file changed, 6 insertions(+), 6 deletions(-)
$ git log --oneline
623a519 Revert "Update data.csv"
25313e5 Added new CSV file
c76ee85 Update data.csv
0d0d493 Added csv data
```

You can change the commit message of the last commit by running the following command:

```
$ git commit --amend -m "New Message"
[master 8a15b20] New Message
Date: Sun Mar 22 00:48:43 2020 +0530
 1 file changed, 6 insertions(+), 6 deletions(-)
$ git log --oneline
8a15b20 New Message
25313e5 Added new CSV file
c76ee85 Update data.csv
0d0d493 Added csv data
083e7ee Added yet another test
```

The `--amend -m` option changes the commit message of the last commit. Notice that the hash changes too, effectively rewriting the history.

UNDO GIT PUSH

In case you've also pushed your changes to a remote, it's possible to revert changes in the push too.

The simplest way is to go for a `revert` and push the new commit that undoes the changes:

```
git revert HEAD~1
git push origin master
```

However, if you also want the other commit(s) to vanish from the remote repository, you first need to go for a `reset` command—deleting the unwanted commit—and then push the changes to the remote. If you perform a normal `git push`, the push will be rejected, because the origin HEAD is at a more advanced position than your local branch. Therefore, you need to force the change with a postfix—`-f`—which forces the push on the remote origin:

```
git reset --hard HEAD~2
git push -f origin master
```

Use `-f` with Caution

Postfixing `-f` is a dangerous move, as it rewrites the remote without confirming it. Make sure you double-check your local changes before going for an `-f` push.

Debugging Tools

The scenarios we've discussed so far help you to undo changes in Git. They've dealt with mistakes you've committed in the near past and want to correct. Now we'll look at dealing with bugs introduced by you or others in the past. This will involve exploring tools in Git that help in the process of debugging. These tools are required when you're working on a relatively large codebase with a large number of contributors.

You may or may not know the location of the bug. If you know which file or set of files is the source of the bug, you can debug with `git blame`. If you don't know the source of the bug, you can debug with `git bisect`. If you've written unit tests, you can also automate the process of debugging. So let's explore the different ways of debugging your code in Git.

GIT BLAME

Running the `git blame` command on a file gives you detailed information about each line in the file. `git blame` lists the commits that introduced changes in a file, along with basic information about the commit, like the commit hash, author and date.

`git blame` is usually used when you know which file is causing a bug. Let's see how it works:

```
$ git blame my_file
^8dd76fc (Shaumik 2019-05-06 15:28:03 +0530 1)
This is some information!
f934591c (Shaumik 2019-05-06 15:31:00 +0530 2)
cc48fb3c (Shaumik 2019-06-11 22:38:21 +0530 3)
Adding Line 1.
cc48fb3c (Shaumik 2019-06-11 22:38:21 +0530 4)
f934591c (Shaumik 2019-05-06 15:31:00 +0530 5)
Changing the content of this file.
7534bc23 (Shaumik 2019-05-15 03:16:48 +0530 6)
cc48fb3c (Shaumik 2019-06-11 22:38:21 +0530 7)
Adding Line 2.
cc48fb3c (Shaumik 2019-06-11 22:38:21 +0530 8)
7534bc23 (Shaumik 2019-05-15 03:16:48 +0530 9)
This change is in the master branch!
96f7c5e6 (Shaumik 2019-05-15 03:17:18 +0530 10)
Another line in the master branch.
cc48fb3c (Shaumik 2019-06-11 22:38:21 +0530 11)
cc48fb3c (Shaumik 2019-06-11 22:38:21 +0530 12)
Adding Line 3.
b1175163 (Shaumik 2019-05-10 00:44:48 +0530 13)
b1175163 (Shaumik 2019-05-10 00:44:48 +0530 14)
Adding yet another line after sum.py
```

As you can see in the code above, the command `git blame` displays each line of the file. These lines are prepended with information in the following order: the hash of the commit that added the line, and the commit author, date, time and time zone.

In this scenario, as you already know where the faulty code is, you can just display the details of the required commit to find out more about the bug that was created. Let's assume it was commit `f934591c` that introduced the bug. You should therefore run the following:

```
$ git show f934591c
commit f934591cd1c04e4009dfa76a9684dda73cb30260
Author: Shaumik <sdaityari@gmail.com>
Date: Tue May 6 15:31:00 2019 +0530

    - Changed two files
    - This looks like a cooler interface to write
commit messages

diff --git a/my_file b/my_file
index 362eab3..0a0bd57 100644
- a/my_file
+++ b/my_file
@@ -1,3 @@
   This is some information!
+
+I am changing the content of this file.
diff --git a/myfile2 b/myfile2
index d4a2d15..ec4dcc2 100644
- a/myfile2
+++ b/myfile2
@@ -1,1 @@
-This is another file!
+This is another file! Changing this file too.
```

The `git show` command shows the author of the commit, the date of the commit and the changes that constitutes the commit. Once you've figured out what caused the error, you can go ahead and fix it in your repository and then commit the changes.

Normally, though, you'll most likely have no idea what caused the bug. So we need to explore some more debugging tools.

GIT BISECT

There's probably no better way to search for a bug than with `bisect`. Even if you have a thousand commits to check, `bisect` can help you do it in just a few steps.

Let's assume you have no idea what's causing an error. However, you *do* know that, at a certain point in time—after a particular commit—the bug wasn't present in your code. Git's `bisect` helps you quickly traverse between these stages to identify the commit that introduced the

`git bisect` essentially performs a binary search through these commits.

To start the process, you select a “good” commit from the history, where you know the bug wasn’t present, and a “bad” commit (which is usually the latest commit). Git then changes the state of your repository to an intermediate commit and asks you if the bug is present there. You search for the bug and assign that commit as “good” or “bad”. This process continues until Git finds the faulty commit. Since a binary search algorithm is used, the number of steps required is a logarithmic value of the number of commits in between the initial “good” and “bad” commits.

An example will help explain how `git bisect` works. Let’s create a file in our repository—`sum.py`—containing a function that adds two numbers in Python. The contents of the file are as follows:

```
#sum.py
def add_two_numbers(a, b):
    """
    Function to add two numbers
    """
    addition = a + b
    return addition

if __name__ == '__main__':
    a = 5
    b = 7
    print(add_two_numbers(a, b))
```

I’ve intentionally added the second block of code to print the response of the function to two dummy values. We can run the program with the following:

```
python sum.py
```

After adding a few more commits, let’s change the file `sum.py` to introduce an error:

```
#sum.py
def add_two_numbers(a, b):
    '''
        Function to add two numbers
    '''
    addition = 0 + b
    return addition

if __name__ == '__main__':
    a = 5
    b = 7
    print(add_two_numbers(a, b))
```

Running the program now, we can see that the result is not 12, but 7. Let's now demonstrate the use of `git bisect`. To decide the good and bad commits, we need to have a look at the commit history:

```
$ git log --oneline
083e7ee Added yet another test
49a6bec Added more tests
5199b4e ERROR COMMIT: Introduced error in sum.py
b00caea Added tests.py
b117516 Dummy Commit after adding sum.py
7d1b1ec Added sum.py
```

As is evident from the history, the latest commit 083e7ee (at the top) is “bad”, whereas the commit two positions before we introduced the bug—7d1b1ec—is “good”. To better identify the bug, I've mentioned in the commit message which commit introduced the error. We must now undertake the following steps to find out the bug:

- start the Git bisect wizard
- select a good commit
- select a bad commit
- assign commits as good or bad as the wizard takes you through the commits
- end the Git bisect wizard

Let's go ahead and start the Git bisect wizard:

```
git bisect start
```


This takes Git into a binary search mode. Next, we need to tell Git the last known commit where the bug was absent, which in our case is 7d1b1ec:

```
git bisect good 7d1b1ec
```

Now assign the latest commit as the bad one:

```
$ git bisect bad 083e7ee
Bisecting: 2 revisions left to test after this
(roughly 1 step)
[b00caea53381979ec1732d919d6f76e3baaf80fc] Added
tests.py
```

Why is git bisect So Fast?

Notice that, in the code above, the bisect wizard tells you that there are two revisions left for us to perform in this process until it ends. Because bisect essentially performs a binary search, at each step it tries to cut the number of revisions to check by half. In our case, there are six commits to check, which will take about two steps. But 100 commits would require roughly seven steps, and 1000 commits would require about ten steps.

To combine the last three commands (start, good, and bad) into one, you may instead start the wizard with the following command:

```
git bisect start 083e7ee 7d1b1ec
```

As soon as you assign the good and bad commits, git bisect starts its work and takes the state of your repository to an intermediate commit. At this point, you're shown the commit hash and commit message, and you're asked whether or not the bug is present in that commit.

Learn More About Each Commit

If you want to know more about a commit during the time the bisect wizard is running, you can run `git show` for the commit.

In our situation, we just run the file `sum.py` to find out if the bug is present. For the commit `b00caea`, we see that the output is `12`. So the bug is absent, and we mark it as good:

```
git bisect good
```

In the next step, we're asked whether commit `49a6bec` is good. We check the commit by running `sum.py` again and assign it as bad:

```
git bisect bad
```

Once we're done with this, Git shows us the faulty commit as `5199b4e`, which is also evident from the commit message I added when I introduced the error:

```
$ git bisect bad
Bisecting: 0 revisions left to test after this
(roughly 0 steps)
[5199b4e10ba04b63ed1e76118259913123fbf72d] ERROR
COMMIT: Introduced error in sum.py
```

Once you've found your faulty commit, you can exit the wizard by running the following:

```
git bisect reset
```

In this case, the use of `git bisect` was overkill and not necessary (as we knew the source of the bug already).

However, in real life there are often bugs that are difficult to trace back to a file, but the bug is visible only in the way your code functions. For instance, you have a complex algorithm to find out the popularity of a person in social media and you find out that the results aren't right. In such cases, you employ the `bisect` tool to find out which commit first introduced the error to rectify it.

AUTOMATED BISECT WITH UNIT TESTS

We've just seen how `bisect` helps you find the commit that introduced a bug. However, this process is tedious, as you need to check for the bug at every single step of the wizard.

The easiest way to automate the process is to write unit tests. You can also write custom scripts that test the required functionalities. In our case, we'll write a custom file—`test_sum.py`—that tests the functionality of the function in `sum.py`. This file is just for demonstrating the functionality of `bisect`. (You don't need to understand the code here. To learn more about testing in Python, you can read about Python's [`unittest` module](#).)

Exit Codes in Custom Shell Scripts

If you create a custom shell script to perform your tests, make sure it has custom exit codes, in addition to printing messages on the terminal about the status of the tests. In general, the 0 exit code is considered a success, whereas everything else is a failure.

```
#test_sum.py
import unittest
from sum import add_two_numbers

class TestsForAddFunction(unittest.TestCase):

    def test_zeros(self):
        result = add_two_numbers(0, 0)
        self.assertEqual(0, result)

    def test_both_positive(self):
        result = add_two_numbers(5, 7)
        self.assertEqual(12, result)

    def test_both_negative(self):
        result = add_two_numbers(-5, -7)
        self.assertEqual(-12, result)

    def test_one_negative(self):
        result = add_two_numbers(5, -7)
        self.assertEqual(-2, result)

if __name__ == '__main__':
    unittest.main()
```

Running the file `test_sum.py` runs the tests specified in it:

```
python test_sum.py
```

Running it on our current code shows errors.

Let's start the bisect process again:

```
git bisect start 083e7ee 7d1b1ec
```

We next inform Git about the command that runs the tests:

```
git bisect run python test_sum.py
```

If you have a custom command to run your tests, replace `python test_sum.py` with your command.

On informing Git about the command that tests our code, the wizard runs it against the remaining commits and figures out which commit introduced the error.

Once the bug has been identified, reset the wizard:

```
git bisect reset
```

Beware of Using Old Test Files

If you're using a testing script for the process of running bisect, be aware that when Git is testing an old commit, it's also checking against the old version of the testing script.

You can instead provide your new test to the command by copying it outside the repository and modifying the test command. Even when old commits are being tested, your latest test files will be used for the testing process.

Once you've found out which commit introduced the error, you can look carefully into it to see the faulty code. Once you identify that, you can fix it and commit it to the repository.

Conclusion

WHAT HAVE YOU LEARNED?

In this chapter, we looked at how Git lets you undo mistakes:

- `undo git add`
- `undo git commit`
- `undo git push`

We've also looked at two debugging tools, which help you find bugs in your Git workflow:

- `blame`
- `bisect`

WHAT'S NEXT?

In the next chapter, we'll look at a list of useful commands that help you use Git to its fullest.

Chapter 7: Unlocking Git's Full Potential

So far in this book, we've covered the fundamentals of Git and some of its advanced commands. In this chapter, we'll look at more of these advanced commands.

Advanced Use of `log`

We saw earlier that you can view the history of your project in Git using the `log` command. However, in busy repositories that handle hundreds to thousands of commits each day, a long list of commits isn't going to be useful unless you know how to navigate through them.

The [manual entry for the `log` command](#) shows the different options that can be postfixed to this command to get a desired output. We'll look at a few tweaks to the `log` command, which could prove useful in such situations.

Since [our dummy project](#) doesn't have a considerable number of commits, we're going to use the open-source repository of an e-learning management system—[ATutor](#)—to explore the different capabilities of the `log` command.

SHORT VERSION

In general, the `log` command shows a list of commits in the active branch, each with the commit hash, author, date and commit message. Depending on your screen size and text size of the output, you get around five to ten commit details in a screen. Each commit occupies four to five lines on the screen, or even more if the size of the commit message is large:

```
commit 8a15b207acabf8abdd1750be48f1d748d51fb857
Author: Shaumik Daityari <sdaityari@gmail.com>
Date: Sun Mar 22 00:48:43 2020 +0530
```

New Message

In case you want to have a quick glance at the list of commits, you can format the output to show only the commit hashes and single-line messages, using the `--oneline` option:

```
git log --oneline
```

A single commit is displayed on each line, and thus many more commits fit onto the screen at once:

```
8a15b20 New Message
cc251c6 Updates analytics account
31fb3d7 Addes 301 header to redirects
af519cf Created a more general function to check
referrer
```

BRANCHES AND HISTORY

The `log` command can also be used to view the workflow and commits in branches other than the current active branch. If you want to view the commits in all branches, just postfix `--all` to the command:

```
$ git log --all
commit 8a15b207acabf8abdd1750be48f1d748d51fb857
(HEAD -> master)
Author: Shaumik Daityari <sdaityari@gmail.com>
Date: Sun Mar 22 00:48:43 2020 +0530
```

New Message

```
commit d04ec3fa6e136d37ae16459ff8bde3ba8f0924a7
(origin/another_feature)
Author: Shaumik Daityari <sdaityari@gmail.com>
Date: Sun Feb 9 01:24:48 2020 +0530
```

Commit Message

```
commit 25313e5016bea8b3ae470230b343c9ae1ebccc87
(origin/master, origin/HEAD)
Author: Shaumik Daityari
```

```
<sdaityari@users.noreply.github.com>  
Date: Tue Oct 8 21:09:00 2019 +0530
```

```
Added new CSV file
```

```
commit c76ee85387b5dcaf82ac676d15ccc952c927528b  
Author: Shaumik Daityari  
<sdaityari@users.noreply.github.com>  
Date: Sun Sep 2 02:51:40 2018 +0530
```

```
Update data.csv
```

This doesn't look very appealing, as you have no idea which commit came from which branch. You can add the `--decorate` option to view which branch each commit belongs to. It also shows the remote branches. Note that I've used `--oneline` to accommodate more commits:

```
$ git log --all --decorate --oneline  
8a15b20 New Message  
d04ec3f (origin/another_feature) Commit Message  
25313e5 (origin/master, origin/HEAD) Added new CSV  
file  
c76ee85 Update data.csv  
0d0d493 Added csv data  
083e7ee Added yet another test  
49a6bec Added more tests  
5199b4e ERROR COMMIT: Introduced error in sum.py
```

The `--graph` option shows you the commit history, with a graphical representation interconnecting the links between commits of different branches (if any). Combining it with `--all` shows you how the different branches in your repository have progressed:

```
$ git log --all --decorate --oneline --graph  
* 8a15b20 New Message  
* 25313e5 (origin/master, origin/HEAD) Added new  
CSV file  
* c76ee85 Update data.csv  
* 0d0d493 Added csv data  
* 083e7ee Added yet another test  
* 49a6bec Added more tests  
* 5199b4e ERROR COMMIT: Introduced error in sum.py  
* b00caea Added tests.py  
* b117516 Dummy Commit after adding sum.py  
* 7d1b1ec Added sum.py  
* b198692 Cleaned junk  
* 7ac171f Made some change to myfile2
```



```

*   cafb55d Merge commit '5ef655a4caf8'
| \
|  * 5ef655a Fixed conflict from another_feature
|  branch
|  * | cc48fb3 Added lines 1 and 3 using add -p
|  * | 96f7c5e Another change in the master branch
|  * | 7534bc2 Some change in the master branch
|  | * d04ec3f (origin/another_feature) Commit
|  | Message
|  | /
|  | * 49ed357 Added another feature
|  | /
|  * 7e0eea2 (origin/new_feature) Removed line
|  * f87d1a5 Dummy change
|  * f934591 - Changed two files - This looks like a
|  cooler interface to write commit messages
|  * 8dd76fc My first commit

```

To understand this concept better, let's take a look at the output again. `8dd76fc` is the first commit of this repository, which appears at the bottom of the output. As you traverse upwards from the bottom of the figure, notice that commit `49ed357` diverges from the master branch into a new branch, `another_feature`. Following the path of `another_feature` shows us that commit `5ef655a` is the last commit in the branch, before it merges back with master at commit `cafb55d`.

FILTER COMMITS

When you view the history, you're shown all the commits in the history's branch. However, if you wish to view only a few of the latest commits, postfix `-n`, followed by the number of commits you want to see:

```
git log -n 2
```

Alternatively, you can use the following command as well, which serves as a shortcut for the previous command:

```

$ git log -2
commit 8a15b207acabf8abdd1750be48f1d748d51fb857
(HEAD -> master)
Author: Shaunik Daityari <sdaityari@gmail.com>
Date: Sun Mar 22 00:48:43 2020 +0530

```

New Message

```
commit 25313e5016bea8b3ae470230b343c9ae1ebccc87
(origin/master, origin/HEAD)
Author: Shaumik Daityari
<sdaityari@users.noreply.github.com>
Date: Tue Oct 8 21:09:00 2019 +0530
```

Added new CSV file

You can also view the commits in a specified time range. This can be achieved by postfixing `--after` and `--before` to the `log` command:

```
$ git log --after='2019-3-1' --before='2020-3-1'
commit 25313e5016bea8b3ae470230b343c9ae1ebccc87
(origin/master, origin/HEAD)
Author: Shaumik Daityari
<sdaityari@users.noreply.github.com>
Date: Tue Oct 8 21:09:00 2019 +0530
```

Added new CSV file

`--after` and `--before` can be replaced by `--since` and `--until`. For instance, the following pairs of commands will produce the same results:

```
git log --after='2019-3-1'
git log --since='2019-3-1'

git log --before='2020-3-1'
git log --until='2020-3-1'

git log --after='2019-3-1' --before='2020-6-1'
git log --since='2019-3-1' --until='2020-6-1'
```

You Must Specify a Range

The specified dates have to signify a date range, as it doesn't make sense for Git to search for a commit at a point in time. If you want to find the commits on a particular day, you need to specify the whole day in the range.

You can also use date references such as “Yesterday” or “1 week ago”, as explained by [Alex Peattie on his blog](#).

TRACE CHANGES IN A SINGLE FILE

If you want to check the commits that resulted in changes in a single file, you can use the `--follow` option:

```
$ git log --oneline --follow tests.py
083e7ee Added yet another test
49a6bec Added more tests
b00caea Added tests.py
```

Tracing the changes in a file may be useful while debugging, especially if you want to see if anyone has changed a particular file since a certain time. It also helps you to check if parts of a file were removed in previous commits.

How Is Tracing Different from `git blame`?

We used the `blame` command earlier to get more information about each line in a file, and which commit it's associated with. `blame` enables you to check only the current contents of a file. The `log --follow` command, on the other hand, lists the changes the file has gone through since Git started tracking the file.

Therefore, any part of the file that was removed in an earlier commit will show up on the output of `log --follow`, but not on `blame`.

TRACK YOUR PEERS

The `shortlog` is a command that shows the authors who've contributed to the repository, their commits and commit messages. You can use this command if you're interested in knowing the contributions of different developers.

The output of this command is sorted by name, and you can postfix `-n` to sort it by the number of commits:

```
$ git shortlog
git shortlog
Shaumik (18):
```

```
My first commit
- Changed two files - This looks like a
cooler interface to write commit messages
Dummy change
Removed line
Added another feature
Some change in the master branch
Another change in the master branch
Fixed conflict from another_feature branch
Added lines 1 and 3 using add -p

Shaumik Daityari (4):
  Added csv data
  Update data.csv
  Added new CSV file
  New Message
```

Notice that there are two different authors with the same name, as the second set of commits was created on GitHub.

You can also view the commits by a single author by using the `--author` option:

```
git log --author='Shaumik'
```

You only need to type just enough of the name for Git to identify the author. If there are two authors matching the string you've provided, both their commits will be displayed. If there are two authors with the same name committing to the same repository, Git differentiates them through other details—such as their email address, or the system the commit was generated from.

SEARCH IN COMMIT MESSAGES

Imagine a situation where you'd like to know when a certain feature was introduced. Searching for a commit through its commit message would be useful. Git enables searching in the commit messages by using the `--grep` option. For instance, if you want to search for the word “redirect” in your commit history, you should use the following command:

```
$ git log --oneline --grep='test'  
083e7ee Added yet another test  
49a6bec Added more tests  
b00caea Added tests.py
```

In the search term, you can also use regular expressions to search in commit messages. This would be useful in a situation where, for example, you'd like to search for all commits that refer to a certain feature name.

The Importance of Meaningful Commit Messages

When I introduced commits in this book, I mentioned the importance of writing meaningful commit messages, even though it's not mandatory. Imagine how difficult it would be to search through commits if your commit messages weren't meaningful!

You can also use regular expressions while using the `grep` command.

Using the `grep` Terminal Command

You can also use the terminal command `grep` (not to be confused with Git's `grep` option!) to search commit messages. The command for that is:

```
git log --oneline | grep 'redirect'
```

The pipe (`|`) passes on the output of the command `git log --oneline` to the second part, which searches for the word "redirect" in it.

The terminal `grep` command works on Linux and macOS, but has no native command substitute in Windows, although there's a `Findstr` command that performs a similar task. You can, however, install third-party utilities like `Cygwin` and `UnxUtils`, which enable the use of the `grep` command on Windows.

Tagging in Git

You've most likely noticed that software updates normally come with a version number. For instance, as of March 2020, the version number of popular screen recording software ScreenFlow is 9.0.0, which was released in November of 2019.

Git allows you to associate these version numbers with specific milestone commits in your repositories, by attaching labels to these commits. The labels are called **tags**. Let's again visit the ATutor repository to check its use of tags.

Tagging can be used to easily find any commit that's important to a developer. Tags can also be used to mark a breakthrough after debugging, or a milestone in development. They can also be used to mark changes being made without creating an extra branch. Tags provide an easy way to go back in branch history if something didn't work out right.

To list the tags in alphabetical order, run `git tag`, which results in the following output:

```
$ git tag
Atutor_1.4.1
atutor_1_3_1
atutor_1_3_1_rc1
atutor_1_3_2
atutor_1_3_2_rc1
atutor_1_3_2_rc3
atutor_1_4_1
atutor_1_4_2
atutor_1_4_3
atutor_1_4_rc2
atutor_1_5
```

There are two types of tags—"lightweight" and "annotated". **Lightweight** tags contain only the tag name and point to a commit. **Annotated** tags contain the tag name, information about the tagger, and a message associated with the tag.

Annotated tags are generally preferred in organizations, because they contain information about the tagger, when the tag was created, and why. Lightweight tags are handy for tagging special commits when you're working on your personal projects.

To view the details of a tag—say `Atutor_1.4.1`—run the following command:

```
git show Atutor_1.4.1
```

You can create a lightweight tag `latest_commit`, associated with the latest commit, by running the following:

```
git tag latest_commit
```

To create an annotated tag, you need to postfix `-a` for annotated and `-m` for an associated message:

```
$ git tag -a latest_commit -m "this is the latest
commit"
$ git show latest_commit
Tagger: Shaumik Daityari <sdaityari@gmail.com>
Date: Sun Mar 15 23:50:25 2020 +0530

this is the latest commit

commit 155526c8a4c35bc15716157837d02c9566b0941e
(HEAD -> master, tag: latest_commit,
origin/master, origin/HEAD)
Merge: da5f598c6 3335389bf
Author: Greg Gay <gregrgay@gmail.com>
Date: Mon Sep 9 14:06:40 2019 -0400

Merge pull request #170 from
MostafaSoliman/patch-1

Update header.php, thanks for the fix.
```

You can also checkout to a tag `Atutor_1.4.1` by creating a new branch `version_1_4_1` (just like you checkout to a commit):

```
git checkout -b version_1_4_1 Atutor_1.4.1
```

When you push your code, your tags aren't pushed to the remote. If you specifically want to push newly created tags to the remote `origin`, you can run the following:

```
git push origin --tags
```

If you specifically want to push a tag to a remote, run the following:

```
git push origin Atutor_1.4.1
```

Refs and reflog

Now that we've explored the `log` command in detail, let's now have a look at something new: refs. You already know that a commit is identified by its hash—a long string unique to a commit. A **ref**, short for a “reference”, is a way of referencing a commit. In other words, the hash is a name, whereas a ref is a pointer.

Refs are stored internally in Git, and we won't go into how Git treats refs. We will, however, use the `reflog` command to utilize refs.

We've discussed what a HEAD in Git points to. At this point, it's important to note that HEAD is also a ref. There are other such special refs like `ORIG_HEAD`, `MERGE_HEAD` and `FETCH_HEAD`.

This brings us to the `reflog`. It's a “log of refs”. That is, any change you make in Git is recorded and accessible via the `reflog` command. For instance, if you create a commit, checkout to a new branch, merge two branches, pull, push or even make a failed merge, `reflog` records them all:

```
$ git reflog
8a15b20 (HEAD -> master) HEAD@{0}: checkout:
moving from 5199b4e10b to master
5199b4e HEAD@{1}: checkout: moving from 49a6bec7c6
to 5199b4e
49a6bec HEAD@{2}: checkout: moving from b00caea to
49a6bec
b00caea HEAD@{3}: checkout: moving from master to
b00caea
```



```
8a15b20 (HEAD -> master) HEAD@{4}: commit (amend):  
New Message  
623a519 HEAD@{5}: revert: Revert "Update data.csv"  
25313e5 (origin/master, origin/HEAD) HEAD@{6}:  
clone: from  
https://github.com/sdaityari/my_git_project
```

The `reflog` command stores the records for each action you perform in your repository. When you push the changes, this data isn't synced with the server. Using the `reflog` command is necessary if you want to review changes to your local repository. It could also be used to recover lost commits.

`reflog` Can Act as Insurance

If you make a hard reset and lose a commit or two, you can safely go back to any commit you made earlier. For instance, you can run the `reflog` command, which would have a record corresponding to the time when the commit was created, mentioning the commit hash. When you know the hash, you can start a new branch based on that commit to go back to the state of that commit.

The `reflog` command is like an insurance policy in Git.

`reflog` Only Tracks Commits for a Certain Period of Time

The `reflog` command only tracks changes back for a certain amount of time. Git is responsible for cleaning up the `reflog` data periodically, which by default is 90 days. You can modify this value by specifying the `expire` option of the command. If you want `reflog` never to forget any action, run the following command:

```
git reflog expire --expire=never
```

Checking for Lost Commits

We've just seen how `reflog` can help you search for commits that might be lost because of the use of a hard reset. However, it's difficult to search specifically for lost commits in a repository with a huge history.

A commit is *lost* when it's not a part of any branch. The `log` command fails to search and show lost commits.

One way of losing commits from your branch history is to do a hard reset. However, deleting a branch without merging it with a different one can also lead to commits that are recorded by Git but not present anywhere in any of your branches.

You can search for commits that aren't a part of any branch by using the `fsck` (file system check) command:

```
>git fsck --lost-found
Checking object directories: 100% (256/256), done.
dangling commit
623a5196c885b7e8fc26d1519f3bf3d38cc97cf1
```

Not to be Confused with the Unix Command

`fsck` is also a Unix command to check for and repair inconsistencies in your file system. Don't confuse it with the `git fsck` command, which checks for inconsistencies in your commits.

If you want to recover a lost commit—say `c9067`—from the list to your current branch, you can run the following:

```
git merge c9067
```

`fsck` versus `reflog`

`fsck` has an advantage over `reflog`. Imagine you cloned a remote branch and deleted it. The commits present there would never show up on `reflog`, because they were never done on your local system. However, `fsck` will list all the lost commits from that branch.

Rebase

We saw earlier how merge works: it creates loops in the commit history of a project. These loops don't really cause any problems for Git, though over time they can make project histories difficult to understand and navigate. For the central repository of a project, it's

preferable to have a linear history, rather than a bunch of interconnected loops.

In this section, we'll discuss a merging mechanism—`rebase`—that avoids loops in the project history. I mentioned `rebase` earlier, when I used it with the `git pull` command. Quite literally, the process of **rebasing** is a way of rewriting the history of a branch by moving it to a new “base” commit.

If you're rebasing a `master` into `new_feature`, the new commits in `master` are put before the new commits in `new_feature` that aren't common to `master`. To do so, run the following command from the `new_feature` branch:

```
git rebase master
```

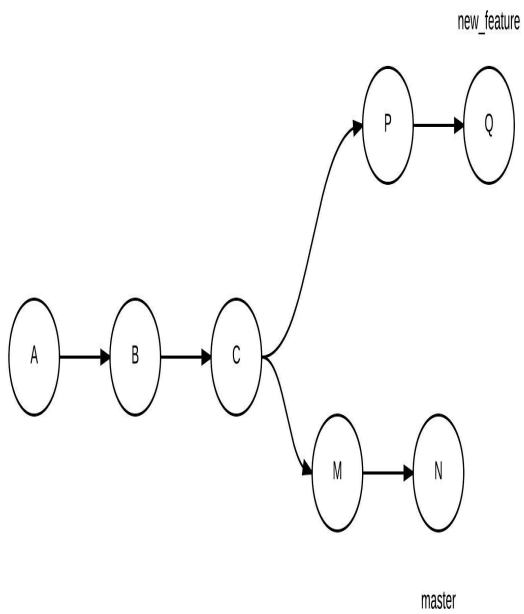
Working in a Team

If you're working in a team, you should first checkout to `master`, pull from the upstream branch to update your `master` with the latest commits, and then switch back to `new_feature` before running the above command.

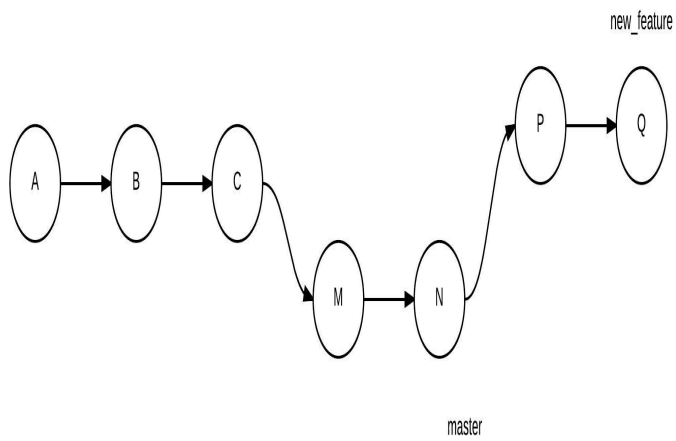
This can also be accomplished by the following:

```
git merge --rebase master
```

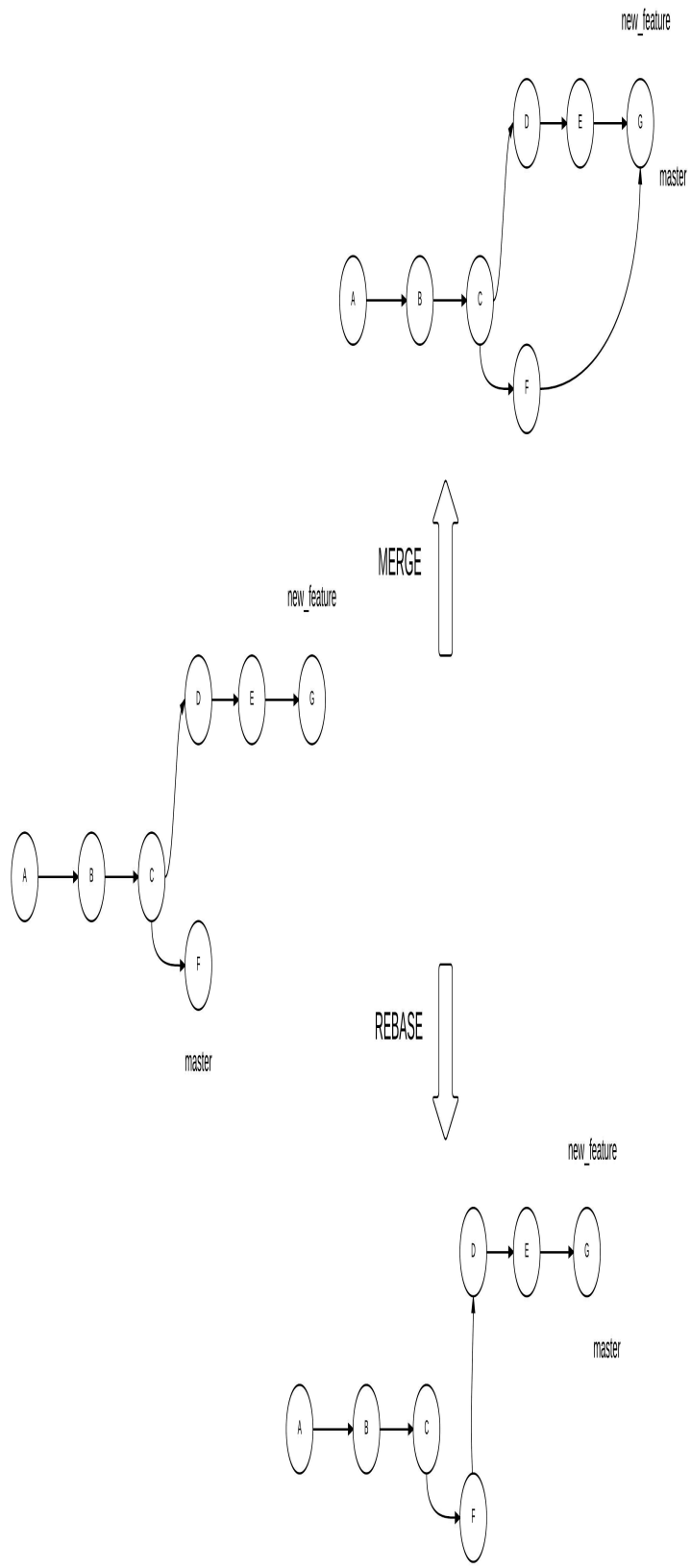
The above command is illustrated in Figure 7-1.



UPDATE new_feature BY
REBASING WITH master



One important observation from the diagram is the presence of a linear commit history, which is not present in a merge. Figure 7-2 shows the difference between a merge and a rebase for two branches.



A rebase operation may lead to conflicts, just like a merge operation. The process of resolving a conflict is exactly the same as we discussed earlier.

You can use `rebase` when you're pulling changes. It essentially puts the new commits in the master of the remote in your history, and then superimposes your commits on them. Any conflicts that arise can be fixed easily, because they've been raised by your code. You can rebase with a pull using the following command:

```
git pull --rebase origin master
```

Just for Illustration

The last command assumes that you added commits to your `master` branch and then updated it from the central repository. This is just for the sake of argument, and not the best way to work in Git. Ideally, when you work in your own branch and keep it updated using pull operations, no conflicts would arise in the `master` branch.

SQUASH COMMITS TOGETHER

When you're contributing to a codebase by working on a different branch, the code may not be accepted at the first go. Once changes in your code have been suggested, you create a new commit with the changes. You may, however, be asked to make more changes and, before you know it, you may have added multiple commits to the pull request. Since you created the pull request asking for your code to be merged, all of the commits would also get merged.

In such a situation, you might have a list of commits, the first of which was an attempt to resolve a bug, whereas the latter were attempts at refactoring the code to follow best coding practices. The group of commits as a whole

signifies a single task that's been accomplished, and hence, it makes logical sense to package them together as a single commit (rather than merging these multiple commits into the main project history).

This can also be done through the `rebase` command (essentially rebasing your current branch). If you want to squash the last two commits, run the following command:

```
git rebase -i HEAD~2
```

The `HEAD~2` refers to the last two commits in the current branch, and the `-i` option stands for interactive (which can be replaced by `--interactive`). You're then taken to an interactive screen, where you need to pick the old commit and squash the latest commit.

```
pick 3c1df4f Squash: Base Commit
squas dd43634 Squash: After First Code Review

# Rebase 083e7ee..dd43634 onto 083e7ee ( 2 TODO item(s))
#
# Commands:
# p, pick = use commit
# r, reword = use commit, but edit the commit message
# e, edit = use commit, but stop for amending
# s, squash = use commit, but meld into previous commit
# f, fixup = like "squash", but discard this commit's log message
# x, exec = run command (the rest of the line) using shell
#
# These lines can be re-ordered; they are executed from top to bottom.
#
# If you remove a line here THAT COMMIT WILL BE LOST.
#
# However, if you remove everything, the rebase will be aborted.
#
# Note that empty commits are commented out
```

You then proceed to provide a commit message.


```
# This is a combination of 2 commits.
# The first commit's message is:

# This is the 2nd commit message:

Squashed last two commits

# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
#
# Date:      Sun May 17 00:35:33 2015 +0530
#
# rebase in progress; onto 083e7ee
# You are currently editing a commit while rebasing branch 'squash' on '083e7ee'.
#
# Changes to be committed:
#   modified:   my_file
#   modified:   myfile2
#   modified:   myfile3
#
```

Let's look at the repository after the squash operation, to make sure the last two commits have been converted into one.

```

SMA:my_git_project danny$ git log --oneline
dd43634 Squash: After First Code Review
3c1df4f Squash: Base Commit
083e7ee Added yet another test
49a6bec Added more tests
5199b4e ERROR COMMIT: Introduced error in sum.py
b00caea Added tests.py
b117516 Dummy Commit after adding sum.py
7d1b1ec Added sum.py
b198692 Cleaned junk
7ac171f Made some change to myfile2
cafb55d Merge commit '5ef655a4caf8'
cc48fb3 Added lines 1 and 3 using add -p
5ef655a Fixed conflict from another_feature branch
96f7c5e Another change in the master branch
7534bc2 Some change in the master branch
49ed357 Added another feature
7e0eea2 Removed line
f87da5 Dummy change
f934591 - Changed two files - This looks like a cooler interfact to write commit messages
8dd76fc My first commit
SMA:my_git_project danny$ git rebase -i HEAD~2
[detached HEAD 30dc1fa] Squashed last two commits
Date: Sun May 17 00:35:33 2015 +0530
3 files changed, 2 insertions(+), 4 deletions(-)
Successfully rebased and updated refs/heads/squash.
SMA:my_git_project danny$ git log --oneline
30dc1fa Squashed last two commits
083e7ee Added yet another test
49a6bec Added more tests
5199b4e ERROR COMMIT: Introduced error in sum.py
b00caea Added tests.py
b117516 Dummy Commit after adding sum.py
7d1b1ec Added sum.py
b198692 Cleaned junk
7ac171f Made some change to myfile2
cafb55d Merge commit '5ef655a4caf8'
cc48fb3 Added lines 1 and 3 using add -p
5ef655a Fixed conflict from another_feature branch
96f7c5e Another change in the master branch
7534bc2 Some change in the master branch
49ed357 Added another feature
7e0eea2 Removed line
f87da5 Dummy change
f934591 - Changed two files - This looks like a cooler interfact to write commit messages
8dd76fc My first commit
SMA:my_git_project danny$

```

Aborting a Squash

If a squash operation gets overwhelming, you can safely run `git rebase --abort` to get back to the pre-squash state.

Squash Modifies the Branch History

A squash operation changes the history of your branch. If you need to push your changes after a squash operation, you need to use the `-f` option, or your push will be rejected.

Stash Changes

Imagine a situation where you're working on a bug or a feature, and many files have been edited since the last commit. However, you need to switch branches to work on something else, or you need to demonstrate the state of the repository at the last commit to your boss. You can't commit your current changes, as they're not complete yet. How do you solve this problem? `stash` allows you to save the changes you've made in your repository and revert back to the state of the last commit. At a later stage, you can get back your changes if you wish. To stash uncommitted changes, run the following command:

```
git stash
```

You can check the list of stashes in your Git repository by running the following:

```
$ git stash list
stash@{0}: WIP on master: 8a15b20 New Message
stash@{1}: WIP on master: 8a15b20 New Message
```

In the code above, note the serial numbers associated with each stash, which Git uses to identify it. The commit hash and message refer to the last commit of the active branch when you stashed the changes.

To apply the changes that were stored in the last stash, you can use the following command:

```
git stash apply
```

To restore an old stash, you need to mention the serial number next to the stash in the list of stashes:

```
git stash apply stash@{1}
```

A stash can only be applied if no files have been modified since the last commit. To apply multiple stashes, you first need to commit the changes from a stash. Applying a stash may raise a conflict if a file in the stash has since been modified in a commit.

stash Untracked Files

The `stash` command stashes the changes that have been made to tracked files only. If you want to add an untracked file to the stash as well, just start tracking it with `git add` before running the `stash` command.

In newer versions of Git (1.7.7+), you can add the `-u` option to stash untracked files without tracking them.

Advanced Use of add

In [Chapter 6, “Correcting Errors”](#), we saw that we can instruct Git to track a new file, or stage changes to a modified tracked file, using the `add` command. In this section, we’ll go a step further and see how we can stage only a part of our modifications to the same file.

It’s generally a good idea to associate a commit with a single bug fix or feature, as commits can then be used to separate different logical ideas. If you’ve solved two bugs by changing parts of the same file and want those changes to appear in different commits, you can do so as follows.

To simplify the process, I’ll add three lines at three different positions in the same file and view the changes that I’ve just added (Figure 7-6):

```
git diff
```

```

Dadas-MacBook-Air:my_git_project donny$ git diff
diff --git a/sum.py b/sum.py
index 085bee..179e125 100644
--- a/sum.py
+++ b/sum.py
@@ -1,12 +1,17 @@
 #add_two_numbers.py
+
+# First Comment
+
 def add_two_numbers(a, b):
     """
     Function to add two numbers
     """
+
+ # Second Comment in between
     addition = 0 + b
     return addition

 if __name__ == '__main__':
     a = 5
+
+ # Third Comment line added
     b = 7
     print add_two_numbers(a, b)
Dadas-MacBook-Air:my_git_project donny$ █

```

Let's say I want to add the second line among the three added lines to my commit. We can start the process with `git add`. Note the `-p` postfix to the add command to initiate this process:

```

>git add -p
diff --git a/tests.py b/tests.py
index 3a722f0..57e1cfc 100644
- a/tests.py
+++ b/tests.py
@@ -3,15 +3,15 @@ import unittest
 from sum import add_two_numbers

 class TestsForAddFunction(unittest.TestCase):
-
+ # First comment
     def test_zeros(self):
         result = add_two_numbers(0, 0)
         self.assertEqual(0, result)
-
+ # Second comment
     def test_both_positive(self):
         result = add_two_numbers(5, 7)
         self.assertEqual(12, result)
-
+ # Third comment
     def test_both_negative(self):
         result = add_two_numbers(-5, -7)
         self.assertEqual(-12, result)
$ Stage this hunk [y,n,q,a,d,s,e,?]? s

```

Git has clubbed all the changes together into a “hunk”. A **hunk** is a group of changes in a file. Notice that Git now asks us to enter an option. These are the options and their uses:

- y: stage the hunk
- n: don't stage the hunk
- e: edit the hunk
- d: exit the process
- s: split the hunk

In this case, we want to add only the second line, but because all three lines are a part of the same hunk, we need to split it:

```
$ Stage this hunk [y,n,q,a,d,s,e,?]? s
Split into 3 hunks.
@@ -3,7 +3,7 @@
 from sum import add_two_numbers

class TestsForAddFunction(unittest.TestCase):
-
+ # First Comment
  def test_zeros(self):
    result = add_two_numbers(0, 0)
    self.assertEqual(0, result)
```

After splitting the larger hunk, we're provided the first of the three smaller hunks. We wish to add only the second one. Therefore, we go to the next one by selecting option n:

```
$ Stage this hunk [y,n,q,a,d,j,J,g,/,e,?]? n
@@ -7,7 +7,7 @@
     def test_zeros(self):
         result = add_two_numbers(0, 0)
         self.assertEqual(0, result)
-
+ # Second Comment
     def test_both_positive(self):
         result = add_two_numbers(5, 7)
         self.assertEqual(12, result)
$ Stage this hunk [y,n,q,a,d,j,J,g,/,e,?]? y
```

Next, we're asked if we want to stage the second line. Therefore, we select option y, followed by option n for the third line. You can run `git status` to check how the repository looks:

```
$ git status
On branch master
Your branch is ahead of 'origin/master' by 1
commit.
  (use "git push" to publish your local commits)

Changes to be committed:
  (use "git restore --staged <file>..." to
unstage)
    modified:   tests.py

Changes not staged for commit:
  (use "git add <file>..." to update what will be
committed)
  (use "git restore <file>..." to discard changes
in working directory)
    modified:   tests.py
```

As you can see, the same file shows up in the list of modified files and in files staged for commit. This means you successfully staged a part of a modified file. You can proceed to commit your changes now.

Don't Commit with the -a Option

After staging a part of a modified file, you shouldn't commit the changes by postfixing -a. This would add the rest of the modified file too!

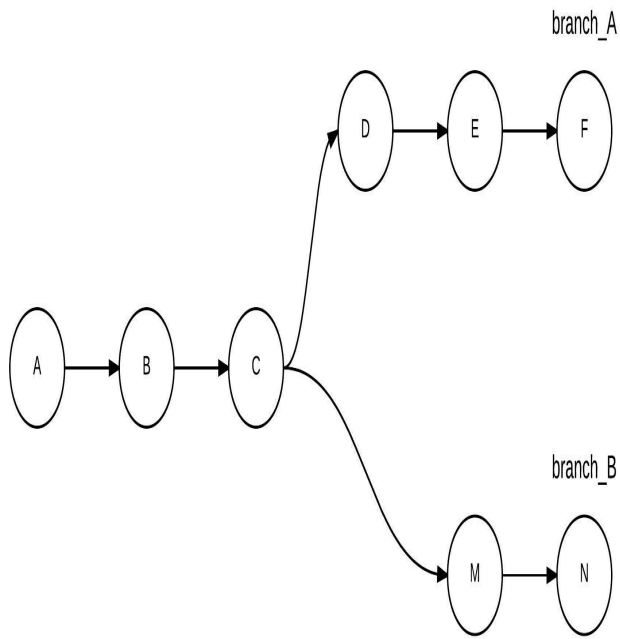
Cherry Pick

Let's say our work is progressing in two branches. If you want to merge a single commit from one branch into another, merge or rebase won't suffice. The `cherry-pick` command allows you to pick a certain commit from a different branch and merge it into your current branch. Just like in merging and rebasing, `cherry-pick` can also result in conflicts, which should be resolved as discussed earlier.

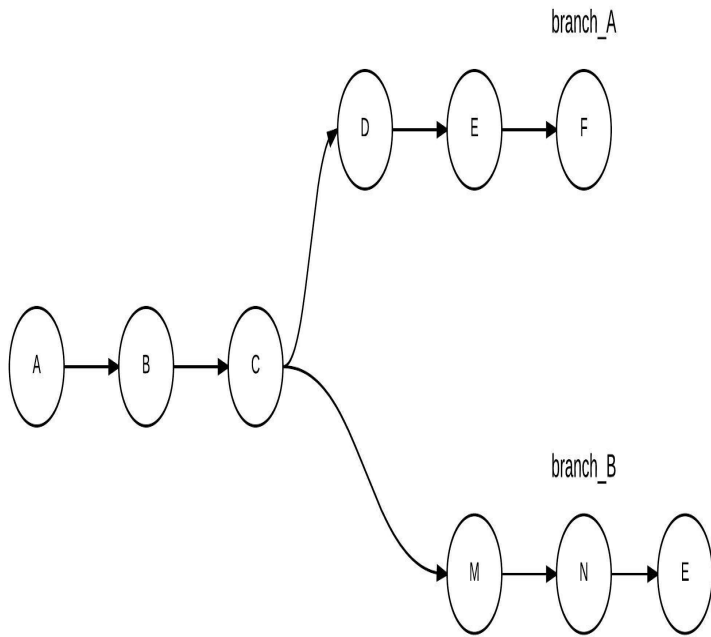
How Does cherry-pick Differ from merge or rebase?

In merge or rebase, you join your current branch with a different branch. All the commits of the other branch—that have happened since it diverged from your branch—appear in your branch after the merge. However, as the name suggests, you can pick a single commit from a different branch and make it appear in your branch using cherry-pick.

The idea of a cherry-pick is illustrated in Figure 7-7.



cherry-pick E from
branch_A to branch_B



To merge a commit 30dc1fa2d from a different branch to your current branch, run the following command:

```
git cherry-pick 30dc1fa2d
```

GitHub CLI

While we've looked at various advanced Git commands, you may have noticed that you still need to access GitHub's website to create [pull requests](#) or [raise issues](#). GitHub has an in-built bug tracker where you can create issues. An issue is a task, enhancement, or a bug in your project. GitHub has now launched a [new CLI tool](#) that allows you to perform GitHub-related operations from the command line.

To use the command-line tool, you need to download and install a GitHub client that enables the use of the `gh` command. On Windows, you can download the MSI installer from the [releases page](#).

On macOS, you can use Homebrew to install GitHub CLI:

```
brew install github/gh/gh
```

On Linux, visit the [releases page](#) and download the installer corresponding to your distro, and run the installation.

You can find detailed [installation instructions for GitHub CLI](#) on GitHub.

Navigate to a local clone of a GitHub repository, and run a `gh` command. The first time you run it, you're redirected to GitHub on the browser to authenticate the tool:

```
Notice: authentication required
Press Enter to open github.com in your browser...
Authentication complete. Press Enter to
continue...
```

There are two broad categories of commands that you can run: pull requests (the `pr` subcommand) and issues (the `issue` subcommand). To check the list of pull requests on your browser, run the following command:

```
gh pr list
```

To check the status of pull requests, use the status keyword:

```
>$ gh pr status

Relevant pull requests in sdaityari/my_git_project

Current branch
  There is no pull request associated with
  [sdaityari:2e-consolidated-first-draft]

Created by you
  #3 Added first draft of git dev cycle
  [sdaityari:git-dev-cycle]
  #2 Rewrite code [sdaityari:code-rewritten]
  #1 Added Git Workflow [sdaityari:new-git-
  workflows]

Requesting a code review from you
  You have no pull requests to review
```

To create a new pull request, run the following command:

```
gh pr create
```

It creates a pull request from the current active branch to the `master` branch. If your local clone is of a fork, the pull request is created on the `master` branch of the main repository.

After you run the command, you're asked to enter details of the pull request, like title and body. You can optionally

enter the text of the body in a text editor. Similarly, you can create, list and check the status of issues on a repository using the `issue` subcommand. For instance, the following command shows the status of all issues:

```
gh issue status
```

While Git in itself is powerful, completing actions on GitHub without leaving the terminal makes the process convenient and more efficient. You can check the latest commands for GitHub CLI in [its manual](#).

Conclusion

WHAT HAVE YOU LEARNED?

We've reached the end of another chapter! In this chapter, we discussed various commands and their uses to make your Git experience easier. In addition to GitHub CLI, here's a list of the commands we covered:

- `log`
- `shortlog`
- `reflog`
- `fsck`
- `rebase`
- `stash`
- `add`
- `cherry-pick`

You should try to incorporate these into your daily workflow to gain the most out of them.

WHAT'S NEXT?

In the next chapter, we'll explore how Git fits into your DevOps strategy and how to incorporate it into your continuous integration process.

Chapter 8: Integrate Git in Your Development Cycle

In earlier chapters, we looked at various Git concepts, managing a codebase in a team environment, and commonly used Git techniques to make your workflow efficient. These things help you become better at managing your code with Git, but one thing we haven't discussed yet is how to integrate Git into your development cycle.

This chapter focuses on using Git to bring efficiency into your entire software development cycle. If you're using Git for your personal projects, you may still find the concepts discussed here relevant.

Git and DevOps

How did DevOps come into being? Traditionally, there was a clear distinction between teams that were responsible for development, testing, deployment, and maintenance. While this distribution of functions was based on the skills required, teams realized over the years that inefficiencies were introduced due to the friction between these teams. This led to the birth of DevOps, a cross between development, operations and QA (quality assurance). **DevOps** is a set of guidelines that automates processes in the software development cycle, thus lowering the time to market. A DevOps engineer oversees the code release cycle and ensures that the process is smooth.

Though the stages in the DevOps cycle can vary, it generally involves the following seven steps:

- **Continuous development.** This involves planning a roadmap depending on the end-user's requirements, and then starting development.
- **Continuous integration.** This involves testing all code on every push, and merging with the main codebase if all tests pass. An additional code-review process may be included.
- **Continuous testing.** This involves testing the application in a live environment to verify the product's functionality.
- **Continuous monitoring.** In this phase, critical metrics of a product's output and usage are monitored in order to identify any anomalies.
- **Continuous feedback.** Feedback from earlier stages is used to update the roadmap and proactively include features and bug fixes.
- **Continuous deployment.** This ensures that the availability of the end product isn't affected by releases.
- **Continuous operations.** The objective of this step is the automation of the release process, which leads to a shorter development cycle.

The use of Git is critical in the early stages of the DevOps cycle. Git ensures proper code management, attribution, and integration with other stages. Git can trigger a set of processes at the onset of an action, like a code push or a pull request. Without the use of any version control, it would be difficult to create triggers based on code changes.

Using Git Hooks

While we've seen that Git is important to DevOps, this section deals with "Git hooks", a technique to seamlessly integrate Git with the DevOps cycle. A **Git hook** is a custom script that executes when a pre-defined action or event occurs. Git hooks are of two types: client-side and server-side. A client-side Git hook is reactive to local changes: it happens on actions such as a commit or merge. A server-side Git hook gets initiated on a network-based action, like a push to a remote.

As Git hooks are executable scripts, there's no limit to the actions you can perform with them. The simplest use

case of a Git hook is to send an email on every new commit. You could initiate unit tests to be run on every commit. You could raise a pull request on every push to a new branch.

Git hooks reside in the `.git/hooks` directory of your repository. When you initialize a Git repository, Git populates this directory with sample scripts with the `.sample` extension.

If you'd like to create a hook, remove the `.sample` extension from any of them. Since they're executed on an action or event, double-check the file permissions to ensure they're executable.

If you view the contents of any of these files, you'll notice that they're Bash scripts with an opening line of `#!/bin/sh`. While the sample files are Bash scripts, you can change the first line to change the language of the script. While local Git hooks generally work around changes (pre-commit, commit-msg, post-commit), server hooks (pre-receive, update, post-update) are mostly centered around updates. Here's a list of common hooks and when they're triggered:

- **pre-commit:** before a commit message is entered
- **commit-msg:** with the commit message (it's a path to a temporary file)
- **post-commit:** after a commit process is complete
- **pre-push:** before changes are sent to a remote
- **post-merge:** after a merge is complete
- **pre-receive:** before a push from a client
- **post-receive:** after a push from a client is completed

The most important Git hook is the pre-commit, where you can integrate unit tests:

```
#!/usr/bin/env bash  
  
echo "Running pre-commit Git hook to run unit
```



```
tests"
python ../../tests.py

# $? stores exit value of the last command
if [ $? -ne 0 ]; then
  echo "Tests did not pass! Aborting commit."
  exit 1
fi
```

While using Git hooks provides you with a lot of functionality to integrate with the software development cycle, it may be difficult for the average user to write custom scripts to achieve the objectives. However, you can utilize continuous integration (CI) tools to achieve this task.

Integrating Travis CI with GitHub

Continuous integration is the second step in the DevOps cycle, and it involves substantial use of Git. Continuous integration ensures a smooth transition between the code push and beta testing. It ensures that any new changes to the repository won't break things.

Travis CI is a popular, hosted continuous integration tool for GitHub repositories. It's free to use for open-source GitHub projects, with paid plans for commercial projects. You can only test repositories hosted on GitHub with Travis CI. However, you can create a workaround by adding submodules to test repositories hosted on other platforms.

Travis CI works like this: on every push to GitHub, the tool essentially clones the repository on a virtual machine on the cloud, installs the requirements on the fly, and runs pre-defined unit tests to determine if the new code breaks the existing functionality of your project.

GETTING STARTED WITH TRAVIS CI

To integrate Travis CI with your GitHub repository, you need to first create an account on [the Travis CI website](#). If you log in through your GitHub account, you can skip the additional step of linking GitHub with Travis CI.

After logging in, go to the [settings page](#) of your Travis CI account to view the list of public repositories in your connected GitHub account. Search the repository you'd like to link to Travis CI and select the toggle button next to the repository.



MY ACCOUNT



Shaunik Daityari

Shaunik Daityari

@sdaityari [@sdaityari](#)

Sync account

[Repositories](#) [Settings](#)

ORGANIZATIONS

We're only showing your public repositories. You can find your private projects on [travis-ci.com](#).



Legacy Services Integration



a11yDropdown



django-hello-world



MISSING AN ORGANIZATION?

[Review and add your authorized organizations.](#)

web.accessibility



web-scraping



Next, you need to add a configuration file to the root directory of your repository. Name the configuration file `.travis.yml`. Here are the minimal settings for your Travis CI configuration file:

```
language: python
python:
  - "3.7"
  - "3.8"

# command to install dependencies
install: "pip install -r requirements.txt"

# command to run tests
script: python tests.py
```

First, you instruct Travis CI to test each build on Python and then specify the versions 3.7 and 3.8. Note that, for each version you specify, a new build will be created and tested, thereby increasing the time to perform the test. Next, you provide the command to install the dependencies of the repository. Finally, you provide the command to run the unit tests, against which the push will be evaluated.

After you've created the configuration file, push a commit to the repository to trigger a Travis CI build. Interestingly, without the configuration file in your repository earlier, the commit that introduces this file in the repository would also trigger a build on Travis CI.

TRAVIS CI BUILD RESULTS

How long Travis CI takes to run tests on a single commit depends on a few factors:

- how many versions your build needs to be tested on
- how many requirements your repository has
- how detailed your unit tests are

Once your tests are complete, you receive an email with the results of the test. You can also view the status of your recent tests on [your Travis dashboard](#).

Travis CI | Dashboard | Changelog | Documentation | Help

Search all repositories

sdaityari/web-scraping build passing

My Repositories + | Current | Branches | Build History | Pull Requests | More options

sdaityari/web-scraping #1

Duration: -

IMCITRcodes/django-files #183

Duration: 56 sec

Finished: about 20 hours ago

master Added empty tests and Travis.yml #1 received

Cancel build

Commit e7e651

Compare e4c381c...e7e651

Branch master

Shaunik Dahiya

Build jobs | View config

Job ID	Language	Environment	Status	Actions
# 1.1	Python 2.6	no environment variables set	-	⌵
# 1.2	Python 2.7	no environment variables set	-	⌵

If the tests are still running, Travis CI shows the live information about the test. You can cancel the build if it's taking a long time, or even restart a build after it has

completed. You can view the configuration file from within the build page to ensure the tests ran the way you intended them to.

ADVANCED CONFIGURATION SETTINGS

Now that we've run a successful test on Travis CI, let's examine some advanced features of Travis CI and their use cases. (You can view a [full list of Travis CI configuration settings](#) in the Travis documentation.)

- Imagine you intend to run Git-related operations in the build. If a unit test fails, you'd like to automate the process of `git bisect` within this build to find out which commit introduced a bug. To go ahead in this scenario, it isn't necessary to download the full history of the project. With the `depth` option, you can set the number of Git commits to clone through the following setting:

```
git:
  depth: 3
```

- Next, while unit tests test how your code runs, your project may have detailed requirements before installation of the prerequisites. In such cases, you may feel the need to run custom configuration commands on the server before running your project's build. You can also set a list of commands to run before installing the prerequisites. You can pass terminal commands as a list to run in this setting. The `echo` command in the example below prints a message in your log:

```
before_install:
  - echo "running before_install
  commands"
  - python -c '# some_python_command'
  - echo "pre-installation config
  complete"
```

- On similar lines, you can set a list of commands to run after the build is complete. For instance, did the right version of your project execute? Do you want to check if data integrity is maintained on your server? The following terminal commands are sequentially run after the build is complete:

```
after_script:
  - echo "running after_script commands"
  - python -c "#some_python_command"
  - echo "after_script commands executed"
```

- It's possible that, during the process of a build, you may only be interested in how the current version of your project runs. To make the debugging process easier, you may wish to remove Git-

related messages from your logs altogether. To ignore Git-related messages in your log files, you can set the `quiet` option in `git` to `true`:

```
git:
  quiet: true
```

- While multiple developers work on their own features, you probably won't want to trigger a build every time a developer sends a push. You can safelist and blacklist branches to trigger builds for, or ignore changes to specific branches:

```
# blacklist
branches:
  except:
    - some_experimental_feature

# safelist
branches:
  only:
    - master
    - dev
```

- While safelisting branches makes sense for a smaller team, you may still be dealing with a large number of push operations. There's an option to ignore build triggers with every push altogether — by setting up `cr` on jobs on Travis CI to run a set of tasks periodically. Go to the **Cron Jobs** section on the **settings** tab of any repository to set up `cr` on jobs.



Cron Jobs

Branch: master

Interval: monthly

Options: Always on

Submit Query

- Travis can integrate with Docker to create a custom build environment to test your code on. You need to enable Docker under `services` and set up custom Docker commands in the `before_install` section:

```
services:
  - docker

before_install:
  # Custom docker commands
```

Conclusion

WHAT HAVE YOU LEARNED?

In this chapter, we covered the integration of Git with DevOps to enable you to bring more efficiency into your software development cycle. We first looked at custom scripts that can be used through Git hooks for various events and actions in Git. We then discussed how to integrate Git with Travis CI to perform the task of continuous integration, one of the critical steps in the DevOps cycle.

WHAT'S NEXT?

In the next chapter, we'll look at some GUI tools for Git, examining how they handle the commands we've already discussed.

Chapter 9: Git GUI Tools

Until now, we've performed all our Git-related actions through the terminal, looking in detail at what each command does. The advantage of terminal commands is that they work across all platforms.

In Chapter 1, I mentioned that there are various GUI (graphic user interface) tools that can be used instead of the terminal. Although GUI tools can appear to make life simpler, applications can use differing UIs, terminology, and Git concepts. GUI tools also lack some of the power and features of the terminal, and terminal commands execute more quickly. For these reasons, I've avoided the use of Git GUI tools so far.

In this chapter, we'll look at the GUI tools that serve as Git clients. First, we'll review [GitHub Desktop](#), GitHub's own GUI tool, and then Atlassian's [Sourcetree](#). Both of these applications have macOS and Windows versions, but neither supports Linux. Other popular GUI clients are [Tower](#) (macOS), [GitBox](#) (macOS), [SmartGit](#) (Windows, macOS, Linux), [Fork](#) (Windows, macOS, Linux), and [GitKraken](#) (Windows, macOS, Linux). All of these applications are either free or have free trial versions.

You can also use Git's capabilities through extensions in your text editor. [Atom](#), a text editor by GitHub, has built-in [Git and GitHub functionality](#). Sublime Text's [Git Integration package](#) enables the use of many Git-related features from within the confines of the text editor. Visual Studio's [version control tools](#) enable you to integrate with multiple version control systems to manage repositories hosted in remote locations.

GUI tools are an attractive option to many developers, as they provide an easy interface for managing a project with Git. Though we arguably gain a deeper understanding of Git by learning it through the command line, GUI tools have their place, especially in simple situations. One issue with using GUI tools is that it's easy to forget proper Git commands. This is problematic if you find yourself in an environment without GUI software, or if you need to run emergency commands from the command line—such as working on a remote server. I suggest using a combination of GUI tools and the command line, utilizing the advantages of each.

I'll now look at GitHub Desktop and Sourcetree in turn, evaluating their features and ease of use. Note that Git GUI tools frequently change their UI, so the screenshots below may differ a little across versions and operating systems.

GitHub Desktop

Let's first take a look at the GUI client of GitHub itself. It supports both Windows and macOS. The Windows and macOS versions of GitHub's previous clients differed, but in August 2015 GitHub launched [GitHub Desktop](#) as a new, unified client for both platforms.

After installation, you should add your GitHub account details.

Not Just for GitHub

You can manage other local Git repositories with GitHub Desktop too, but it's tailor made for GitHub repositories. Although a bit confusing, you can even [manage Bitbucket repositories](#) through the GitHub GUI tool!

When you successfully log in to your account, all your repositories are linked to your GUI tool. You can create a new repository through the **New Repository...** option

from the **File** menu. You can also see a list of your repositories under the **Clone Repository...** option.

Clone a Repository



GitHub.com

GitHub Enterprise Server

URL

Filter your repositories



Your Repositories

 sdaityari/a11yAccordeon

 sdaityari/a11yDropdown

 sdaityari/ATutor

 sdaityari/atutor-api

 sdaityari/blog

Local Path

/Users/shaumik/Documents/GitHub

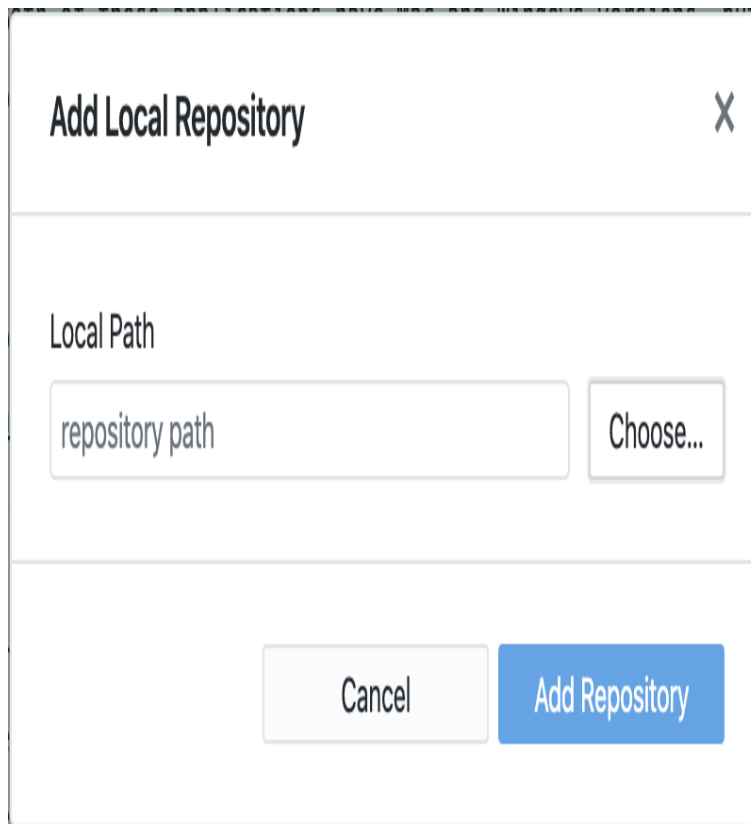
Choose...

Cancel

Clone

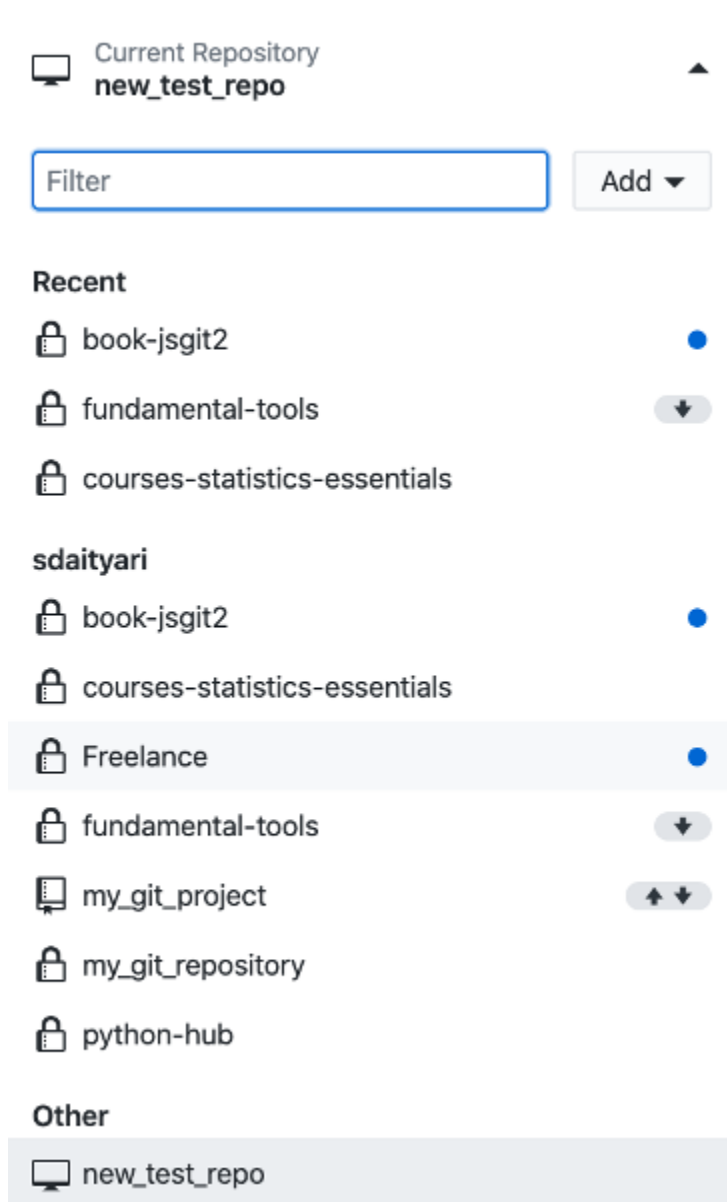
Select the repository you want to clone, and click on **Clone** to clone it.

Alternatively, you can add a local Git repository by choosing the **Add Local Repository...** option in the **File** menu. You're then asked to select the path to an existing Git repository on your local system.



The image shows a dialog box titled "Add Local Repository" with a close button (X) in the top right corner. Below the title bar, the text "Local Path" is displayed. Underneath, there is a text input field containing the placeholder text "repository path" and a "Choose..." button to its right. At the bottom of the dialog, there are two buttons: a "Cancel" button on the left and an "Add Repository" button on the right.

Once you've added your repository, you'll notice that it's now listed among the tracked repositories in the repositories list (**Current Repository** tab, top-left). If you add a GitHub repository, it will be listed under your username on GitHub, whereas if you add a local repository, it will be listed under **Other**.



Once a repository is selected, the commits in the current branch are listed. The UI resembles the GitHub website. If any commit is selected, the commit details are shown too. The workflow in the current branch is shown at the top.

On selecting **Show History** from the **View** menu, you're shown the commits in the active branch. On selecting a specific commit, you're shown the changes that were made in that commit:

Current Repository: my_org_project | Current Branch: master | Push origin: Last fetched just now

Changes | History

Updated CSV file

Select Branch to Compare...

Shaunik Dajpari committed 5762001 | 1 changed file | Hide Whitespace

Merge branch 'master' of https://github.com/shaunikdajpari/my_org_project
Shaunik Dajpari committed just now

data.csv	@@ -1,6 +1,6 @@
1	1 type,amount
2	2 labelA,4

Added new CSV file
Shaunik Dajpari committed Oct 8, 2019

3	-labelB,2
4	3 labelC,7

Updated CSV file
Shaunik Dajpari committed Apr 26, 2019

5	4 labelD,5
6	5 labelE,6
6	+labelF,1

Update data.csv
Shaunik Dajpari committed Sep 2, 2018

Added csv data
Shaunik Dajpari committed Sep 2, 2018

Added yet another test
Shaunik committed May 10, 2015

Added more tests
Shaunik committed May 10, 2015

ERROR: COMMIT: introduced error in sum.py
Shaunik committed May 10, 2015

Added tests.py
Shaunik committed May 10, 2015

Let's move on from comparing branches to creating or changing a branch. To create a branch, click on the **Current Branch** tab and enter the name of the new branch. It will be created from the current active branch.

Current Repository: my_git_project | Current Branch: new_feature | Fetch origin: Last fetched just now

Changes | History | **Branches** | Pull Requests 1

Select Branch to Compare... | new_branch_github_desktop | New Branch

Removed line
Shaunik committed May 15, 2014

Dummy change
Shaunik committed May 15, 2014

- Changed two files - This looks like a cooler inte...
Shaunik committed May 6, 2014

My first commit
Shaunik committed May 6, 2014

@@ -1,2 +1 @@
1 This is another file! Changing this file too.
-Another change

Sorry, I can't find that branch
Do you want to create a new branch instead?

Create New Branch

ProTip! Press **⌘ + Ⓞ + N** to quickly create a new branch from anywhere within the app

To change your current active branch to a different one, simply select a new branch from the list of branches.

On the top right of the window, there's the **Fetch Origin** option, which gets the latest commits from your origin remote. You can create a pull request from within the GUI client by first comparing two branches and then creating the pull request, just like you do on the GitHub website.

Any changes made to the repository are visible in the **Changes** tab (top left). It lists the changes in the files, but note that there's no mention of the term "staging". You simply select the files you want to include in the commit and add a commit message before committing the changes, which makes the process simpler for beginners.

Current Repository: my_git_project
Current Branch: another_feature
Fetch origin: Last fetched 6 minutes ago

Changes 1 | History | my_file

1 changed file	@@ -1,3 +1,4 @@
1	1 This is some information!
2	2 1
3	3 I am changing the content of this file.
4	+2

Commit Message

Description

Commit to another_feature

Once you've committed the changes, the **Fetch Origin** button changes to **Push Origin**, which first fetches commits from the `origin` and then pushes your new commits. On pushing to a branch of `origin` that's not the `master` branch, a **Create Pull Request** button comes up, which redirects you to the appropriate link on the GitHub website.

GitHub Desktop tries to simplify the process of source code management, which is good for a beginner who's trying to learn Git. Let's now explore Sourcetree, which has a wider range of functions.

Sourcetree


Sourcetree is a GUI client developed by Atlassian. It's compatible with repositories managed by both Git and Mercurial, another distributed VCS. Sourcetree can use the version of Git already installed on your local system, or a version that's bundled with Sourcetree itself. You can download and install the application from [the Sourcetree website](#).

Sourcetree offers a wider range of features than GitHub's tool, and gives you more control over your repositories. Its various options also better match the corresponding terminal commands.

During installation, you're invited to add details of any accounts you hold at code sharing websites like GitHub and Bitbucket. If you skip this step, you can add accounts later in the **Accounts** tab of **Preferences**.

Accounts

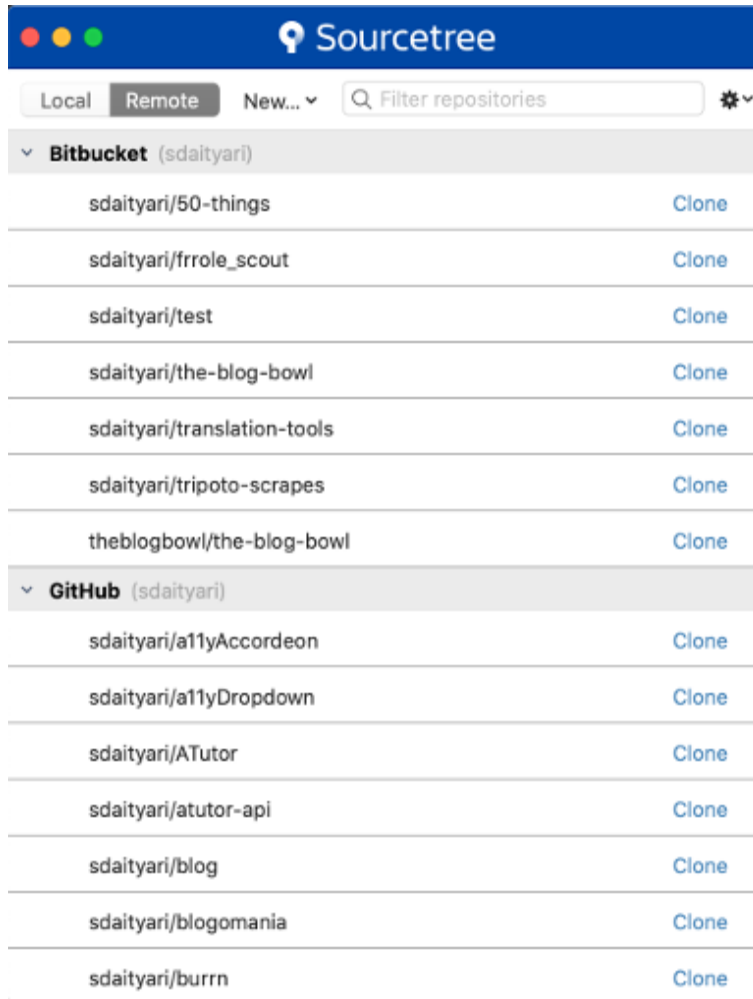
General Accounts Commit Diff Git Mercurial Custom Actions Update Advanced

 sdaityari

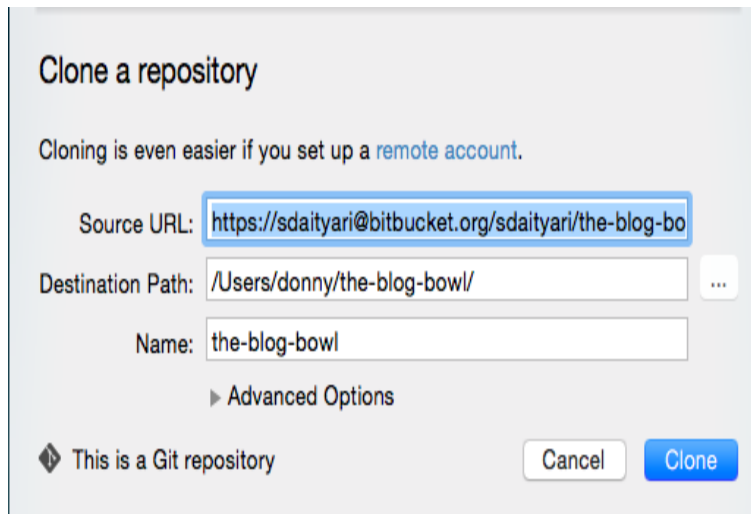
Bitbucket	OAuth	HTTPS	Default
-----------	-------	-------	---------

Add... Edit... Remove... Set Default...

After adding your cloud accounts, you're shown the list of repositories in your connected accounts.



The repositories listed here are present only on the cloud, so they need to be cloned before you can start working on them locally. Click on the **Clone** link on the right of any repository to clone it.



After confirming the details, the remote repository is cloned to the location you specified in the last step.

Alternatively, you can add a local repository to Sourcetree by clicking on the **+New Repository** button. Once you've added a repository, a new window opens with the details of the repository.

my_git_project (Git)

Commit Pull Push Fetch Branch Merge Stash View Remote Show in Finder Terminal Settings

WORKSPACE All Branches Show Remote Branches Ancestor Order Jump to:

Graph	Description	Commit	Author	Date
	↳ another_feature ↳ origin/another_feature Commit Message	d04ec3f	Shaumik Daitya...	Today at 1:24...
	↳ master 2 ahead Merge branch 'master' of https://github.com/sdaityani/my_git_p...	399f698	Shaumik Daityani...	Today at 1:14 AM
	↳ origin/master ↳ origin/HEAD Added new CSV file	25313e5	Shaumik Daityani...	08-Oct-2019 at...
	Updated CSV file	676390f	Shaumik Daityani...	26-Apr-2019 at...

another_fea... Sorted by path Search

master 2↑ posts.csv

new_feature

File contents Reverse hunk

```

1 + type, amount
2 + Post A, 4
3 + Post B, 2
4 + Post C, 7
5 + Post D, 5
6 + Post E, 6

```

Merge branch 'master' of https://github.com/sdaityani/my_git_project

Commit: 399f69812cd8b6491b7228f0a65290e118ec86fd (399f698)

Parents: [676390fd09](#), [25313e5016](#)

Author: Shaumik Daityani <sdaityani@gmail.com>

Date: 9 February 2020 at 1:14:55 AM IST

Labels: master

As highlighted in the image above, the window has three parts: the top menu, the left menu, and the main body. The top menu contains buttons that perform important actions in Git. The left menu lists the branches, remotes, stashes and submodules. The main body contains the list of commits in the active branch and the details of each commit.

If you look at the top menu (Figure 9-11 below), you'll notice that it contains buttons for performing basic Git actions like commit, pull and push. There's also an option to open up a terminal in case you want to run a custom command.



The **Branch** button helps you checkout to a new or an existing branch.

my_git_project (Git)

Commit Pull Push Fetch Branch Merge Stash View Remote Show in Finder Terminal Settings

WORKSPACE

All Branches

Graph Description

New Branch Delete Branches

Jump to:

File status

History

Search

BRANCHES

another_fea...

master 2↑

new_feature

TAGS

REOTES

origin

STASHES

SUBMODULES

SUBTREES

posts.csv

Sorted by path

Commit: Working copy parent

Specified commit:

Pick...

Checkout new branch

Cancel Create Branch

Merge branch

https://github.c

Commit: 309f09812cd8b6491b72280a65290e119ec86fd (309)

Parents: [67c390fd09_25313e5016](#)

Author: Shaumik Dahiya <sdahiya@gmail.com>

Date: 9 February 2020 at 1:14:55 AM IST

Labels: master

Author Date

Shaumik Dahiya... Today at 1:24...

Shaumik Dahiya... Today at 1:14 AM

Shaumik Dahiya... 08-Oct-2019 at...

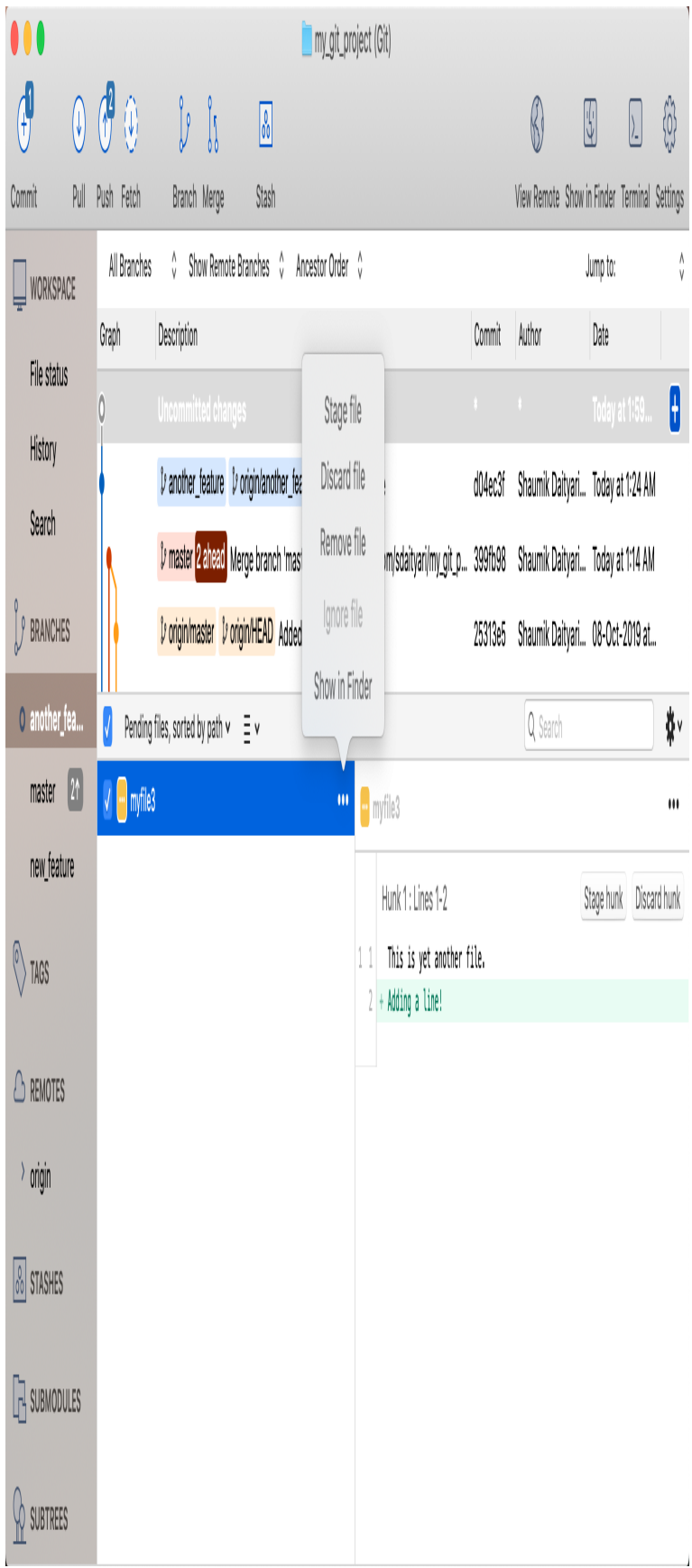
Shaumik Dahiya... 26-Apr-2019 at...

Q Search

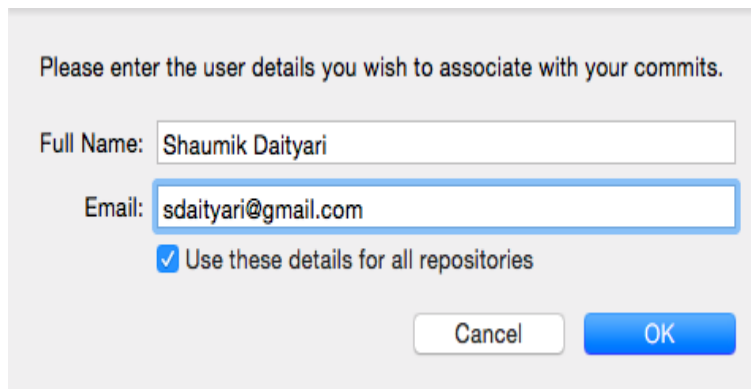
Reverse hunk

6 + Post E,6

When you make changes to any file, the list of changed files pops up in the space for unstaged files (Figure 9-13 below). You can stage them by clicking the **Add** button on the top—after which they appear in the staged list. You can also remove staged files using the **Remove** button at the top.



Once you're ready to make a commit, click on the **Commit** button. For your first commit, you're asked to nominate a name and email address to be associated with your commits (Figure 9-14 below). This is similar to setting the global configuration settings through the terminal. From now on, your email address and name will be associated with this commit, as well as any future commits.



Please enter the user details you wish to associate with your commits.

Full Name:

Email:

Use these details for all repositories

After adding your name and email, you're asked to add a message describing your commit.

my_git_project (Git)

Commit Pull Push Fetch Branch Merge Stash View Remote Show in Finder Terminal Settings

WORKSPACE Pending files, sorted by path Search

File status

- Staged files
 - myfile3
- Unstaged files

History

Search

BRANCHES

- another_fea...
- master 2↑
- new_feature

TAGS

REMOTES

- origin

STASHES

SUBMODULES

SUBTREES

Hunk 1: Lines 1-2 Unstage hunk

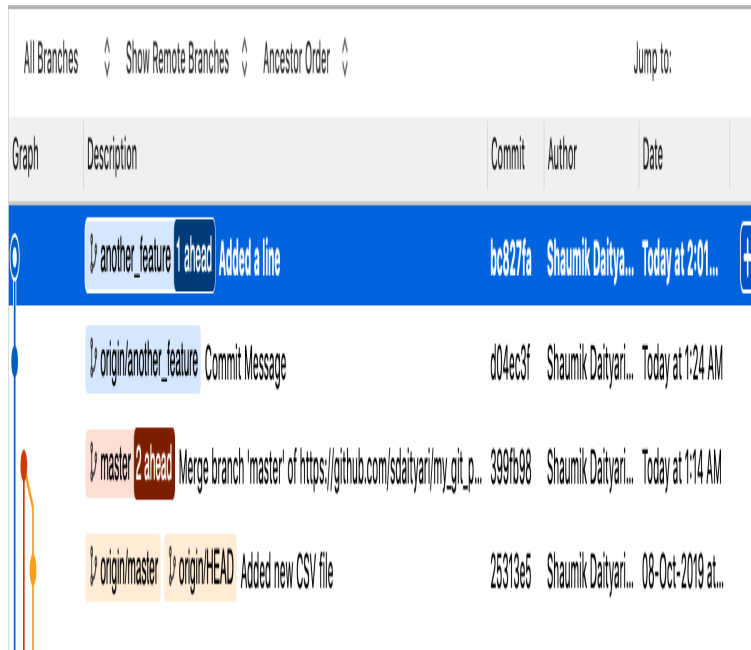
```
1 1 This is yet another file.
2 + Adding a line!
```

Shaunik Daityari <stdaityari@gmail.com> Commit Options...

Push changes immediately to origin/another_feature

Cancel Commit

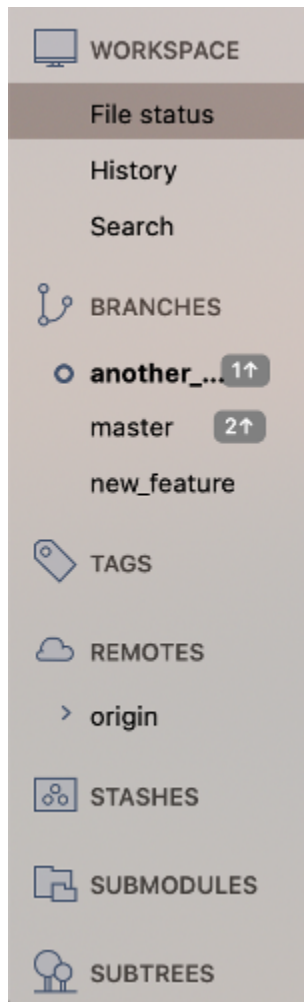
After a successful commit, notice the state of the repository and the change in the branch workflows: the blue color shows the current commit—which hasn't been merged with `origin/master`, denoted by yellow.



You can add or remove branches by clicking the **Branch** button in the top menu. You can force delete a branch even if it hasn't been merged yet, as shown in Figure 9-17 below. (This is analogous to the `-D` option in the terminal.) You can merge branches through the **Merge** button in the top menu. If you want to merge `branch_A` into `branch_B`, make sure `branch_B` is active when you perform the merge operation.



Let's now have a look at the left menu, shown in Figure 9-18 below. It shows a list of branches, tags, remotes, stashes and submodules.



In this case, `master` and `gh-pages` are the two branches, and `origin` is the only remote. We also have one stash created on the `master` branch, which is shown in the screenshot above. Sourcetree's stash option is a powerful, easy-to-use feature. You can apply any stash to your HEAD, with the option of keeping or removing the stash. Submodules are Git repositories within a parent repository. We haven't covered submodules in this book. This repository uses a `google_app` submodule.

In addition, commit-based actions like checking out to the commit, cherry picking or creating a patch can be performed by right-clicking on a commit, as shown below.

my_git_project (Git)

Push Fetch Branch Merge Stash

All Branches Show Remote Branches Ancestor Order

Graph	Description
Sorted by path	
myfile3	
	Added a line
Commit: bc827facddc8c7e23a3705	
Parents: d04ec3fa6e	
Author: Shaumik Daityari < sdaityari@gmail.com >	
Date: 9 February 2020 at 2:01:24 AM IST	
Labels: HEAD another_feature	

- Checkout...
- Push revision...
- Merge...
- Rebase...
- Rebase children of bc827fa interactively...
- Tag...
- Sign...
- Bookmark...
- Archive...
- Branch...
- Reset another_feature to this commit
- Reverse commit...
- Create Patch...
- Cherry Pick
- Copy SHA-1 to Clipboard
- Custom Actions ▶

Sourcetree versus GitHub Desktop

Both Sourcetree and GitHub Desktop are free to use.

Sourcetree has a lot of features, with an information-rich display that directly relates to Git's terminal commands. Desktop, on the other hand, focuses more on bridging the gap between a local GitHub repository and the GitHub website, often substituting standard Git terms and processes with easier terms for beginners. It eases the process of hosting your repositories on GitHub, but makes it difficult—though not impossible—to host your repository elsewhere.

Finally, Desktop simplifies the whole process by cutting down on certain features, whereas Sourcetree offers a full-featured dashboard that might be overwhelming for beginners. I encourage you to try both GUI tools, perhaps along with a few more listed at the beginning of this chapter, to work out which best suits your needs.

Conclusion

In this chapter, I reviewed two GUI tools for Git—Sourcetree and GitHub Desktop.

GUI tools are definitely useful. When using them, the history of a project, with respect to the different branches, is easily visualized. Even when you're working on a project, it's useful to graphically analyze the changes you've made before committing them into the project history. Even when you're reviewing the work of others, it's a good idea to use a GUI tool to quickly review the changes.

Even though I find GUI tools to be great, if you're a beginner, I'd still recommend you learn the terminal commands first. As I mentioned above, GUI tools aren't

cross platform, whereas terminal commands are. There's no single tool that works the same in Windows, macOS and Linux. Also, if you're working on a remote server (which is often a virtual machine), only command-line tools can help you work with Git. And knowing terminal commands will help you understand how these GUI tools work.

So for beginners and experienced users alike, I recommend using a combination of GUI tools and the terminal. Each has its pros and cons, which you'll discover through practice.

Chapter 10: Conclusion

As this book has guided you through the uses of Git, the focus has been on using it to manage a codebase. This is the most common use for Git, but certainly not the only one. In this concluding chapter, I'd like to discuss Git's meteoric rise, and then give you a glimpse of other innovative uses of Git, as well as its limitations, alternatives to Git, and what the future holds.

Git's Meteoric Rise

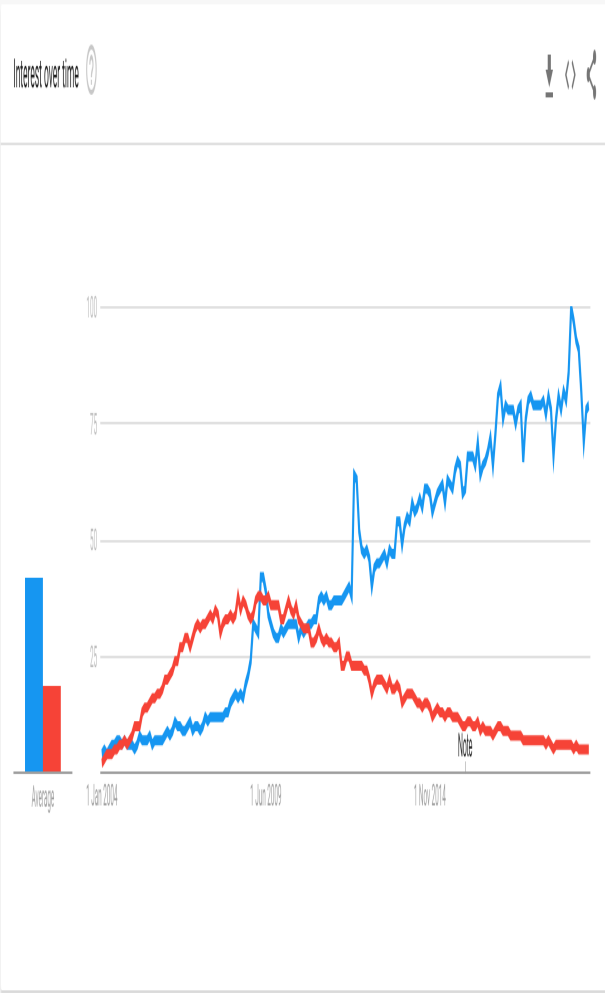
Back in 2009, over 57.5% of repositories used Subversion, whereas Git only had a 2.4% share of the SCM market, according to [the Eclipse Community Survey](#). Git's rise in popularity has been so tremendous that the question of a primary source code management system typically doesn't appear in recent developer surveys, with tools like fancy text editors and cloud-based IDEs taking over as questions. [Google Search trends](#) indicate that Git and Subversion had roughly the same interest around the year 2011 (Figure 10-1). However, since then, the popularity of Subversion has declined, whereas that of Git has grown steadily.

git Search term

svn Search term

+ Add comparison

Worldwide 2004 - present All categories Web Search



From these data sources, one thing is obvious: Git's meteoric rise proves it's doing something right. The future is definitely Git. Over the last decade, Git has solidified its position as the top choice for version control among developers. In my opinion, these are the aspects of Git that have most led to its rise:

- the concept of a distributed system
- a powerful branching system
- the introduction of a staging area
- its functional, easy-to-use cloud-based systems

Will Git Continue to be Popular in the Future?

A significant contributor to Git's rise is GitHub. While GitHub only started out as a company in 2008, Microsoft acquired it in 2018. This ensures that the future of GitHub is secure and that it will continue to scale based on demand.

Git is a great tool, and is often the first choice among developers. Naturally, organizations big and small choose Git to manage their projects, which is evident from the "Companies & Projects Using Git" section of the Git website.

However, one of Git's major failings becomes evident when it's used in very large projects: it doesn't manage large repositories in the most efficient way. How large are we talking about here? Facebook large. Facebook eventually shifted from Git to Mercurial, another distributed VCS. Let's look at why.

When engineers at Facebook extrapolated their future growth, they found that file status operations in Git would become a major bottleneck, as Git examines each file for changes. With thousands of commits every day, it

would have taken a few seconds to run even a `git status`. Integrating their own file monitor with Mercurial made for a much more efficient process, which is why Facebook shifted to Mercurial, and why their developers still contribute significantly to its development. However, even though Facebook shifted its main codebase to Mercurial, it's interesting that many of their important side projects like [React](#) and [RocksDB](#) are still managed through Git.

Prasoon Shukla, a Mercurial contributor, has described [the differences between how Git and Mercurial work](#), and why Mercurial is more efficient when you scale to the size of Facebook. However, few developers will ever work with repositories the size of Facebook.

In recent times, Git has made progress in the management of large repositories—both in terms of history and the size of files in a repository. If you have a codebase with a very large history, you can perform a shallow clone, which enables you to clone only a specified number of latest commits. For instance, if you want to clone only the ten latest commits from our dummy project, you can specify 10 using the `--depth` option:

```
git clone --depth 10
https://github.com/sdaityari/my_git_project
```

Previously, Git had only limited support for shallow clones, especially if your shallow history wasn't long enough. You often wouldn't be able to push from your shallow clone. However, recent versions (Git 1.9+) give you a greater ability to push and pull.

Another way of managing a large repository is to clone only a single branch. You can do so using the `--single-branch` option. To clone only the master branch of your dummy project, run the following:


```
git clone
https://github.com/sdaityari/my_git_project --
branch
└─ master --single-branch
```

Beyond Source Code Management

After reading this book, you hopefully feel very safe with Git. Once you create a commit, there's no way you can lose it (unless, of course, someone messes with the `.git` directory). You've seen the potential of Git. So isn't it natural that people are starting to use Git for tasks other than just managing code?

If you're a student, you can safely use Git to manage your academic assignments. As a tech blogger, I write my articles in Markdown in GitHub's text editor Atom, while maintaining a private repository on GitHub with all of those articles.

Git can be used to efficiently manage any project that contains text-based files. It's even used in publishing. In fact, the team behind this book worked on it using Git! I created pull requests on GitHub, where the editors then suggested changes.

Git can also be useful for designers. Even though Photoshop or CorelDRAW files aren't comparable to source code, they can be tracked by Git efficiently through LFS. And of course, any front-end code you write can be tracked using Git.

I mentioned in Chapter 1 that Google Docs is a good example of version control in action. There are many other applications built using Git with a similar premise of enabling change tracking. An example is the WordPress plugin [VersionPress](#), which tracks changes in a WordPress site using Git in the background.

The End

Although we've now come to the end of this book, you should be just beginning your journey with Git. Get out there and do some amazing things with version control! I hope you've enjoyed reading this book as much as I've enjoyed writing it.